503.4
Anomalies and Behaviors

SANS

Anomalies and Behaviors

Our first topic is network architecture for monitoring. You need to get the right equipment to capture traffic for analysis. When traffic volume was slower, you could simply put a sensor on a network and let it sniff traffic. But faster speeds and greater volume often require some type of device to capture the traffic before it is sent to the sensor.

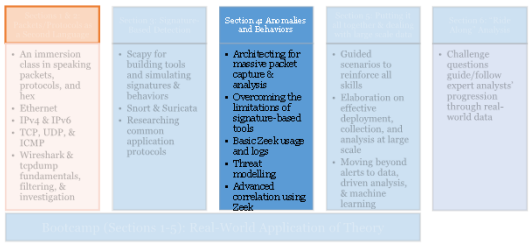
You need to decide the right place for the sensor(s) depending upon what you need to monitor and protect. We'll have a chance to examine the implementation issues that might arise when you do the hands-on exercise that follows this section. The decision of what needs to be monitored and how close to the source of the data that needs to be monitored is often overlooked or not given enough attention.

Following this, we will have a short discussion regarding the use of TLS and possible analysis and decryption strategies. We will also have time to try a lab wherein we will decrypt some TLS data to reveal the underlying HTTP2 traffic.

We then move on to Zeek, another open-source solution that has a different approach to monitoring traffic. Zeek provides a scripting language for the user to write code to analyze behaviors, especially those that represent events of interest or indicators of compromise for your specific network environment. We begin with a Zeek overview, talk about Zeek signatures, and then discuss the real power behind Zeek: its scripting language.

The demo files for Section 4 are found in the directory **/sec503/Demos/**.

This page intentionally left blank.



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

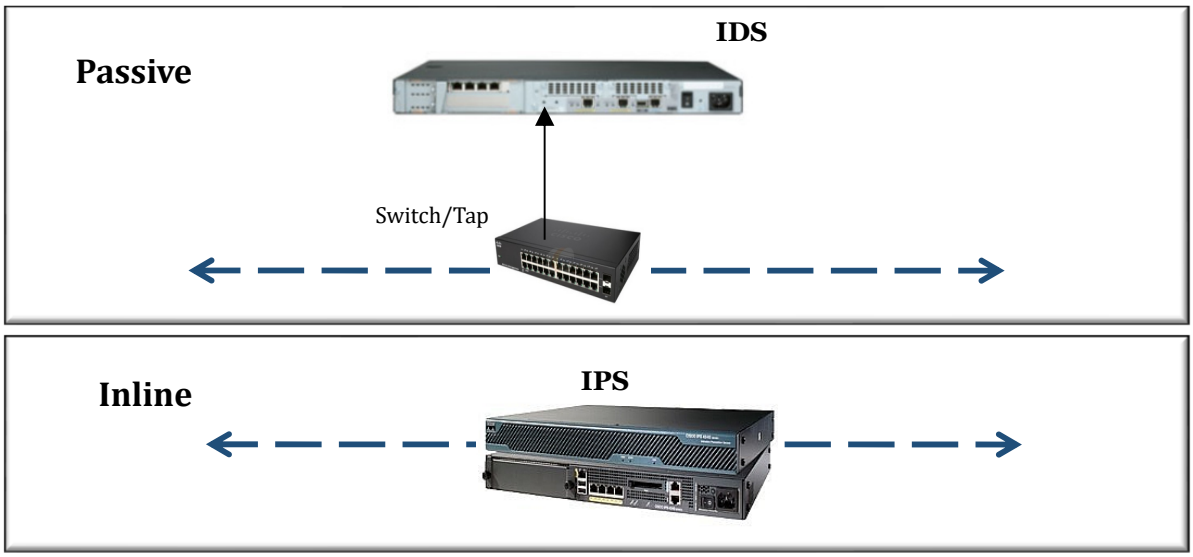
Anomalies and Behaviors

IDS vs IPS

Tapping Networks

Collecting Packets at Scale

Deployment: Passive Versus Inline Sensor



An IDS is deployed as a passive sniffing sensor that alerts and/or logs noteworthy traffic. Another deployment option is inline, where traffic actually traverses two interfaces of an IPS device. This method allows traffic to be blocked.

You or your Chief Security Officer (CSO), if you have one, will have to determine the mode of operation. Passive mode will not perturb the network flow of traffic, but the downside is that you learn after the fact that a noteworthy event has occurred. Inline provides better protection, but at a cost of potentially blocking good traffic for rules that are inaccurate or not precisely written. Always use caution when adding new rules to the inline configuration where prudence recommends placing the rules in logging mode before actually determining that they are production candidates for blocking.

If you have several different sensors on your network, you may consider a mix of passive and inline, where the passive ones are deployed in a generic mode to warn of potentially harmful traffic but not perturb it in any way. As you deploy more focused sensors for the assets deemed most valuable or sensitive, consider placing those in inline mode. The focused coverage allows you to be stingy about the rules you employ in the first place, and of those, the ones that you consider to be accurate enough to actually block traffic.

IDS Sensor Configuration

- Uses minimum of two NICs
- Passive listening interface to monitor network traffic:
  - Attached to network to be monitored
  - Has no IP address
  - No way to discover interface on monitored network
  - May have additional protection of blocking outbound traffic
- Communication interface to management console:
  - Connected to internal network, preferably an isolated network
  - Assign an internal network IP address

One of the first steps in building your IDS architecture is to properly configure the sensor to monitor traffic from one or more network interfaces, and at the same time, make sure that the IDS is not vulnerable to attack. Monitoring interfaces should be used solely for monitoring, not for management or for packet injection. The monitoring interface should be a separate adapter, connected to a separate network. It's recommended that your sensors all be interconnected on a private network, accessible only from the management console. This limits the number of individuals who can potentially communicate with these systems and also limits exposure should there be necessary delays in patching sensors.

The interfaces that monitor traffic need to be connected to physical ports on a device capable of forwarding the monitored traffic. These interfaces should not be assigned IP addresses, thereby making the sensor virtually impossible to attack. In addition, it need not and should not transmit any data; in fact, it is quite likely that the monitoring device to which it is connected will not accept packets.

These requirements are handled by using a sensor with a minimum of two NICs. The communication interface that reports to the management console needs to be accessible via the private monitoring network for troubleshooting, updating signatures and software, and general management. This interface should be connected to a port in a device such as a switch from a segment or VLAN connected to the private monitoring network, and this interface must have an IP address to be accessible.

Intrusion Prevention System

True IPS sits inline with two inline NICs, one management NIC

- All packets must pass through it.
- Packets are checked before being passed through it.

Some have built-in firewall or simply **are** firewalls… Palo Alto, for instance

- In this case, IPS features are almost always more money.

IPS sounds great in theory, but there are weaknesses:

- Must be very fast… and false positives can be career-influencing
  - So they tend to look for fewer things…
  - …and they tend to be less sensitive overall, thus minimizing false positives.

True network Intrusion Prevention Systems (IPSs) sit inline on the network, with one NIC identified as external and one as internal, often referred to as a *Port Pair*. The IPS can have several Port Pairs based on its hardware capacity. An IPS is a proactive defense mechanism that checks, tracks, and potentially blocks a flow containing malicious activity. If a flow is tagged as suspicious, all subsequent packets associated with that session can also be dropped with little effort. IPS systems can typically be configured with a set of IP addresses (acting as a sort of firewall/router) or transparently (without addresses).

One of the most important factors in evaluating an IPS is its latency. The high-end products tend to use a hardware parallel Application-Specific Integrated Circuit (ASIC) that introduces lower latency than software solutions to process traffic. Although inline, many IPS solutions can also be configured to monitor only while still alerting on suspicious traffic without blocking the attacks. This mode is intended to be used when deploying a new IPS, enabling you to observe its performance and customize it for your site.

An IPS may act as a proxy to ensure a three-way handshake has been completed before allowing the traffic through. Other IPS solutions include a packet-screening firewall to offer additional protection against known malicious sites (IP or range) or ports, which can be used with specific tasks to drop traffic from these pre-identified sources or unwanted ports.

One final and important point is that it is possible that rules or signatures can generate false positives because they may not be correct or may not be tuned for your specific site. The consequences of this are that valid traffic could be dropped. Most vendors have a smaller recommended set of rules or policies that should be applied when placing the solution in IPS versus IDS mode. Still, the best advice is to first place your IPS in monitor/alert mode, observe the results, and customize the rules and policies for your site. This may take several iterations to tune.

You should know that IPS solutions look for fewer signatures overall than do IDS devices. This choice is made since speed is critical, and to attempt to offset the effects of a false positive. A false positive from an IPS means your operations stop. You will only be permitted to halt operations once or twice before you are strongly cautioned by management. This can lead to a decision to run the IPS in monitor mode full time, which actually provides less coverage than a comparable IDS.

IPS Sensor Configuration

- Use minimum of three NICs:
  - Management interface on protected network reports to management console
  - Inline interfaces observe traffic as it passes through:
    - Inline of traffic flow
    - No IP addresses (transparent) or with addresses (firewall/proxy)
      - No way to discover transparent interface on exposed network
    - Can fail open or closed
    - Typically, the filters will either log, deny, or log and deny traffic at wire speed.
- IPS systems are often deployed in "Monitor" mode long term.
  - False positives are not an option! Tend to have fewer signatures

A task in building an IPS architecture is to properly configure the IPS to analyze traffic and protect the network segment efficiently against various attacks. Also, the IPS must report to a management console using some kind of secure communications protocol.

These requirements are handled by using a minimum of three network interface cards. The communication interface is used to report back to the management console and needs to be accessible via the network for troubleshooting, updating signatures and software, and general management of the IPS. This interface should be connected to a port in a device, such as a switch from a segment or VLAN connected to the internal network.

IPS devices tend to be somewhat less sensitive than their IDS counterparts. There are at least two reasons for this. First, the IPS *must* add minimal latency since it is sitting inline. Sacrifices are sometimes made in terms of how many signatures will be analyzed or which aspects of the behavior will receive attention so as not to introduce a delay into the traffic.

The second issue is that false positives are not an option! A false positive means that your security device has alarmed even though there is nothing really wrong. If we are talking about a device that sits inline and acts to block traffic that it believes is "bad," then a false positive means that our IDS is impacting production data.

This second issue is what leads many organizations to reconfigure the IPS to operate in monitor mode *only*. Clearly this isn't ideal. Since the system is configured to look for fewer bad things, it is more likely that bad things are happening, and you no longer know.

Special Purpose Versus Generic Sensor

When deploying monitoring tools, use targeted configurations where possible.

- Sensor monitoring screened network:
  - Nginx web server
  - Exchange mail server
  - Bind nameserver
  - Real-time monitoring requires only rules matching that profile.
- Immediately behind firewall:
  - Most rules enabled because we have far less control and knowledge
    - Even with strong configuration control, we all know things "creep in."
- Outside of firewall
  - Targeted real time; Could we reprocess with a full reset periodically for threat intel?

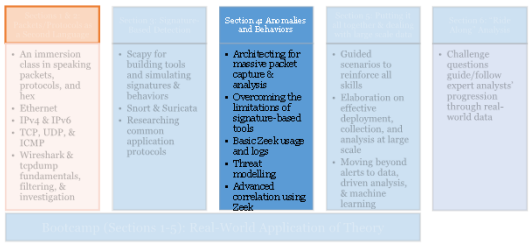
When deploying monitoring tools, we should think carefully about what our expectations are for each sensor. If we are deploying a sensor to perform monitoring of a screened network with public access, we would likely expect near real-time alerting for relevant services. What would make services relevant?

The rules on a focus sensor should be targeted to the traffic that you expect to see on your network. Say that the focus IDS runs Snort. If there are no FTP servers, you can comment out inclusion of rules or preprocessors associated with FTP. At the same time, if you are running Exchange, all of the Exchange-related rules should be enabled.

There are sensors that cannot be configured so precisely, so we use more of an "umbrella coverage" approach. For example, it is wise to have a sensor immediately behind the firewall. This "catch-all" sensor will need to have many more rule groups enabled. Even with strong configuration controls in an environment, we all know that additional protocols and services will creep into the network over time. Frankly, the same thing happens in cloud environments! Administrators will quietly enable additional services as they realize there is some production or development need that is undocumented.

What about a sensor sitting out beyond the firewall? There is tremendous benefit in such a device. While there are very few things that this device reports that we might need to react to in real time, it could be incredibly valuable to reprocess all the data that the host sees periodically, using it to develop internal threat intelligence and reputation information.

Consider: If you are using a paid threat intelligence feed, how much of that data is already available publicly? How much of the unique data is actually useful to you? On the other hand, what if you were to develop internal threat intelligence based on how aggressive hosts are? How valuable would it be to know that an aggressive host is very interested in you, but isn't showing up in the threat intelligence feeds?



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

IDS vs IPS

Tapping Networks

Collecting Packets at Scale

Instrumenting Your Network for Traffic Collection and Monitoring

- What traffic should you monitor?
  - What are the priorities for monitoring?
    - What compliance must be followed?
    - What type of risk is tolerable?
    - What type of budget is available?
- One approach: Information centric—begin instrumentation at the perimeter and move inward to priority aggregation points until you run out of money
  - Perimeter
  - Server
  - Sensitive data

Before any decisions are made about what solutions and hardware are required for monitoring, you have to assess what traffic needs to be monitored. In a utopian world where budget is unlimited and false positives are nonexistent, all networks would be monitored. More likely, we will only monitor critical networks.

What is critical data is unique to each individual organization, and the organization must define the priorities for monitoring. These priorities can be based on several different factors. First, are there compliance standards that mandate monitoring for specific data on your network? For instance, storage and transfer or Payment Card Industry (PCI) data requires tracking and monitoring of access to cardholder data.

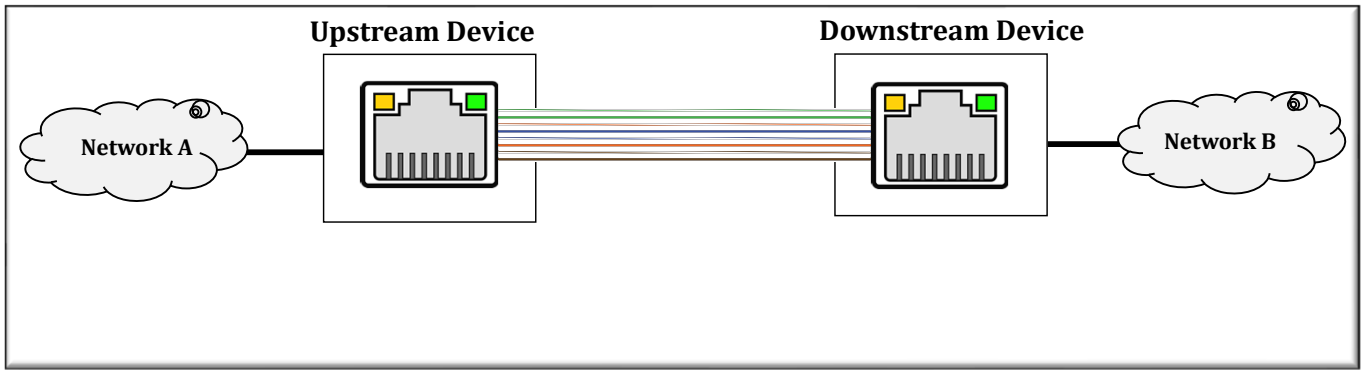
The best intentions of implementing extensive monitoring can be tempered by a limited budget. This requires a more frugal approach of acquisition and placement of monitoring solutions. One solution is to deploy focused scrutiny of sensitive data only, leaving ourselves blind to activities outside of these sensitive subnets. Some prefer to deploy a broad perimeter view of all data passing in and out of the infrastructure (north-south) while being blind to all east-west traffic in the organization.

When possible, start at the perimeter(s) of the network, where a sensor is placed either outside or inside the external firewall. Place it outside if you want to monitor traffic that exits through the firewall from the inside and traffic that fails to enter from the outside. This provides an added benefit of the sensor auditing firewall access control lists.

Another option is to place the sensor just inside the external firewall. This reduces the noise of the externally placed sensor false positives but may generate false positives of malicious outbound traffic that is blocked by the external firewall. A perimeter sensor provides general broad insight into what activity is entering and leaving your network.

Next, consider placing a sensor on any network where internet-facing servers are located. Moving inward, place a sensor(s) on networks where hosts store or process sensitive data. Continue toward less and less sensitive hosts until your budget runs out.

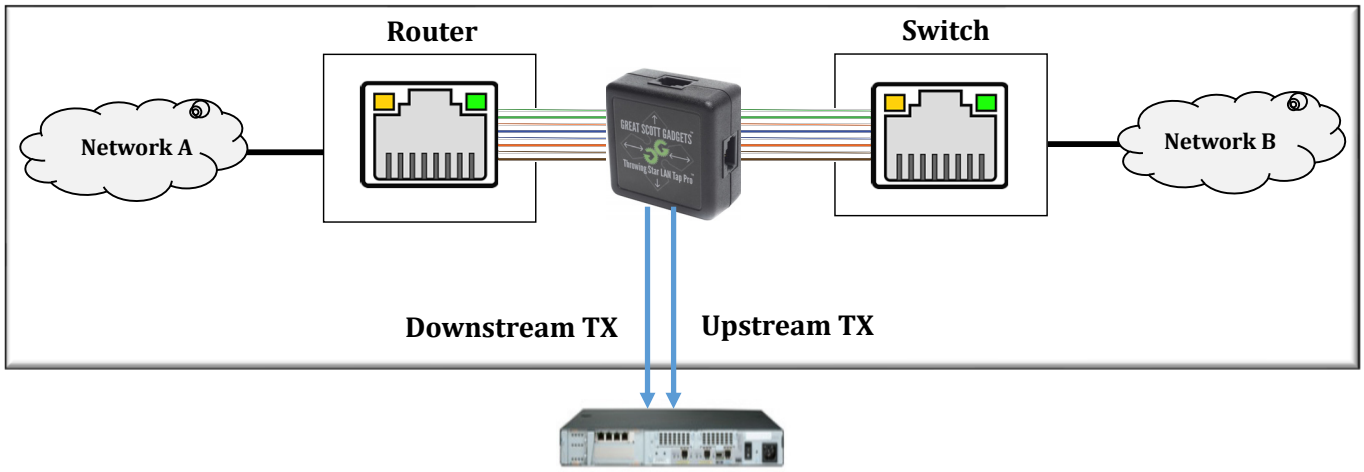
Need to Collect Traffic at a Given Choke Point (1)



When we wish to install an Intrusion Detection System to perform passive monitoring, we are seeking to *tap* into the wire or fiber optic cable that runs between two network devices. Typically, these devices will be things like border routers and firewalls, routers and switches, switches and other downstream switches, etc. The idea is that we want to monitor this link because all the traffic flowing between Network A and Network B passes across this link, whether that represents intranet traffic between two subnets, inbound/outbound internet traffic, traffic destined to or coming from a VPN, or any other network source.

What sort of devices and what configuration is necessary to tap this link?

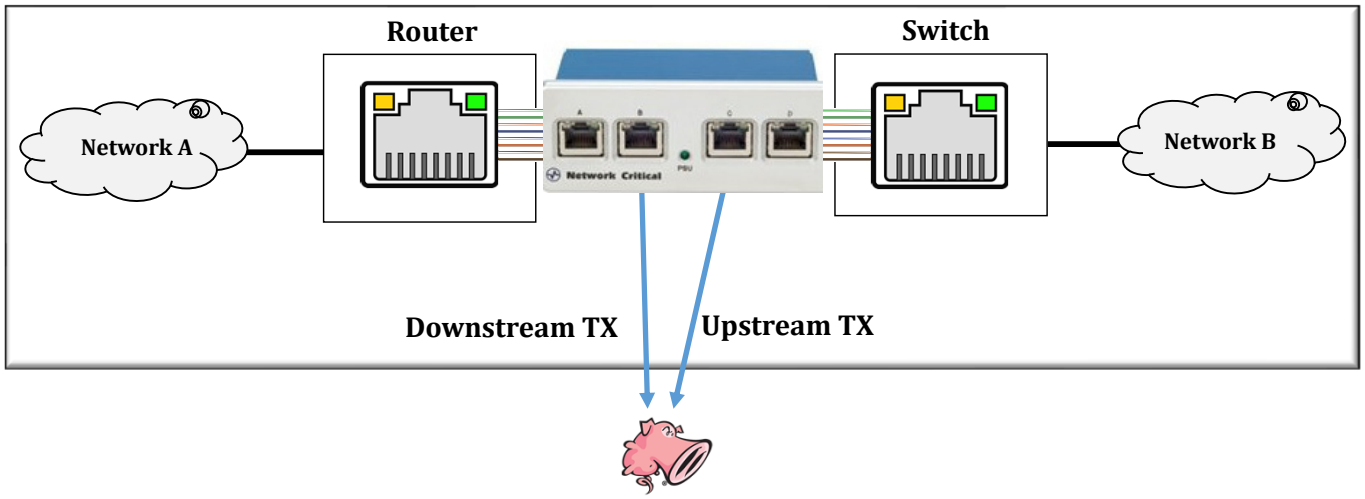
Need to Collect Traffic at a Given Choke Point (2)



A very low-cost option is an unpowered tap. The one pictured here is simply an arrangement of diodes and capacitors that connect the transmit pairs (TX) from each side of the connection with a pair of RJ-45 connectors on the tap. This type of tap has four ports on it: two ports for traffic that is passing between networks A and B, and two ports that are tied to each TX pair.

This arrangement is very inexpensive, but it's far from ideal. What if the tap fails? How do we reintegrate the data at the IDS? In addition to these issues, unpowered taps of this sort can significantly shorten the life span of the ports on the upstream and downstream devices, requiring more power for the signal that is being shunted to the IDS.

Need to Collect Traffic at a Given Choke Point (3)

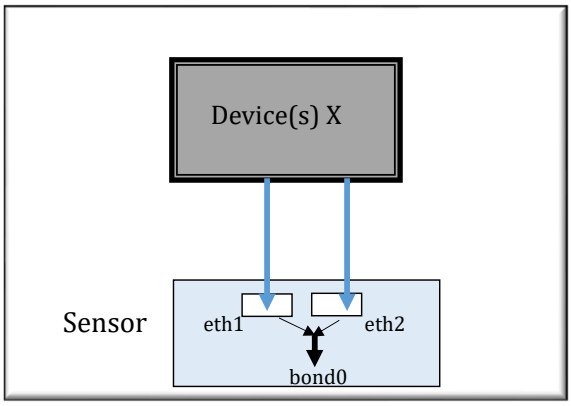


A bit more expensive, but well worth the money, is a powered tap. Powered taps eliminate the problem of drawing too much current through the upstream and downstream devices, while also regenerating the signal to the monitor ports. If using a powered tap, it is very important to ensure that it has a redundant power supply and that it will fail open should power be lost completely.

Even though this is an improvement, we still have the problem of having two separate wires coming to two separate interfaces. How can this data be reintegrated? Snort, for instance, will view each adapter as a distinct stream of packets. This will render all session-based alerting disabled!

Channel Bonding: Stealing Resources from the Sensor

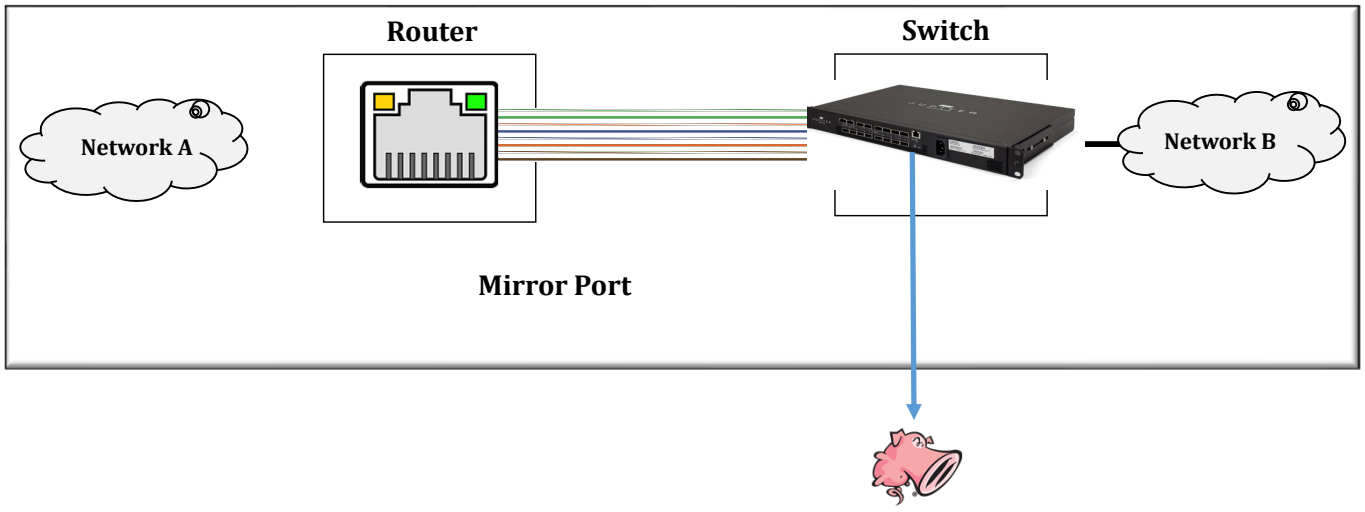
- "Bond" two interfaces into one logical interface for aggregation.
  - Requires the "bonding" kernel module
  - Uses the `ifenslave` command to bond the interfaces together
  - `ifenslave bond0 eth0 eth1`
    - This creates the interface bond0
    - This merges the packets seen on eth0 and eth1
  - Execute 'ip link set bond0 up'



An inexpensive solution for signal aggregation is to configure the kernel to perform channel bonding. Interface or channel bonding enables binding multiple interfaces together into one logical interface that can be used by a sensor. The output signals from the device(s) could be directed to separate network interface cards of a sensor (in this case, eth1 and eth2). These two interfaces would then be bound together to form an interface bond0. This interface allows a sensor to sniff bond0 traffic as if it originated from a single interface.

Suppose there is a device or multiple devices that produce signals that need to be aggregated. These do not have to be half-duplex only. Because high-end taps can be expensive, we may choose to opt for less expensive taps and rely on the sensor to reintegrate the data with channel bonding, but this increases the load on the sensor. Whenever possible, we prefer for the streams to be integrated before they arrive at the sensor, leaving the maximum processing available for monitoring.

Need to Collect Traffic at a Given Choke Point (4)



We might be able to save some money on taps if we leverage existing switches. For example, routers feed data into a network through switches. What if we were to configure a switch port to mirror all the frames that pass in or out of the port connected upstream to the router? While we wouldn't have the frames that pass between hosts connected to the switch, we would have every frame that passes in or out of the router.

This arrangement can work very well, but be aware that there are some downsides. First, switches are managed by the network operations team. It has been our experience that, from time to time, unrelated changes may inadvertently disable your mirror port entirely or reconfigure the port so that you see only half of the data. If the port is completely disabled, you are likely to notice relatively quickly since there are no packets showing up. If only one direction has been inadvertently turned off, you can go weeks or months before you notice. The reason is that packets are still arriving, so it seems as though things are OK. Unfortunately, since our monitoring tools generally will not generate alerts unless they view a connection as established, *our sensor has been completely blinded!* Why? Because, if you are seeing only one side of the connection, you will *never* view a connection as established!

One other thing to keep in mind is that the mirror port must have adequate bandwidth to support the data flow. For example, if the port that the router is connected to is a 10-gigabit port, you need a 40-gigabit mirror port. Why? Well, simply, there isn't a 20-gigabit port option available. Why can't you use a 10-gigabit port? Because the 10-gigabit port is full duplex! This means that there are potentially 10 gigabits up and 10 gigabits down, simultaneously! You need to get all that data moving out of the mirror port. A 10-gigabit port just isn't big enough!

What Differentiates Taps? (1)

- For monitoring purposes, taps are differentiated by (1):
  - The number of link choke points that they can accept for input
    - A basic tap has four ports:
      - Two network ports to copy/tap the half-duplex traffic from a single link choke point
      - Two monitor ports to send the half-duplex traffic downstream to a monitoring device(s)
    - More complex taps have additional tap ports to copy half-duplex traffic from multiple link choke points.
    - Some allow tap port input to come from one or more switch span ports.

There are several different tap types with ever-improving functionality being added with new technologies. Rather than get wrapped up in nomenclature of taps, our interest is the capabilities that they offer for monitoring. Simply stated, taps are differentiated by the ways in which input is accepted into the tap ports and what processing occurs in the tap before the traffic is sent to the monitoring port(s). This slide discusses the former; the next slide discusses the latter.

The most basic type of tap gathers input from a single link choke point using two tap ports: one to copy the half-duplex transmit signal and the second to copy the half-duplex receive signal. These are sent as half-duplex traffic to two monitor ports. It is the responsibility of the monitoring device or intermediary device to aggregate the signals back to full-duplex. More complex taps have more than four ports to accommodate input from multiple link choke points, and some allow the input to a tap port to come from span ports on switches that have already done some aggregation.

What Differentiates Taps? (2)

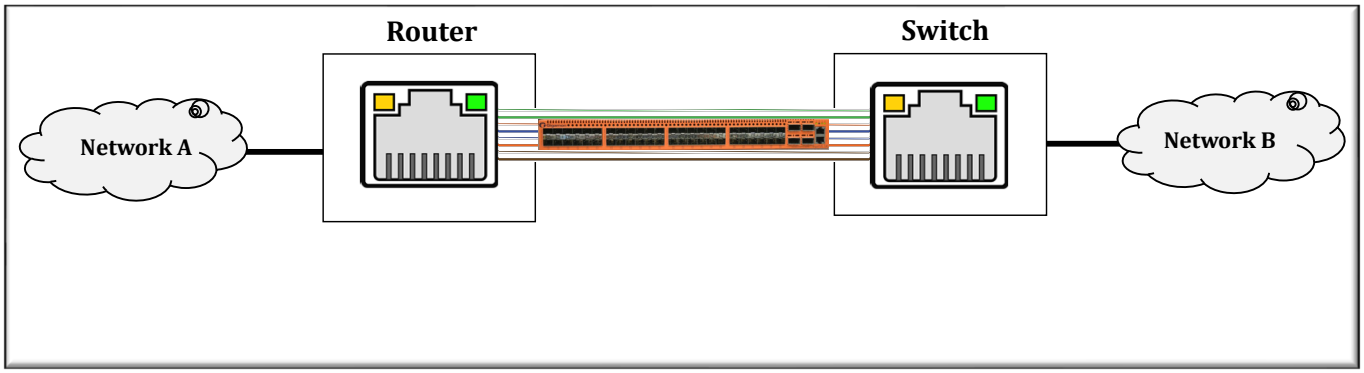
- For monitoring purposes, taps are differentiated by (2):
  - Capabilities of what they can do with the signals before sending the output to the monitoring port(s)
    - Aggregate the half-duplex signal to full-duplex
    - Configurable to indicate which tap ports are aggregated to which monitoring ports
    - Attach filters to monitoring ports to push out specific traffic—port 80 or 443, for instance—to particular monitoring ports

The second distinction is the processing that occurs in the tap that prepares the passing traffic for output to the monitoring port(s). Taps can aggregate the half-duplex signals to full-duplex before sending the traffic to monitoring port(s). This means that it is now unnecessary to have a device attached to the monitor ports that provide the aggregation.

Others can be configured to direct particular tap port input traffic to specific monitoring port(s), with the capability of aggregating the traffic for full-duplex as well. Some high-end taps can be configured to filter particular traffic, for instance port 80 or 443, to particular monitoring ports. In essence, the more preprocessing that the tap does, the less processing is required after the traffic is sent to the monitoring port(s).

Aggregating Tap

- Guarantees all packets get to where you want them
- Great for aggregating high flow links in one place
- Each port/port pair can act as both a tap and as a mirror.



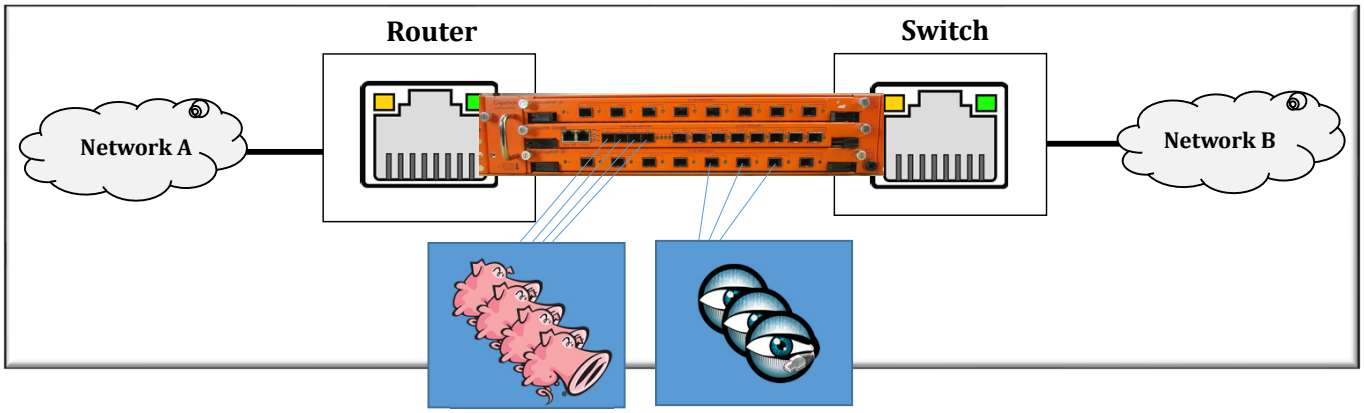
An aggregating tap is a possible alternative when you have a sufficient number of networks that you would like to aggregate down to a relatively small number of sensors, yet a load balancer would be overkill or too pricey. The advantage of the aggregating tap is that, much like with the load balancer, you are guaranteed that you will not drop any packets. While it doesn't permit you to split data out based on protocol, as does a load balancer, it does still permit you to perform "what" and "where" sorts of configurations.

An example would be that you might configure ports 1 and 2, which are typically paired, as a set of tap ports. These can now be connected just as any normal tap would be. Port 3, perhaps, is configured as a monitor port and connected to a tap that is installed between a core router and a downstream switch. Port 4 is connected to a switch that is configured to mirror the port that is connected to the upstream router. The same mixed and matched tap and mirror configuration continues through all the ports. Finally, we configure the sensor ports to place all the data from ports 1 and 2 on the first sensor port. Ports 1–10 are configured to come out of the second sensor port. Ports 3, 4, and 5 are configured to come out of sensor port 3, and so on.

This type of device allows for very flexible configuration of "what" (or which) data you want to show up "where." This can be an ideal solution when your volume of data is not so great that you need to load balance it, but you are willing to invest money in a higher-end piece of hardware that can fill almost all your tapping needs.

IDS Load Balancer

- Provides aggregation, round-robin traffic distribution, session-/stream-based distribution, and/or service segregation
- See traffic as **streams of data**, not packet-by-packet

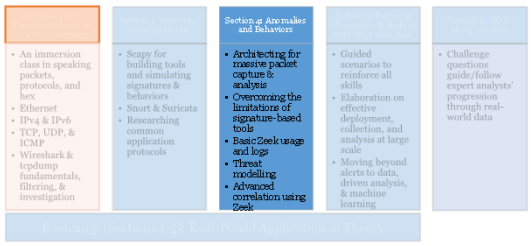


An IDS load balancer or packet broker provides a solution where multiple sensors can share the load of monitoring high-throughput traffic. As we saw with the switch and switch/tap combination, the spanning port can send the monitored traffic to one sensor only. An aggregating tap improves this, but generally doesn't provide a way to load balance the data.

The IDS load balancer or packet broker solves this problem by allowing multiple sensors or monitoring devices to receive the output. This allows many different configurations. The simplest configuration is where many single sensors handle an aggregate capacity of the throughput of the network. Instead of relying on one sensor to capture all traffic, the load is distributed. The traffic load balancer may provide a round-robin distribution of traffic based on flow monitoring, its definition of conversations, or sessions. Alternatively, some use a simple hashing algorithm to ensure that all the packets for an IP port pair are sent to the same monitoring device(s). In other words, instead of taking single packets and evenly distributing them among the sensors, it takes an entire conversation (such as a TCP session) and directs it to one sensor. Otherwise, you'd be left with unaffiliated packets and unintelligible garbage on the sensors.

Another configuration option enables you to direct certain services to different sensors. Essentially, you may have a sensor where you send HTTP traffic only, one where SMTP traffic is sent, and a third to receive all other traffic. This also enables you to use different types of sensor software to provide the best solution for your needs.

You may hear the term *traffic balancer*. The concept is similar to an IDS load balancer, where a device takes traffic and distributes it to multiple devices. The difference between an IDS load balancer and traffic balancer is that the IDS load balancer understands the notion of associated traffic, such as streams or ports, whereas a traffic balancer deals on a packet level only. An IDS is not useful if it cannot see all traffic for a given session or stream.



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

IDS vs IPS

Tapping Networks

Collecting Packets at Scale

Show Me the Packets!

If nothing else, this class should convince you that you need packets!

- There are many approaches.

Free solutions:

- Daemonlogger + scripts
  - This is the architecture long used by Sguil and others.
- Stenographer
  - Go-based solution from Google (unofficially)
  - Wire-speed logging with low-cost commodity hardware

Whatever you choose, we suggest keeping the packets on the sensors.

- We strive for a minimum of 7 days of full packet capture on the busiest link.

We feel confident that, by this point in the class, you have come to believe that packets are incredibly valuable! If we have the packets, there are so many questions that can be answered. At times, without the packets, there will be lingering questions that can *never* be answered. How, then, do we collect them at scale?

The answer to this question depends on the scale of your organization, the budget for the project, the number of networks to be monitored, and the total volume of packets you need to retain. Some will choose to use an inexpensive approach to the collection problem in order to spend more money on other elements of the overall monitoring infrastructure (a TLS decryption system or large packet broker/load balancer, for instance). On the free side (costing nothing but time, effort, and ongoing maintenance) are **Daemonlogger** and **Stenographer**.

Daemonlogger was written by Marty Roesch and released as a free tool for packet collection, allowing you to offload the collection to a streamlined tool with no interpretation capabilities. Daemonlogger can automatically rotate files but is typically used in concert with automation scripts to manage the captures, rotate files, and extract data. The **analyze** script in the virtual machine is just such a tool, intended to be used with packet captures collected with Daemonlogger.

Stenographer, on the other hand, is a free tool that is (unofficially) from Google. It is a Go-based solution that is intended to maximize the amount of data that can be streamed to disk, allowing you to maximize commodity hardware. It includes a set of tools and a simple language that can be used to query packets out of the repository.

Whether you choose a free or a commercial solution, you need to determine how much data you need to retain. With our customers, we aim for a minimum of 7 days of full capture on the busiest link. With the cost of storage today, this is an easy target. In fact, with a gigabit or 10-gigabit link, we will aim to retain far more than 7 days as a minimum.

Commercial Packet Capture

Commercial offerings can be quite expensive:

- We tend to work with these only on our highest-end customers.
- Can be very costly!
  - Most of our customers do not view this data as mission-critical, so a FOSS solution is okay.

SentryWire

- Top solution supports full packet capture at 100 Gbps with 6+ petabytes storage
- Can be clustered/federated to aggregate larger pipes

Endace

- Long-time player in the high-speed packet capture space
- Clustered/stacked solution for effectively any size pipe or retention requirements

We may prefer or require a commercial solution. A simple reason to prefer a commercial solution is that your organization will not be dependent on someone with very specific knowledge about a system that has been built from scratch and has no commercial support (that is, an FOSS solution such as those on the previous slide).

There are a number of commercial solutions available. The author has specific experience with **SentryWire** and **Endace**.

Endace is a very well-known name in the network monitoring community of practice. It has long made some of the best (and fastest) packet capture devices, though it can be prohibitively expensive for a single network card. Endace offers a stackable solution that can effectively be scaled to any size pipe or storage requirements.

Similarly, SentryWire also offers a highly scalable solution. Its top-tier system supports a base speed of 100 Gbps with over 6 petabytes of storage. This solution can easily be federated to aggregate the data from much larger pipes, at least as large as 1 Tbps.

High-End Tap Required

In this territory, a high-end tap becomes a requirement.

Vendors we have experience with:

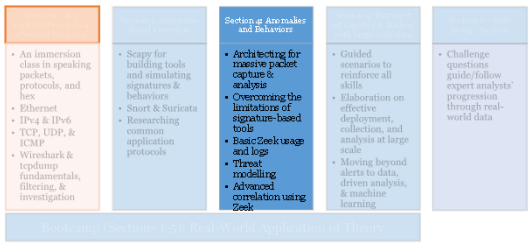
- Endace
- Gigamon
- NETSCOUT
- Garland Technology

Very important to understand licensing of features and ports

At these prices, you can absolutely get them to send you a demo model.

When you are dealing with traffic flows in excess of 10 Gbps, you should very seriously consider investing in a high-end tap or packet broker. We always recommend that an organization buy a device that is "larger" than the current requirements. For example, if the fastest link on the network is currently 40 Gbps, we would recommend it purchase a solution that can support 100 Gbps.

Our reasoning is that our network is unlikely to need *less* bandwidth in the next 5 years. If we invest well now, we can purchase a solution that can scale with us for 5 to 10 years. This has the benefit of stretching the total cost of ownership out over a much longer time span. With server hardware, we would never recommend this; with a network tap or packet broker, this is completely feasible. These devices have no trouble running for a decade or more with little to no downtime.



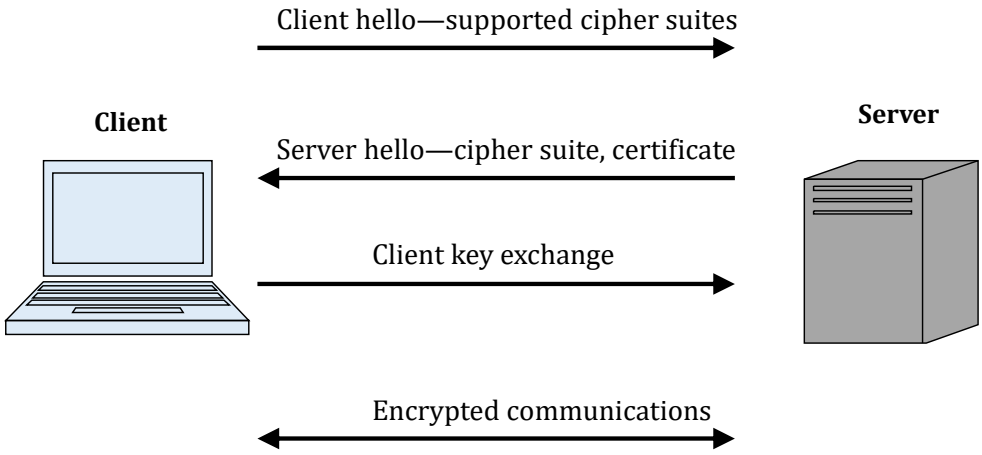
- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

Versions and Operation

Decryption Approaches

Transport Layer Security (TLS) Protocol



The *Transport Layer Security (TLS)* protocol is used to establish encrypted sessions to a server. You may hear the term *Secure Sockets Layer (SSL)*; this was a predecessor to TLS. While the terms tend to be used interchangeably, you should absolutely not be using SSL today. Normally, TLS is an innocuous protocol used to encrypt sessions where the legitimate data must remain private. Yet, it can be used in command-and-control traffic, exfiltration, and other malicious activity. Let's first discuss TLS 1.0 through 1.2.

This is not a thorough discussion of the precise details of the protocol—just enough for you, as an analyst, to understand in terms of network communications. There is an option to use client certificates; however, we examine the use of the server certificate only. We discuss the handshake portion of the TLS exchange that establishes the encrypted session that follows.

When a client wishes to establish a TLS connection, the client sends a **hello** that includes the version of TLS supported, the cryptographic ciphers that it supports, and a random 32-bit string (known as a "nonce"). This 32-bit string is used later in the handshake with a random 32-bit string from the server to generate a *premaster secret.* This is eventually used to derive the session encryption keys.

The server replies with a **hello** containing its supported version of TLS, the selected cipher for the connection, and its certificate. The server also returns a random 32-bit nonce to be used by the client for session key exchange.

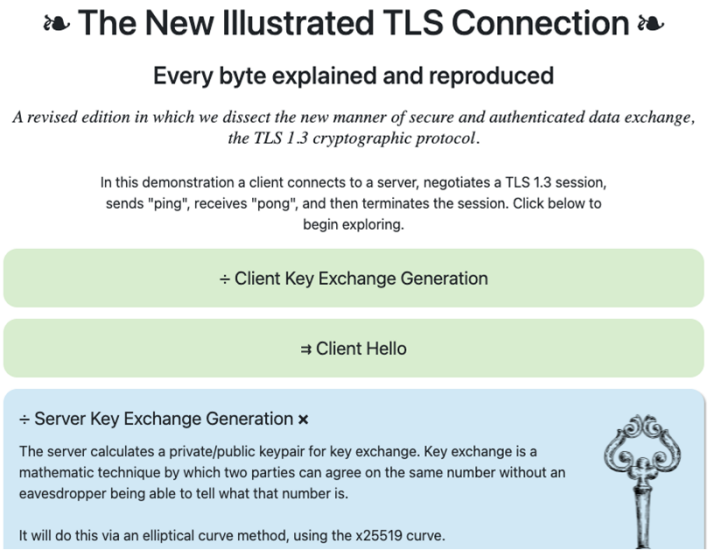
The client validates the certificate by comparing the signed fingerprint found in the certificate, generated by a *Certificate Authority (CA)* using the CA's private key, with the client's own computation of the signed message of the certificate data using the CA's public key.

The client uses the server's nonce along with its own to generate a premaster secret, which is encrypted with the server's public key (found on the server's certificate), sending it to the server. The server decrypts this using its private key. Both the client and server use the pre-master secret to generate the *master secret*, which is then used to derive the session keys to encrypt the session's traffic.

Understanding TLS

https://tls.ulfheim.net – TLS 1.2
https://tls13.ulfheim.net – TLS 1.3

An incredibly useful breakdown of TLS phases!



There really is an incredible amount of detail that you can find in a TLS establishment. This includes things like the SNI (Server Name Indication), which can provide hints about what the name of the site is that's being connected to, even though the actual URL is encrypted, and more. Aside from the TLS RFCs, which can be challenging to read and understand, one of the very best resources for understanding and analyzing TLS operations and packets is the website **https://tls.ulfheim.net**, which details TLS 1.2 connections, or **https://tls13.ulfheim.net,** which does the same breakdown for TLS 1.3.

The author of this site has taken the time to fully break down every phase of every aspect of a TLS connection. The page is fully interactive, allowing you to click through to detailed explanations, all the way down to the bit and byte level. We do not plan to do a byte-by-byte analysis of TLS, but if you ever need to do one, this site is your friend!

TLS 1.3

Formally adopted in August 2018

- Initially created monitoring problems
  - Disabled many older algorithms upon which decryption relied
- Leverages TCP Fast Open
  - Subsequent connections rely on previously negotiated key material
- If coupled with DNS over TLS, it can be game over for detection
  - We would only know that two addresses spoke; we would know almost nothing else.

Two main strategies:

- Log/obtain the pre-master secrets from the clients.
- Use a terminating proxy.

TLS 1.3 was adopted as a standard in August 2018. It promises to close some gaps that TLS 1.2 left open that affect privacy. This includes things like the SNI (Server Name Indicator). It also approaches the session startup in an entirely new way that effectively means that most of the session is encrypted from the very first packet (especially after the first connection, since it leverages TCP Fast Open). This can represent some challenges for older decryption solutions.

In TLS versions older than 1.3, the client and server perform a complete handshake, including key establishment, in every connection. In TLS 1.3 with TCP Fast Open, the session key is negotiated in the first connection. That same key is then reused (though it can be changed periodically) in all future connections during that time period. This allows the SYN of every connection after the first to both carry data *and* be fully encrypted.

Previous decryption solutions took advantage of older encryption algorithms that would allow you to install the private key (which we could view as a master key) for your defended systems, allowing all the data in and out to be decrypted easily. TLS 1.3 eliminated all algorithms of this sort. As you can imagine, this created a bit of a stir because we suddenly could not see what our endpoints were doing. If this were coupled with DNS over TLS, we would have no good way of even guessing what was happening in a connection.

There are two main strategies for decrypting this data. One is to use an agent on the endpoints to obtain the pre-master secret (the endpoint's "secret" that is used to generate all the keys in a connection) and the nonce (a random value also used to generate all the keys) and relay them to the monitoring system or the decryption tool.

The second, and more common, approach is to use a terminating proxy. We'll discuss this solution more soon.

STARTTLS

- Called "opportunistic" versus "mandatory" encryption
- Protocol extension to transform unsecure communication channel to a secure one using TLS
- Does not require a separate listening port for communications; one less port to allow through firewall
- Server announces support for STARTTLS; if requester supports/accepts, conventional TLS session negotiation occurs over the existing port
- Gained traction by popular email providers after revelation of NSA surveillance
  - Encrypts messages and metadata such as subject, sender, and receiver, so cannot be seen
- Mostly used between SMTP message transfer agents

A protocol extension known as *STARTTLS* can be used with services such as SMTP, IMAP, LDAP, FTP, and POP3 to transform a plaintext communication exchange into an encrypted one. STARTTLS is known as *opportunistic encryption* because it is not mandatory and it opportunistically rides across an existing protocol port that does not use encryption.

This is different from SSL because encryption is mandatory. There is no choice about it: either the communication is encrypted or there is no communication at all. One of the advantages of using STARTTLS is that a single port is required, such as port 25 for SMTP rather than port 25 for cleartext communications and TCP port 465 for encrypted communications. This has the added benefit of simplifying firewall access control rules for permitted ports.

A server that supports STARTTLS will indicate this via a **STARTTLS** message (for instance, for SMTP by using a 250 code after the client **EHLO** message). Support on the client side is designated by a **STARTTLS** message in return, followed by TLS session negotiation for encryption. Lack of support by the client causes the communication to occur in cleartext.

The *Electronic Frontier Foundation (EFF)*, self-described as "defending your rights in the digital world," recommended, in 2013, the use of STARTTLS by all SMTP *message transfer agents (MTAs)*, also known as SMTP servers. This resulted from the revelation that the NSA was conducting surveillance of SMTP traffic. STARTTLS encrypts email subjects, senders, receivers, etc., most likely of interest to those doing surveillance.

STARTTLS is mostly used between MTA servers. This means that anyone with access to either the sending or receiving server, including hackers, can see the cleartext in transit to or from the MTA before encryption or after decryption. Also, any email that has been archived from the MTA is in cleartext. MTAs associated with ISPs must be aware that this may be of interest to law enforcement or the government. STARTTLS is not intended to be an end-to-end encryption method, just MTA to MTA.

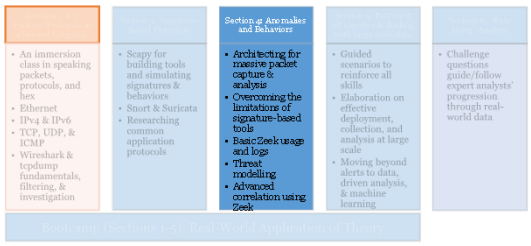
.

STARTTLS Downgrade Attack

- Fails open to unencrypted
- STARTTLS command not encrypted, enables tampering
- Machine-in-the-middle attacks to disable STARTTLS
- Networking equipment, for instance, used by Cisco performs downgrade for security to inspect email, look for spam
- ISPs deliberately turning this off for SMTP servers

There is a potential vulnerability with using STARTTLS because a failure defaults to an unencrypted state if support is missing. There is no notification that there is a downgrade to cleartext. As you saw, the intention to use STARTTLS is declared with a simple STARTTLS message by both sides. This is in cleartext, enabling possible tampering such as a machine-in-the-middle attack, where traffic from either the source or destination MTA can be modified to exclude the STARTTLS message.

Also, some networking equipment, such as that used by Cisco, purposely downgrades the exchange to cleartext to inspect email (for instance, looking for spam). And for whatever reason—perhaps because of the overhead required to establish and transport encrypted messages—ISPs have been turning off STARTTLS.



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

Versions and Operation

Decryption Approaches

Encrypted Data

- The internet today is encrypted.
  - In the developed world, you would be hard-pressed to find any significant internet presence that does not require TLS.
  - Browsers default to TLS rather than HTTP.
- Decryption is possible:
  - Introduces risks
  - Simplest approach is a terminating TLS proxy

Before we dig into command-and-control traffic, we want to take a few moments to finish a conversation that we started back in Book 3. When we were discussing TLS, we said that there was more to talk about, but we wanted to complete our analysis sections first. The topic that we did not cover is, "What do I do with all of this encrypted traffic?"

This is becoming a more pressing problem because more and more services and sites are moving to encrypted-only solutions. While this is great for privacy and security in general, it's not so great for us. EFF (Electronic Frontier Foundation) reported that, in 2017, more than half of web traffic was encrypted on the internet at large for the first time. In 2018, *Computer Business Review* reported that not just web traffic but 50% of *all* internet traffic was encrypted (https://www.cbronline.com/news/internet-encryption-sandvine). No doubt the percentage has continued to increase. We've seen browsers shift to preferring HTTPS over HTTP and using HTTPS as the default to connect to URLs.

Is decryption possible? Yes. Before we get into how, realize that decrypting client traffic to the internet creates risks for us, not the least of which could be privacy issues. Could you end up collecting or looking at data that you would really prefer not to know about?

If your organization decides that decryption is a must, our experience is that intercepting or terminating proxies are the best way to go.

Terminating Proxy

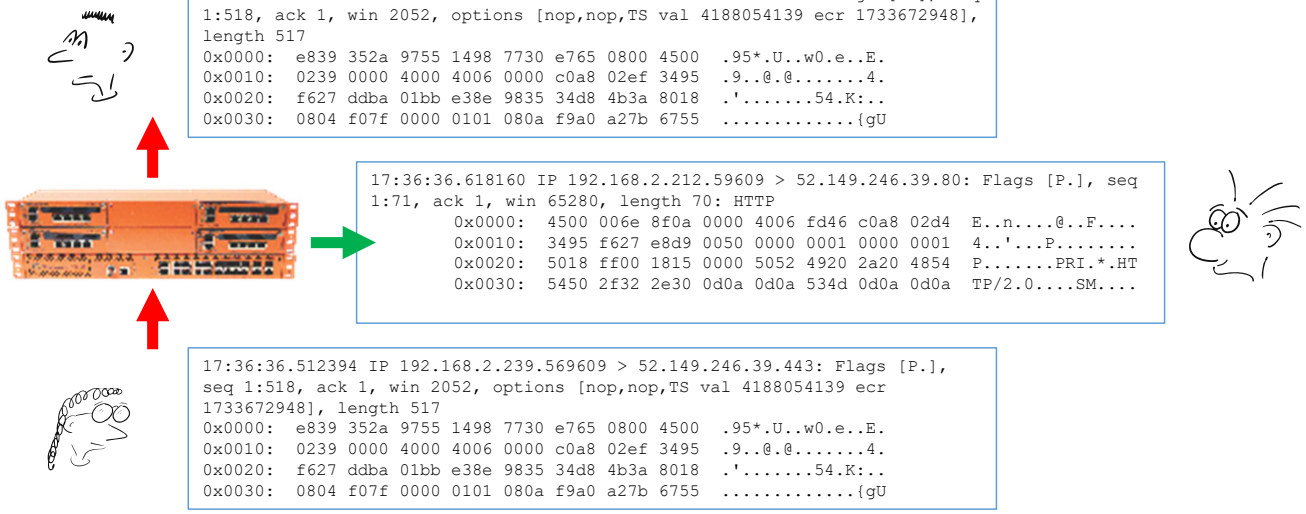
- TLS encrypts only in transit:
  - Client connects to TLS proxy over TCP
  - Proxy connects to server over TCP
  - Client begins TLS handshake
    - SNI and other elements used to hint at the DNS name being requested
    - Generate a certificate on the fly
  - Complete exchange between client and server
  - Data is in cleartext on the proxy, or the keys for the client-side connection can be saved.
- Risk!
  - Potential privacy and related security issues
  - Clients can no longer tell when a suspect certificate is presented by the server.

A terminating, or intercepting, proxy operates by acting as an endpoint in the TLS conversation. Normally, proxies will simply pass TLS data through, allowing the two endpoints to speak directly. With a terminating proxy, the client's outbound connection is *terminated*, or connected to the proxy itself. The proxy then establishes a secondary connection to the target site. You might be reading this and thinking, "That's what a proxy always does… how is this different?" The difference is that, rather than passing the TLS through unmolested, the proxy now *acts* as an endpoint. This means that the client handshake is performed between the client and the proxy, not the client and the target site!

Still, for this to work, the proxy must present an acceptable certificate. This is somewhat easier than it sounds. First, if the client is a system under our administrative control, it is trivial to install a CA certificate from our domain. This causes any certificate signed by a signing subordinate of that CA to be trusted. Second, while much of the TLS conversation is indecipherable, there are things like the SNI (Server Name Information) sent in the clear that can be used to hint to the proxy what the domain name is with which the connection is being established. This allows the proxy both to send the correct request to the server and to generate the correct false certificate to present to the client.

Having done this, we can now inspect the data at the proxy or export the data for offline analysis. Alternatively, your proxy might support a configuration that allows all of the keys for all of the connections to be exported, allowing us to decrypt any pcap for which we have the proper session keys.

Intercepting Proxy in Action



In case you're having trouble visualizing what an intercepting proxy might look like in action, consider the illustration in the slide. Here we can see a host on the bottom, Alice, establishing a connection to an external host, Alvin, using TLS over port 443. In between them, and transparent to them both, an inline TLS decryption device that acts as a terminating proxy is installed.

When Alice connects out to Alvin, her connection is actually to the proxy. The proxy, in turn, connects to Alvin. Since the proxy is a party to all of the communication, the unencrypted content is available to the proxy. In the example in the slide, the proxy has been configured to relay the unencrypted packets out of a "tool" port, as they are termed in Gigamon's documentation. In this case, the proxy has been configured to output the decrypted TLS data over port 80, but this is optional. The data could as easily be sent out over the tool port to port 443 to match the original packets.

Proxies like this can be used to intercept and decrypt more than web traffic. They also typically support standard services running over TLS, such as IMAP, SMTP, POP3, and others. The data would be presented unencrypted out of the tool port in the same way, just to whichever port you choose to configure it for.

Decryption Solutions

Gigamon GigaSMART

- Up to 100 Gbps
- Decrypts on the fly, generating sort-of duplicate packets with decrypted payloads that you stream to your monitoring tools

F5 Networks SSL Visibility and Orchestration

- Focuses on preselecting data that should not be decrypted vs data that should be
  - For example, outbound connections to banking sites, employee benefit sites, etc.
- Like Gigamon, allows you to stream decrypted payloads past any number of tools

We're not recommending either, and there are more options out there!

- Try PolarProxy if you want something quick for testing.

There is very little available in the world of free software that can handle TLS data at any scale. The problem is that the encryption and decryption are computationally expensive. There's no problem handling a handful of connections, but 50 or more hosts attempting to establish TLS connections through your border will rapidly swamp any of the free possibilities.

The reason that the commercial solutions work well is that much of the encryption/decryption work has been offloaded into custom ASICs (Application-Specific Integrated Circuits). Implementing complex encryption algorithms in hardware is time consuming, but once you have done, so they are several orders of magnitude faster than software solutions.

We have had experience with both the F5 and the Gigamon solutions for decryption. They both work quite well, decrypting the data while acting as a machine-in-the-middle. This decrypted data is then placed into packets that have the same IP addresses and port numbers as the original packets, except that they are fully decrypted. These are then streamed past however many tools you need to inspect the decrypted traffic with.

The F5 solution is interesting if you are very concerned about selectively ignoring certain types of traffic. For example, if you are concerned about liabilities resulting from decrypting employee communications with banking institutions, insurance companies, employee benefit programs, and other types of traffic, it has a pre-filtering engine built into it that allows you to select types of traffic that should never be decrypted.

If you want to give interception a try without investing any money up front, have a look at PolarProxy (https://www.netresec.com/?page=PolarProxy). You can use their free license for up to 10 gigabytes of traffic per day.

Can We Profile Traffic?

- Do you need to decrypt traffic to hypothesize what's going on?
  - How does web browsing behave?
  - How does streaming behave?
  - How does watching YouTube behave?
  - How does C2 behave?
  - How does exfiltration behave?
- Cisco implemented the idea we've been teaching for 20 years!
  - Cisco Enterprise Traffic Analytics
    - Cisco Joy: https://github.com/cisco/joy

We would even argue that, while having the decrypted packets (or the ability to decrypt the packets) would be amazing, we're not sure that it's all that critical. Given the knowledge that you've developed while taking this class, we feel that you're ready to take it to the next level. How so? Think about this: Is it possible to make a *really good guess* as to what is happening inside of a connection *without ever reading the content of the packets?* This really is starting to sound like the Zen of packet analysis!

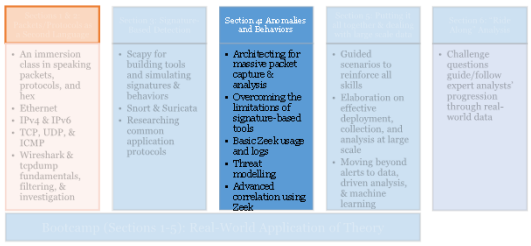
How could this be possible? Well, ask yourself: What would you expect the behavior of packets to look like in a session that is being used for web browsing? What if, instead of simply browsing the web, the user is streaming a movie? What if, instead of watching a movie, the user is on a video-sharing site (like YouTube or similar)? Can you imagine what the differences between these would look like?

This is an idea that we've been talking about and teaching people about for years. You can imagine our angst when we heard that Cisco released a product in early 2018 called "Enterprise Traffic Analytics." This tool automates what we are describing here. It looks at the behavior and attempts to fit that to possible activities. While it can't *read* the data in the packets, it can *guess* what's happening… which is something that any good analyst should be able to do!

TLS Decryption

Workbook exercises "Decrypting TLS with Wireshark"

This page intentionally left blank.



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

Introduction to Zeek

Zeek and Signatures

Zeek Scripting Basics

Threat Analysis

Scripting Correlations

Scripting Behavioral Anomalies

Spicy

What Is Zeek?

- Open-source software originally written by Vern Paxson
- Zeek cluster configuration supports high-bandwidth networks
- Network traffic analysis framework
- Generates log files of network activity
- Event driven
- Default functionality to assist with analysis and detection
- Customizable for site-specific analysis using scripting language
  - Enables analysis among streams/sessions
- Supports rudimentary signature creation

Zeek is open-source software originally written by Vern Paxson at Lawrence Berkeley Labs. Today, Zeek is supported by several different developers at the International Computer Science Institute in Berkeley and the National Center for Supercomputing Applications in Urbana-Champaign, Illinois. Zeek is deployed in just about every industry but seems to have the most ardent following at academic sites.

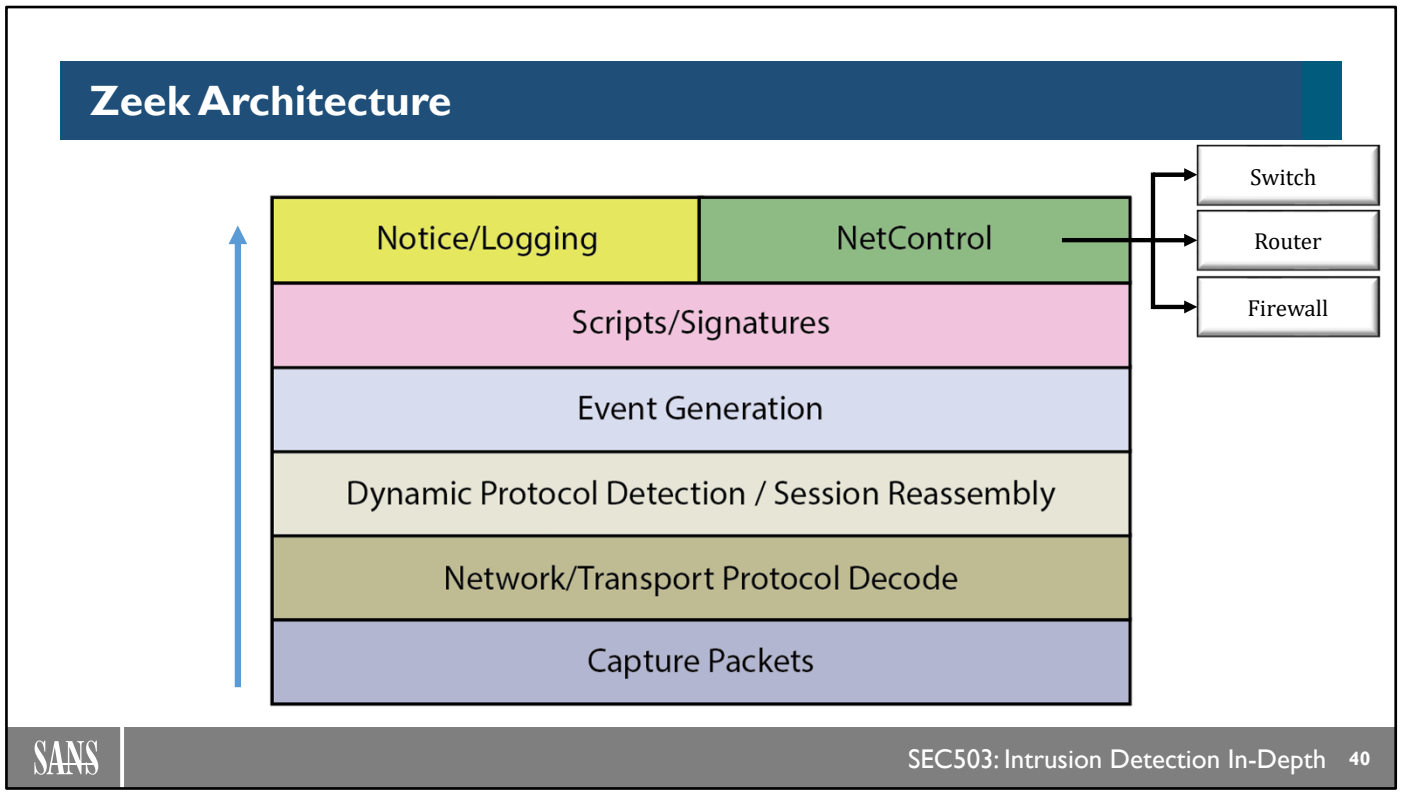
It has been associated with academia since its inception, and as such, it supports the concept of free and open-source software (FOSS) on affordable commodity hardware. Zeek can be configured to run on a single host for smaller networks or in a cluster for larger networks. The cluster facilitates expanding network coverage, when network traffic or speed increases, with the addition of more hosts that sniff traffic. This doesn't require an upgrade or a difficult port to more powerful hardware; it simply relies on adding Zeek *workers* (sensors) to the existing Zeek infrastructure.

Zeek is fundamentally different from a signature-based IDS/IPS. It provides traffic analysis on every connection and records notable details of each in various tab-separated logs for post-processing.

Other IDS/IPS solutions focus and report on detection of anomalous or malicious traffic. Zeek is capable of doing this, but this is not its exclusive purpose. It also logs what it defines as events and records summarized data about each connection. Events can be normal or anomalous network events—something as simple as a new network connection. These events are predefined in the software and can be customized by the user to do some post-event processing that includes creating some kind of notification or action of noteworthy activity. In Section 5, you'll learn about the concept of *alert-driven* versus *data-driven* sensors. Zeek falls in the category of data-driven because it collects the data, but it is up to the analyst/software to determine its significance.

Zeek has its own complex and full-featured scripting language that permits site customization for just about any detection and analysis. The scripting language is extremely powerful and has functionality equivalent to many other high-level languages like Python, for instance. However, the purpose of the Zeek scripting language is to facilitate parsing and analysis of network traffic. Other languages like C and Python support analysis of network traffic by importing libraries that are capable of examining elements of TCP/IP and related network activity.

Zeek does support rudimentary signature creation yet relies more on its scripting language for advanced analysis.



Zeek's architecture relies on *libpcap* at its base to sniff and filter packets, much like Snort. Libpcap is an API to capture and store network traffic. It is at the foundation of many of the tools and products for network analysis. *PF_RING* is software that can be used to improve packet capture performance. It can be wrapped around libpcap when there are multiple Zeek sniffing processes on a single host, permitting the traffic to be directed to individual cores on multicore architecture hosts. Think of it as host load balancing.

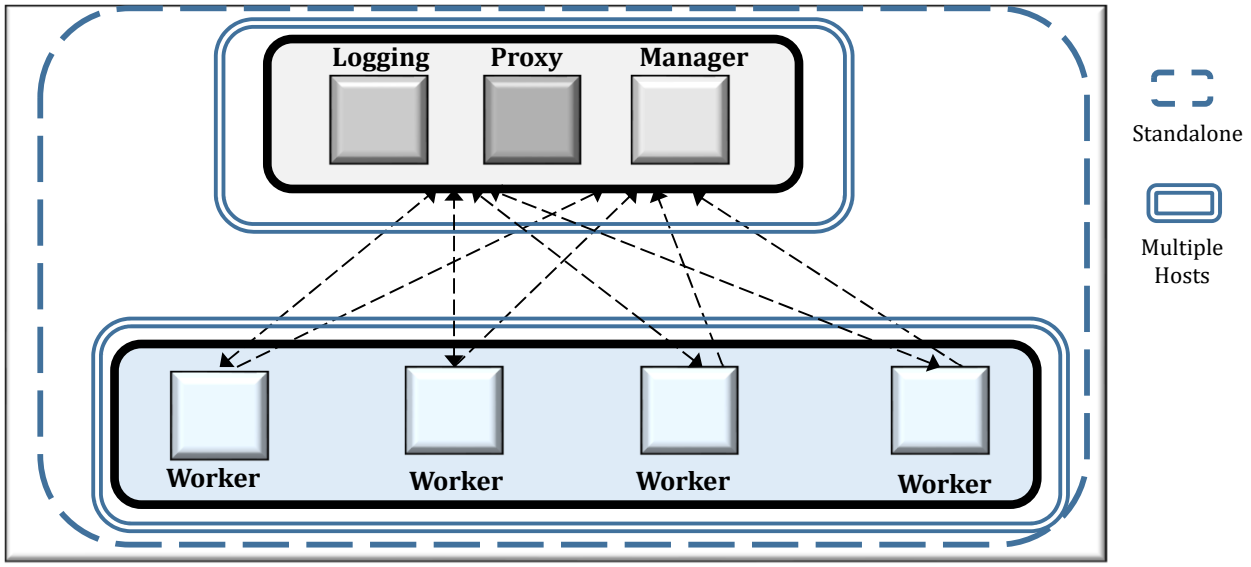
The packets are first decoded in terms of network and transport protocols, from which Zeek can build session information. These sessions are further analyzed by the dynamic protocol detection engine to decide which application protocols are present. This data is all made available to the event engine.

The event engine takes the traffic reassembled as streams and generates network events for several hundred noteworthy activities. There are normal activities such as a new connection being established, an observed HTTP request, and many others. They pertain to different network layers, application protocols, along with various stages of communication with those protocols.

The events that are generated are published to the rest of Zeek as events that can be reacted to from within scripts. In a sense, there is a parallel path wherein signature detection is performed as well. Based on processing within the scripts, you could choose to use the NetControl capability to reach out to and reconfigure a variety of network switches, routers, firewalls, and other devices. This creates the ability to perform some real-time reaction.

You can *raise a notice* to cause a message of your choice to be written to a file called **notice.log** (or any other Zeek log) on detected activity. Alternatively, you can raise a notice with an action of emailing a chosen recipient(s). The term *raise a notice* means creating some kind of notice for the event to signal that the analyst should examine the activity. It is similar to a Snort alert. There are many logs that are generated automatically by Zeek through the protocol analyzers.

Zeek Cluster Configuration



Zeek has four different types of nodes: the *worker*, the *manager*, the *proxy*, and the optional *logging* node. (Logging is typically performed at the manager, but for particularly busy sites, you might choose to have this function reside on a different system.) Together, the nodes form what is known as a Zeek *cluster*. Zeek even has its own communications protocol, **broker**, to transmit Zeek traffic between nodes.

The worker, as the name suggests, is the workhorse that sniffs the traffic, reassembles it into streams, and performs protocol analysis on the streams. The worker is commonly known as a sensor for most other IDS/IPS solutions. The deployment form that the worker takes can be varied. A given worker can be an individual separate host, or a process on the same host as other workers sharing one or more cores on that host. A worker can share the same host as the other node roles. This is a *standalone* implementation. The number and placement of the workers depends on the traffic volume and the type of hardware available for traffic collection. Because the worker sniffs and processes traffic, it needs to be deployed with abundant memory and CPU speed, which is more likely found on one or more multicore hosts.

The manager is the central log and notice collector, creating a single log for all the traffic collected by the workers. These log messages can be processed into actions such as email of an event. Communications between the manager and workers are initiated by the workers.

The proxy is responsible for making it appear as if all the workers are operating in a single unified environment instead of as different hosts or different processes on separate cores on the same host. The proxy synchronizes the state of Zeek by sharing information about active hosts and services on the network as well. There may be multiple proxies in a Zeek implementation if one is insufficient. The proxy and manager may be run on the same physical host if the number of implemented workers and associated activity are manageable.

A logical distinction is made between *standalone mode,* where all nodes are on the same host, and *cluster mode,* where at least one node is a remote host.

Clustering for Collection

- Clustered workers allow:
  - Better use of multiprocessor/multicore systems
    - Each process can be pinned to a specific processor.
- There are a number of approaches; here's one:
  - Packet broker retargets packets to specific workers by rewriting the MAC
    - This can be done by host
    - This can also be done by stream
    - The workers use BPF filter to limit which packets are passed into the processing stream
  - Prevents any one worker from being overwhelmed, leading to lost packets

The ability to cluster using Zeek cannot be overemphasized. Since the system was designed as a clustered solution from the outset, it has built-in features for cluster management and cluster communications and can easily handle very large bandwidth sites. Additionally, we can potentially put multiple workers onto a single device, which allows us to use the multiple cores and processors in a sensor far more efficiently. (Recall that Snort is single threaded: It doesn't matter how many cores you have, Snort will only ever use one.)

How would this work in practice? Especially when clustering to allow for analysis of high-bandwidth links, multiple workers will all be examining the data stream from that link. In this arrangement, we would typically install a packet broker or load balancer that will split the data out to the different workers, allowing each of them to focus on data for a subset of the hosts or streams that are present. This could be done by the packet broker or load balancer splitting the streams out over multiple ports, each of which is connected to a worker. Alternatively, some implementations will rewrite the destination MAC in the frames. When this approach is taken, all the workers see the same stream but use BPF filtering to limit which destination MAC they will capture and analyze traffic for. Either way, this can serve to prevent any one worker from becoming overwhelmed and dropping packets.

Clustering for Correlation

Clusters can also be much bigger:

- Manager still acts as the "brain" and potentially logging system
- Depending on the number of workers, proxies are deployed strategically.
  - Take care of disambiguating UIDs for connections among workers and other tasks.
- Workers deployed throughout the enterprise
  - Perhaps one proxy per physical site

We now have the ability to perform large-scale correlation across the enterprise.

- We may even have high-speed Zeek clusters within this arrangement.
- The documentation mentions this as a "cluster of clusters."

Clustering is not limited to high-bandwidth situations. We frequently cluster multiple groups of workers across an enterprise, which is when things can become extremely powerful in terms of correlation. The Zeek documentation has only just begun to discuss this type of deployment, which they refer to as “a cluster of clusters," though the documentation doesn't actually explain how to do it or what to do with it. We'll see some examples of correlation that can come from this later.

The idea here is that we still have a central Zeek manager that acts as the "brain" for the entire cluster. This is where all the correlation occurs. This manager has dozens or even hundreds of Zeek workers sending information back to it from all over the enterprise. Since we now have so many widely deployed workers, deploying more proxies makes more sense. In fact, to maximize performance, we would generally suggest that you deploy at least one proxy at each site (whether this is a physical site or a separate cloud infrastructure/region). This proxy is used to disambiguate some of the activities that the workers do, ensuring we have consistent UIDs and other values across the events generated.

With such widely deployed sensors, we can now have a worker at one site generating events about some activity a host is engaged in and then, later in the day, another sensor somewhere else in the enterprise generating events about that same host. While neither of these activities are independently interesting, the manager can see both of them and, perhaps, realize that there is something important happening!

IOC and EOI via Behavior-Based Alerting

- Single-event examples:
  - Extremely large MIME attachment
  - Number of bytes for HTTP(S) upload significantly greater than number of bytes for HTTP(S) download
  - md5 hash of a transferred file is the same as the md5 hash of a known malicious file
- Series of related events example:
  - User on a workstation fails to authenticate multiple times
  - Same system later accesses multiple file shares
  - On each file share, the user seems to be retrieving all available files.

The power of Snort lies in its capability to do content matching for signs of *indicators of compromise (IOCs)* or *events of interest (EOIs).* Yet it cannot assess other aspects of the content or behavior patterns that can signal interesting activity. For instance, an extremely large MIME attachment in SMTP traffic could be an indicator of a malicious file. Detection requires examination of the length of the file, something Snort cannot do. Or take the example of the anomalous behavior of more upload than download bytes in HTTP(S) traffic, which could be an event of interest. A final example is comparing the md5 hash of a transferred file with the md5 hash of a known malicious file.

Another capability of Zeek that distinguishes it from Snort is discovering a related pattern of behavior in the network traffic that might be an IOC. Consider this example. Perhaps a workstation reports a number of logon failures for the local user. Eventually, the user successfully authenticates. Shortly thereafter, that workstation then begins connecting to every file share in the enterprise. While the user has the rights to do this, it is somewhat alarming that he or she is connecting to *every* file share, especially after having several failed authentication attempts. Next, that user seems to be retrieving a copy of every file on every file share! This certainly seems like something we would want to investigate. Please take note: There is no way to create a signature to identify these correlated events.

Zeek can identify and note all the above traits or behaviors. These require the users to implement detection of the activity by writing Zeek scripts or by analyzing Zeek logs.

Event Driven: Difference Between a Signature and an Event?

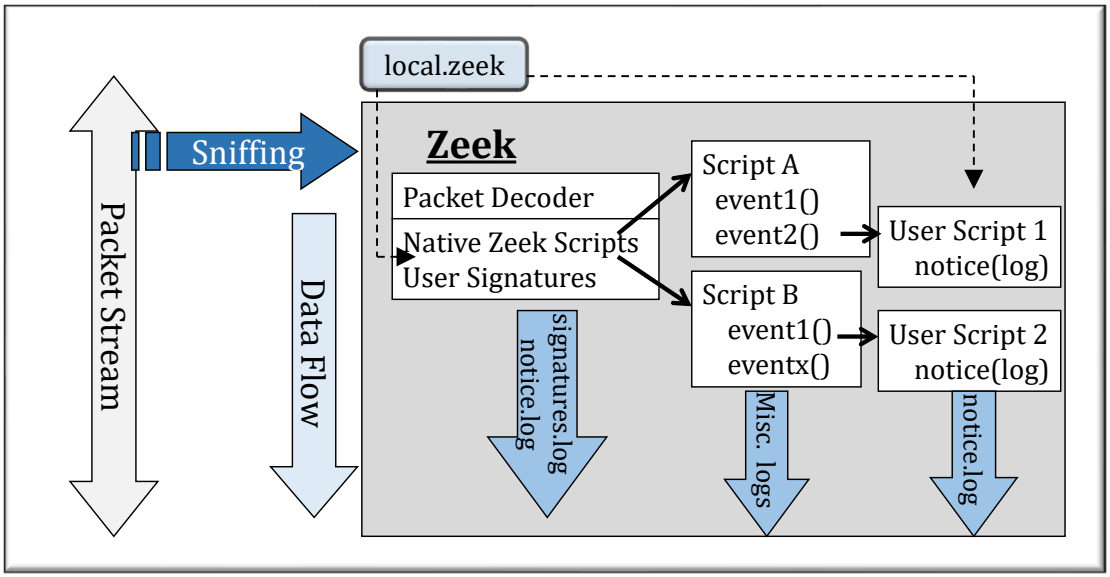
- Signature defines static properties of packets
- Event defines:
  - State:
    - TCP connection established
  - Condition:
    - Protocol violation
  - Protocol characteristic
    - IPv6 DNS reply
  - Availability of data:
    - Identification of software version

The difference between Snort rule detection and Zeek event detection needs some explanation. A signature defines static properties of a packet or stream that can be found in the packet headers or payload.

A Zeek event can define several different behaviors or characteristics of network traffic. A user can create a script that can be triggered when a particular event is found. This lets the user insert unique processing that will be performed upon triggering a Zeek event. Compare this with the purpose and capability of Snort, where matching packet(s) and rule header and payload values trigger an alert. No inherent post-processing can be performed.

It's difficult to categorize the breadth of what Zeek events encompass. Some of the categories are events that are triggered by a given state, such as a complete or partial connection establishment. An event can be triggered upon some condition such as a protocol violation. Others are more closely related to what Snort signatures do, identifying a particular characteristic such as a DNS request or an HTTP response. A final, but not all-inclusive example, is that an event can signal the availability of some data, such as a version of software that might be of interest, like outdated and vulnerable SMB version 1.

Zeek Process Flow



Don't think of events as alerts or even as something that should not occur. When an event is logged, Zeek is simply making a record of something interesting. However, a user can write a script to do something after a particular event is triggered. This is optional code.

Typically, a user script contains logic examining a particular aspect of the event or the data provided by the event. As an example, it might look for a DNS query for address resolution of www.evil.com. Zeek processes the packet and follows the event-triggered code to analyze a DNS packet. On the way, it encounters Zeekdefined events, including the fact that a DNS query has occurred.

This event causes Zeek to log the DNS packet data in the **dns.log**. In addition, Zeek knows of the existence of a user script that is subscribed to the **dns_query** event. The script references the **dns_query** event and contains the post-event processing of matching the query seen in the traffic with **www.evil.com**.

Upon matching, the user can perform any action, perhaps raising a notice to log the traffic. This allows the user to customize the action and log message, assigning it a notion of risk, importance, danger, and so on specific to the site. As Zeek is processing a packet/stream, it is also matching characteristics with those found in any user-created signatures. A signature that fires creates a log entry in the **signatures.log** file. Signatures are not often used but are mentioned for completeness. We will use them for demonstration purposes to introduce a method of triggering an event that will execute Zeek scripts that we will create.

Logged data from events is recorded in various files depending on the activity. Additionally, the notices that are raised by the user scripts are often recorded in **notice.log**, though the script author can choose to write them to any log, perhaps even defining his or her own log. Zeek native scripts will also record entries in the **notice.log**; however, these tend to be more informational, not necessarily indicative of important activity.

All the scripts and signatures that are loaded upon startup in mode when sniffing traffic are found in a file called **$PREFIX/share/zeek/site/local.zeek**, where $PREFIX represents the directory where Zeek is installed on your class VM.

Zeek Log Output

- Several logs are generated by Zeek as default output:
  - Tab-delimited columns of values that can be fed into Zeek "cutting" or post-processing of your choice
  - Each log reflects different activity depending on the traffic observed and frameworks employed.
  - Correlation of traffic from same session/stream is possible with the shared UID field
- Default location of logs is $PREFIX/logs on manager or log host
  - Currently, working directory is the default location for creating logs when running Zeek in file readback mode
- Excellent log reference:

  http://gauss.ececs.uc.edu/Courses/c6055/pdf/bro_log_vars.pdf

Logs are the default output for Zeek reporting. Values in the logs are tab-delimited into associated field columns. The logs generated depend upon the activity that is observed as well as customization that you might use, such as the Notice framework or signatures, for instance. Any output from a particular connection is assigned a common *Unique ID* (*UID*) that is logged in every log where this connection activity appears. This makes it easier to correlate activity for a given connection by searching for that specific UID in all of the logs.

Invoking Zeek in mode causes it to place all of its log files/directories in **$PREFIX/logs**. There is a directory named yyyy-mm-dd (with the appropriate year, month and day values) that houses all the log files per day. By default, Zeek rotates logs on a daily basis, although this value is configurable in **$PREFIX/etc/zeekctl.cfg**. The current day's activity is found in **$PREFIX/logs/current**, although it is really stored in **$PREFIX/spool/zeek** in standalone mode. There is a symbolic link from $**PREFIX/logs/current** that points to $**PREFIX/spool/zeek**.

Zeek generates new log files every hour, assigning each a filename that reflects the type of log followed by a time range indicating when log recording began and stopped. Normally, this is the beginning and end of the hour, though if you happen to stop and then restart Zeek, two different sets of logs are created. The first one reflects the set of current logs that are saved with a timestamp range of when Zeek was started, usually the beginning of the hour, and an end timestamp of when Zeek was stopped. A new set of logs is created with a name that will later contain a time range of the Zeek restart time and the time reflecting when the hour was up or when Zeek was stopped before the hour was up. Zeek compresses each of these hourly files at the end of the day.

Many logs have the same fields, such as the source and destination IP addresses and ports. Protocol-specific ones like **http.log** have unique fields such as the URL. Suppose you saw a connection from 192.168.11.11 that looked suspicious. You could search in all the logs for that particular IP to gain more insight into the related activity.

```
sans@sec503:/sec503/Demos$ cat ./http.log | zeek-cut -u ts uid id.orig_h id.resp_h uri
2000-01-01T13:46:09+0000        CXFEx21mPumKhZ8kvb      173.255.224.88  10.100.100.111  is all my files, identity information
, company secrets, passwords, etc.  Leaving for hacker paradise
2000-01-01T13:51:19+0000        CZb1144AIv6jwg8jei      173.255.224.88  10.100.100.200  /img/pfqa.php
2000-01-01T13:53:34+0000        CnMJ4H2E54hGnHGW0l      173.255.224.59  10.100.100.100  /img/pfqa.php
sans@sec503:/sec503/Demos$
sans@sec503:/sec503/Demos$ cat ./http.log
#separator \x09
#set_separator  ,
#empty_field    (empty)
#unset_field    -
#path   http
#open   2018-11-15-12-16-19
#fields ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p       trans_depth     method  host    uri r
eferrer version user_agent      request_body_len        response_body_len       status_code     status_msg      info_code   i
nfo_msg tags    username        password        proxied orig_fuids      orig_filenames  orig_mime_types resp_fuids      resp_
filenames       resp_mime_types
#types  time    string  addr    port    addr    port    count   string  string  string  string  string  string  count   count
count   string  count   string  set[enum]       string  string  set[string]     vector[string]  vector[string]  vector[string
]       vector[string]  vector[string]  vector[string]
946734369.870189        CXFEx21mPumKhZ8kvb      173.255.224.88  33187   10.100.100.111  8888    1       This    -       is al
l my files, identity information, company secrets, passwords, etc.  Leaving for hacker paradise -       -       -       0   0
-       -       -       -       (empty) -       -       -       -       -       -       -       -       -
946734679.870189        CZb1144AIv6jwg8jei      173.255.224.88  1492    10.100.100.200  80      1       GET     trughtsa.com/
img/pfqa.php    http://trughtsa.com/    1.1     Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)      0       1146    200 O
K       -       -       (empty) -       -       -       -       -       -       FkryGf1PH7KiQeBS0g      641.pdf application/p
df
946734814.870189        CnMJ4H2E54hGnHGW0l      173.255.224.59  1699    10.100.100.100  80      1       GET     trughtsa.com/
img/pfqa.php    http://trughtsa.com/    1.1     Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)      0       1146    200 O
K       -       -       (empty) -       -       -       -       -       -       FCy0Pqsg2qdu4QMvf       641.pdf application/p
df
#close  2018-11-15-12-16-19
sans@sec503:/sec503/Demos$
```

It's often easier to look at examples than it is to talk about the theoretical. Here you can see an example of an http.log file. This file, along with a number of other logs, was generated by running the **phishing-attack.pcap** file from the Section 1 demos through Zeek by running **zeek –r phishing-attack.pcap**. You can find the resulting log files in the **Demos/phishinglogs** directory on the VM.

Notice at the top of the slide, we stream the log file through the **zeek-cut** tool using **cat**. The zeek-cut tool has options that allow you to change the interpretation of the timestamps and other fields, in addition to allowing you to select which fields from the log you would like to see. If you simply process the log using zeek-cut, the default is to output all of the columns with their titles.

The output at the bottom of the slide provides a view of the unprocessed, raw log file. Notice the first lines of the file, each of which begins with a hash mark (#). These header lines define the format of the log file, including the names of all of the columns (which has a box around it). You will find that the documentation for what is in the logs that Zeek produces is scanty; the best way to figure out what's in the log is by looking at the header line. The reason is that Zeek scripts can modify and add fields to the logs, making documentation instantly obsolete.

Stitching Together Events

The UID is very much like the ID field in a database table; you can use it like a foreign key to see all events generated for a given session

```
sans@sec503:/sec503/Demos$ grep CgBJ3mFq8NJg0kbib *.log
conn.log:1546350358.870189  CgBJ3mFq8NJg0kbib    173.255.224.88 33187   10.100.100.111
8888    tcp http    307.000000 120161  0   SF  -   -   0   ShADaFf 55  123029  55      2868    -
http.log:1546350369.870189  CgBJ3mFq8NJg0kbib    173.255.224.88 33187   10.100.100.111
8888    1   This    -   is all my files, identity information, company secrets, passwords, etc.
Leaving for hacker paradise   -   -   -   -   0   0   -   -   --  (empty) -   -       -       -
-   -   -   -   -
weird.log:1546350369.870189   CgBJ3mFq8NJg0kbib  173.255.224.88 33187   10.100.100.111
8888    unknown_HTTP_method    This    F   zeek    -
```

Note that the UID is ***not*** a hash! If you run Zeek multiple times against the same capture, you will see a different UID!

To correlate events within a single session, the UID field is used to find related log entries. In a very real sense, you can view the UID value very much like the ID column in a relational database. This ID is used as a foreign key field to associate data.

Looking at the example in the slide, we have chosen to find all events generated in the session with the UID of CXFEx21mPumKhZ8kvb. Using **grep**, we can see that this UID appears in three different logs, all of which are related to this same connection.

When first using this tool, it is easy to assume that the UID is some kind of a hash. It is not! In fact, if you think about it, it can't be. If we were to use a hash, perhaps based on the tuple of data including the originating host, responding host, originating port, responding port, and the protocol, what would happen if there were another connection some time later involving the same hosts, protocols, and ports? We would get the same ID! Therefore, you should view the UID as a random unique value that is assigned to the session. In fact, if you run Zeek against the same packet capture multiple times, you will find that the UIDs change *every single time*.

Sample of the Types of Logs Generated

Log Name	Purpose
conn.log	Initial IP/protocol connections
pe.log	Identifies portable executable files found
known_hosts.log	New hosts seen in past hour
known_services.log	New services seen in past hour
dpd.log	Dynamic protocol detection
weird.log	Anomalous activity
loaded_scripts.log	Scripts loaded upon start/restart
reporter.log	Severity of issues with Zeek
software.log	Version numbers of vulnerable application layer software
Various protocols (http, dns, ssl, smtp, etc.)	Logs with relevant activity pertaining to a given protocol

Zeek will only generate the log if the protocol is in the data and the parser has been loaded!

Logs are numerous, and you should become familiar with the types of content along with the potential for use in host or network forensics. We'll discuss many of these in more detail in upcoming slides. As a reminder, the $PREFIX reference in a filename refers to the install directory.

The **conn.log** creates an entry for the initial connection of TCP and UDP traffic and summarizes some characteristics, such as the number of packets and bytes generated by either side of the connection.

The **conn-summary.log**, created in mode, provides a very useful summary of connection activity for the duration of time that the log spans, usually an hour.

The **known_hosts.log** records are created in mode for every completed TCP handshake and keeps track of IP addresses in use on a network each day.

The **known_services.log** records are created in mode for an IP and port that responded to a SYN, and protocols detected in the session are also logged per day.

The **dpd.log** records protocols discovered on nonstandard ports by using Zeek's Dynamic Protocol Detector (DPD), which works by using "heuristic methods."

The **weird.log** is a catch-all log for anomalous behavior.

The **loaded_scripts.log** lists the Zeek scripts that were loaded upon start/restart from the supported subdirectories of **$PREFIX/share/zeek**: **base**, **policy**, and **site**.

The **software.log** records the version numbers of potentially vulnerable application layer protocol software that is seen in the network data.

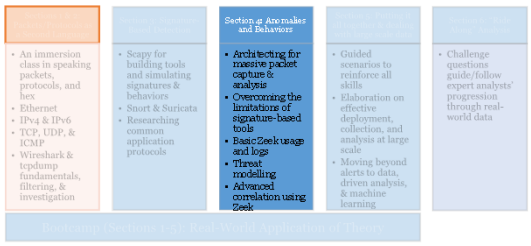
The **reporter.log** reports on Zeek status errors. There are several ratings: "informational," which, as the name implies, is for information; "message," which signals a potential problem; "warning," which indicates a nonfatal but definite problem where Zeek doesn't terminate; and "error," which is a fatal error that terminates Zeek.

There are various protocol-specific logs for protocols that Zeek detects, such as **http.log**, **dns.log**, etc. Take note that Zeek will only produce a log if there are packets within the data source that contain the given protocol. Also, while most protocols are loaded by default in Zeek, support for experimental or especially chatty protocols can be enabled by reconfiguring the defaults.

Zeek and Zeek Output Exercises

Workbook exercises "Running Zeek and Zeek Output"

This page intentionally left blank.



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

Introduction to Zeek

Zeek and Signatures

Zeek Scripting Basics

Threat Analysis

Scripting Correlations

Scripting Behavioral Anomalies

Spicy

Customizing Zeek: Policy Neutral to Site Specific

- A major goal when developing Zeek was to make it policy neutral.
- Several means of allowing customization to site-specific policy:
  - **Scripts**: Scripting language to react to and correlate between events
  - **Signatures**: Simple rules to find noteworthy activity and trigger scripts
  - **Notice capability**: Post-event processing enables you to associate a notion of importance to events and perform some kind of action.

Zeek *can* use signatures, but it's really about events. We start with signatures because they are familiar.

A primary philosophy that the developers of Zeek felt was fundamental is what they deem *site-neutral policy*. This permits a site to customize evaluation of any given event to determine its significance and what action to take as a result of that evaluation. Snort, on the other hand, has a one-size-fits-all philosophy of ascribing a priority to each rule/alert, thereby assigning a universal value reflecting the potential significance of the detected activity. This notion of priority may be very different, depending on acceptable activity per site.

There are three basic ways to customize Zeek's detection and post-detection activity. The first way is by using the Zeek scripting language. One description of this language by one of Zeek's developers, Robin Sommer, in his presentation "The Open Source Zeek IDS: Overview and Recent Developments" found at http://www.icir.org/robin/slides/Bro-CACR-Indianapolis.pdf is "A Framework for Network Traffic Analysis." It permits the user to develop basic or very advanced processing after one of Zeek's events is triggered. The Zeek scripting language was developed expressly for the purpose of analyzing network traffic, as opposed to some more general language like Python that needs to retrofit libraries or routines to facilitate network traffic analysis. This scripting language enables a given site to create code for what is considered noteworthy traffic on that particular network. This is the primary method Zeek uses to expose the detection process to the user.

The second way to customize policy and detection is via signatures. Using Zeek signatures is a very basic method of identifying and recording or reporting on some pertinent characteristics about the traffic, mostly based on content, and is modifiable by such characteristics as traffic IP addresses and ports. You have some limited choices to specify the location of the content, say, in the entire payload, or some more protocol-specific location like the HTTP header. Regular expressions are supported to describe the content that is sought.

The final way to customize Zeek for your site is to use the *Notice Framework* to *raise a notice*, somewhat like a Snort alert. It is a means of defining actions for notices, such as logging and emailing. We will not discuss the Notice Framework; it is included here to let you know that it exists should you care to use it.

Zeek Signatures

- Simple rules to define:
  - Conditions to match
  - Action to perform upon match
- Different types of conditions:
  - Header: Examine specific header fields/values
  - Content: Look for particular payload
- Action:
  - Place notice in signatures.log
  - Trigger a user-written Zeek script

As you know, Zeek supports the notion of signatures. Snort rules provide much more functionality than Zeek signatures because Snort rules are considered the primary way that noteworthy activity may be found. Because Zeek has a scripting language that is capable of performing many of the more advanced features of Snort rules, its signature support is lightweight in comparison. So do not expect the same level of functionality in Zeek signatures.

Zeek signatures define conditions to match in network traffic and the action to be performed upon matching. Zeek defines several types of conditions that can be used in signatures; we'll look at the two most commonly used. The first is known as the *header* condition, which is capable of examining source and destination IP addresses, a list of IPs, is designated either as a single IP address or in CIDR format. Also, it can match on source and destination ports or a port list. And finally, it can match on protocols IP, IPv6, TCP, UDP, ICMP, and ICMPv6.

A second type is the *content* condition, which is most often combined with a header condition. The content is designated using a regular expression. The content can be matched with the entire payload using the **payload** designation, or it can be matched against some HTTP designations such as **http-request**, **http-request-header**, etc.

A signature match raises a signature event in the **signatures.log** and **notice.log** file containing pertinent data about the packet/stream that caused the signature to match. A Zeek script can be invoked from a matching signature, as will be demonstrated in an upcoming slide.

As mentioned, we cover Zeek signatures not because of the outstanding capabilities, but as a learning tool for the much more useful capabilities of Zeek scripts. A signature triggers an event, which can be an easy way to demonstrate how to execute a Zeek script that is subscribed to a signature event.

A more comprehensive description of signatures can be found in the following Zeek signature documentation:

https://docs.zeek.org/en/master/frameworks/signatures.html

Signature Components

- Specify header:

  <header field><comparison operation><value list>
  - Header field
    - Source/destination IP address (single, list, CIDR notation)
    - Source/destination port (single, list)
    - IP protocol: ip, ip6, tcp, udp, icmp, icmp6
  - Comparison operator: ==, !=, <, <=, >, >=
  - Value list
    - Single or multiple values
- Specify content:
  - Payload
  - Defined HTTP fields

When using the header part of a signature, you can match three header fields: IP addresses, ports, and the protocols **ip**, **ip6**, **tcp**, **udp**, **icmp**, and **icmp6**. Source and destination IP addresses can be expressed as a single IP address, as a list of IP addresses, or in CIDR notation. Source and destination ports allow a designation of a single port or list of ports. Finally, you can examine traffic for any of the protocols available.

The comparison fields are relatively straightforward—equal, not equal, less than, less than or equal, greater than, or greater than or equal. A comparison value must be supplied.

The content portion of a signature defines what value(s) are sought in the payload or some specific defined HTTP fields. Unlike Snort, where multiple content keyword and value pairs can be defined, Zeek allows a single expression of the content. And unlike Snort, Zeek has no facility for defining the content offset other than via the use of regular expressions.

Zeek requires the content to be defined using a regular expression unless you supply the entire content of the payload. Take the case where you'd like to find a string of "evil" in the payload of "this is evil stuff". A Snort rule would trigger on **content: "evil",** assuming the rest of the rule conditions matched. A Zeek signature needs to know if anything precedes the content, then stops the search upon finding the string "evil". A Zeek signature would define the content search in the signature as **/.*evil/,** where the regular expression syntax for a wildcard ".*" specifies that there is some content before "evil".

Signature for Windows Command Line Using Regular Expression

```
signature cmd-exe {
   ip-proto == tcp
   dst-port == 4444
   payload /^Microsoft Windows.*Copyright \(c\).*Microsoft Corporation/
   event "Windows Command Line Banner using Metasploit"
}
```

Header

Payload

Beginning of line followed by "Microsoft Windows"

Wildcard

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved
```

Followed by "Copyright (c)"

Wildcard

Followed by "Microsoft Corporation"

windows-banner.sig

The next several slides will demonstrate how to use Zeek to detect what we assume is malicious traffic. Consider the example that we used in the slides in the book covering signature-based detection. There we were looking for connections involving port 4444 and content strings indicating a Microsoft command line was started by looking for the banner. We'll begin with a Zeek signature, find that it does not do what we want, and accomplish detection using a Zeek script.

This shows a Zeek signature with the same characteristics as the Snort rule. The protocol is TCP, and the destination port is 4444. So far, so good.

We are attempting to use Zeek to replicate the multiple **content** options in Snort, yet without the precision, by using a regular expression in the signature **payload** value. The regular expression syntax "^" means that the content is found at the beginning of whatever protocol data unit Zeek is analyzing. We follow that with the text "Microsoft Windows" and then append a wildcard designation of ".*" to account for the unknown or unpredictable content. Next, we add the "Copyright (c)" string.

We need the escape character backslash before the parentheses in regular expression syntax to designate that we are looking for the string value. Otherwise, there is confusion with the syntax because regular expressions use parentheses to group values together. We finish the regular expression with another wildcard and the content of "Microsoft Corporation".

The Snort rule language has the capability to easily add efficiency by restricting the depth of part of the content. It is possible to make this Zeek signature more efficient using some additional features of regular expression syntax; however, that is beyond our current scope. Zeek itself has no features to perform this same function.

Contents of signatures.log

```
sans@sec503: zeek -r inbetween.pcap -s windows-banner.sig


sans@sec503: cat signatures.log | zeek-cut -d ts uid src_addr src_port dst_addr dst_port
sig_id event_msg sub_msg
ts                              uid                 src_addr        src_port
2016-06-16T09:21:38-0400        CwzULKmpDoNMugby    192.168.11.17   1770


dst_addr      dst_port   sig_id        event_msg
192.168.1.8   4444       cmd-exe       192.168.11.17: Windows Command Line Banner using
                                       Metasploit


sub_msg
\x0d\x0aCopyright (c) 2009 Microsoft Corporation.  All ri
reserved.\x0d\x0a\x0d\x0aC:\\Users\\judy\\Downloads>
```

We will return to our signature and automating a response in a while...

We can perform a simple run to test the signature by invoking Zeek with the **-s** command-line switch specifying the filename containing the signature (in this case, a file we named **windows-banner.sig).** Once again, many logs are created, but for our purposes, we want to look at the **signatures.log** that is created when a signature is triggered. Whenever you run Zeek in readback mode, be aware that Zeek creates the log files in the current working directory. It is often more beneficial to create a separate new directory to contain all the log files generated for the run.

The **zeek-cut** command is used to show some of the content of **signatures.log**. Much like the other log files, the **signatures.log** file has a timestamp for the activity that is normalized using the **zeek-cut -d** option as well as the source and destination addresses and ports. The field **sig_id** is the name of the matching signature, **event_msg** is a descriptive message of the signature-matching event, and finally, **sub_msg** is defined as "extracted payload data or extra message." The payload seen from this field is not the entire matched signature content. The payload is split between two packets, and it appears Zeek dumps the last one only.

It is interesting that **signatures.log** and the signature parameter names themselves have different field names for the same fields found in many of the other logs. For instance, **src_addr** represents the source IP address in **signatures.log** but is known as **id_orig.h** in the others.

By default, Zeek searches the first 1024 bytes only for a signature match. A variable called **dpd_buffer_size** stores this configurable value and is found in **$PREFIX/share/zeek/base/init-bare.zeek**.

To run the Zeek commands on the slide, execute the following:

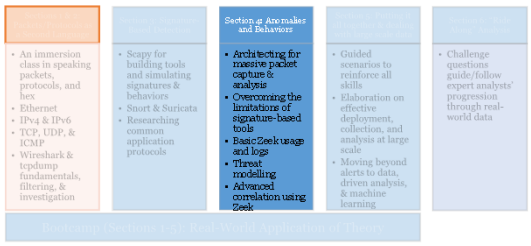
zeek –r inbetween.pcap –s windows-banner.sig

cat signatures.log | zeek-cut -d ts uid src_addr src_port dst_addr dst_port sig_id event_msg sub_msg

Zeek and Zeek Signatures Exercise

Workbook exercises ”Zeek Signatures"

Let’s try to put this basic signature technique into action with a quick exercise!



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

Introduction to Zeek

Zeek and Signatures

Zeek Scripting Basics

Threat Analysis

Scripting Correlations

Scripting Behavioral Anomalies

Spicy

Zeek Scripting

- Full-featured language with many of the same capabilities as higher-level languages
- Offers network-related representations of traffic: hosts, ports, and protocols
- Permits you to write your own code or customize existing code for site-specific purposes
- Complete understanding of this complex language is beyond the scope of the class; we just touch on the basics
- Default scripts found in the directory **$PREFIX/share/zeek/base/**

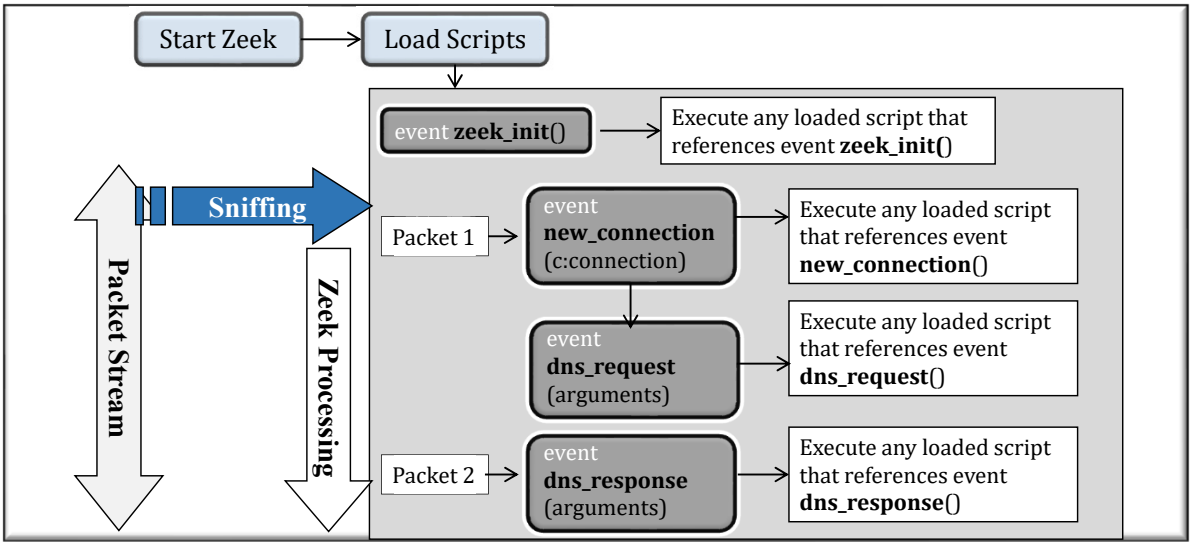
One of the most useful and advanced features of Zeek is its own scripting language that was predicated upon interpretation of network traffic rather than importing a library, as a more generic language such as C does. This means that you are likely to find far more functionality and native support for network protocols that you would like to process.

The language offers the analyst a platform to customize code. We will cover some basic concepts associated with scripting and create some useful scripts in our lab exercises. But this is by no means comprehensive. An entire week, if not more, could be devoted to teaching the Zeek scripting language and its applications. You may not be interested in learning Zeek scripting because of its complexity or because you don't require this degree of customization. Many users never learn to write customized code for Zeek, yet find benefit using Zeek, particularly the logs that Zeek generates. It is useful to know the scripting language is available, and the material that follows will assist you in creating and executing basic scripts.

The Zeek documentation available, especially on scripting, is scattered and makes assumptions that you are aware of the environment in which scripts are run. It does not take you step by step, demonstrating how to start and how to proceed as the following material intends to do.

The scripts that come with Zeek are installed in the **$PREFIX/share/zeek/base** directories with a file extension of **.bro**. These scripts are not intended to be changed; if the user wants to make customizations, those user-created scripts should be placed in the directory **$PREFIX/share/zeek/site**. The directory **$PREFIX/share/zeek/policy** contains scripts that are considered to be less universally required and may be more computationally intensive; therefore, the user must elect to load them. It should be mentioned that the site-specific scripts need to be identified to Zeek; this is done in the configuration file **$PREFIX/share/zeek/site/local.zeek**.

Zeek Event Processing Flow



The notion of a Zeek event is powerful because it presents an opportunity for customization by adding your own scripts. If you decide that you'd like some custom processing, you need to understand that a Zeek event is your vehicle for identifying the trigger condition to execute your code. Let's scrutinize the process more closely.

Upon starting in mode, Zeek examines the **$PREFIX/share/zeek/site/local.zeek** file for entries that begin with **@load *filename***, where **@load** is equivalent to a Snort configuration file or C programming language **include** statement in that it references specific files, in this case, Zeek scripts that should be loaded. The filenames can be those included with Zeek or those that you create. These files/scripts reference a particular Zeek event found in **$PREFIX/share/zeek/base/bif/event.bif.zeek, $PREFIX/share/zeek/base/bif/plugins/zeek*bif.zeek,** or **$PREFIX/share/zeek/base/protocols/*protocolname*/main.zeek**, along with the code that is to be executed upon Zeek's processing of that event.

For instance, in our slide example, Zeek is started and in its processing encounters the **zeek_init** event, as shown in the slide. The event processes any loaded script that references the **zeek_init** event. If, for example, you wanted Zeek to write a message that Zeek started, maybe with a date and time, you would supply a script that references **event zeek_init** to accomplish this. You would place the full filename of that script in the **local.zeek** file using **@load filename**, and it would be loaded after being installed and restarted in mode. Now, after Zeek starts and determines that the **zeek_init** event occurs, your script is triggered and executed.

Each packet sniffed thereafter may cause Zeek to perform different code execution, perhaps based on the protocol. Suppose packet 1 is a DNS request and packet 2 is a DNS response. Zeek will decode the packets and determine that they contain the DNS protocol. Say this is a new never-before-seen connection. If there is native Zeek code or a script that you added to be processed upon seeing a new connection, it will be processed when the event **new_connection** condition is encountered. Eventually, Zeek will follow the code for processing DNS. Now, all events that are associated with DNS can trigger an associated user-written script.

That's a lot of dense detail crammed into one slide, so here are the key points summarized.

First, a Zeek-defined event is the entry point for adding custom code. An event can denote an activity or state that Zeek defines. It does not mean that something malicious has occurred.

Second, not all Zeek-defined events will be processed for every packet. An event is triggered when event-specific conditions exist, such as when a new connection is detected or when a particular protocol is encountered. Zeek-defined protocol event conditions are met only when the protocol is detected and other specifications are met.

Third, a user-written Zeek script is subscribed to a particular event that the user identifies when creating the script. The occurrence of this event triggers execution of the script.

Intro to Scripting

- The hardest thing to get about Zeek is that it doesn't react to *packets;* it exposes *events*.
  - To access and handle these events, we must write scripts.
  - That isn't nearly as difficult as it might sound!
  - It uses a Publisher-Subscriber design pattern.
- What you must identify to run a Zeek script:
  - Name of Zeek-defined event that will trigger your script to run
  - Name of parameters and parameter attributes

The biggest mental adjustment you will need to make to begin to use Zeek effectively is understanding that we are no longer reacting to *packets* but are instead reacting to *events*. While it is true that there is an event generated every time a packet arrives, Zeek scripts generally should not be making use of this event (except, perhaps, when we're debugging a script or are in an offline mode during an investigation) since this event will be *very* expensive.

So, what is an event, then? Think about the things we've discussed this week. In fact, the time you spent building fluency in the language of packets and protocols will really help you here. We have a notion that there can be a thing called a "connection." Would it surprise you, then, that Zeek will generate an event, called **new_connection**, whenever a new TCP or UDP "connection" is seen? We also saw things like the connection to a Microsoft share. Zeek exposes an event (**smb2_tree_connect_request**) that will be dispatched every time there is a connection to a share or named pipe within SMB traffic.

Why does this matter, and why is it powerful? Let's look…

Would This Be Harder in Wireshark?

```
sans@sec503:/sec503/Demos$ analyze backbone | zeek -r - ./smb.tree.zeek
\\DC1.packetrat.net\SysVol
\\DC1.packetrat.net\SysVol
\\fileserver\IPC$
\\fileserver\common
\\fileserver\IPC$
\\DC1.packetrat.net\sysvol
\\DC1.packetrat.net\SysVol
\\DC1.packetrat.net\IPC$
\\DC1.packetrat.net\SysVol
\\DC1.packetrat.net\sysvol
\\DC1.packetrat.net\IPC$
\\DC1.packetrat.net\sysvol
…
```

```
event smb2_tree_connect_request(c: connection, hdr: SMB2:
:Header, path: string)
{
 print path;
}
```

This is the *entirety* of the script that produces this output!

Consider this example. On the left side of the slide, you can see that we are processing the packets from the **backbone.cap** capture file, which is part of the data you have been working with in the bootcamp and which you will make heavy use of during the capstone challenge. We have asked Zeek to read back that capture file and to also load the **test.zeek** script. The entire contents of the **test.zeek** script are inset to the right.

The output on the left side of the slide is showing every SMB share to which there was a connection in the packet capture. The output looks deceptively simple. Hopefully, you can recall how complex a protocol SMB is and the effort that you perhaps had to go to to identify tree connections and find share names when using Wireshark to highlight similar information. Consider that effort and compare it to the four-line script. In that script, the first, second, and last lines are all more or less predefined by the scripting language. The only actual "script" that we wrote was the instruction to print the **path** field.

This should give you just an inkling of why Zeek matters. We will look at a much more complex example in a bit, but first, let's discuss some additional scripting basics.

Non-Linear Flow

- In most scripting languages, the script executes from the beginning to the end.
- Zeek scripting is much more similar to GUI programming.
  - In GUI programming, there is an *event loop* that is always running.
  - Some of your code might run at startup, but after that, your application sits idle.
  - When your program starts, it registers or *subscribes* to events it wishes to see.
  - Whenever an event occurs, the event loop sends the event to the *first responder*.
- Zeek is very similar but slightly different.
  - While you *can* change the priority for event subscriptions, unlike a GUI application, all subscribers are first-class citizens; everyone gets the event!

The idea of generating events is not unique to Zeek. It is a pretty common model when you do GUI, game, or a number of other styles of programming.

Most people who casually write scripts or who learned some basic programming skills in high school or college were taught either a purely imperative language (such as C or Pascal) or an object-oriented language (such as C#, Java, or C++). Unless you took a class that forced you to write a GUI application, it is unlikely you were introduced to event loops. With that in mind, you might expect that when you write a script, the script will begin executing at line one and continue executing, possibly following a variety of flow control directives (like **loops** and **conditionals** such as ***if…then…else*** statements) until it reaches that last line, at which point it will exit. This is *not* how Zeek scripts execute.

Instead, think about it like subscribing to events that are interesting for you. It's almost like you are looking at your TiVo or Hulu management screen, deciding which TV shows you are interested in. While you can watch *any* show, you will likely only tell your DVR to record shows you are actually *interested* in. Effectively, you are *subscribing* to those shows.

Similarly, when you copy the definition of an event out of the Zeek documentation and put it into a script, you are telling Zeek that you wish to be notified whenever that event occurs. In a GUI application, when an event occurs, the topmost application (also known as the first responder) is notified of the event and can choose whether it wishes to handle the event. If it handles the event, the event will generally *not* be available to any other application that is running.

Zeek handles things differently. In Zeek, all scripts are first-class citizens. This means that *every* script that subscribes to an event will receive a copy of the event. If desired, you can force Zeek to trigger the events in certain scripts before others by attaching the *&priority* attribute to the event handler. Higher priority handlers are always executed first. This might be desirable if the handler in one script updates or maintains information that the handlers in other scripts rely on.

What Are the Names of the Available Events?

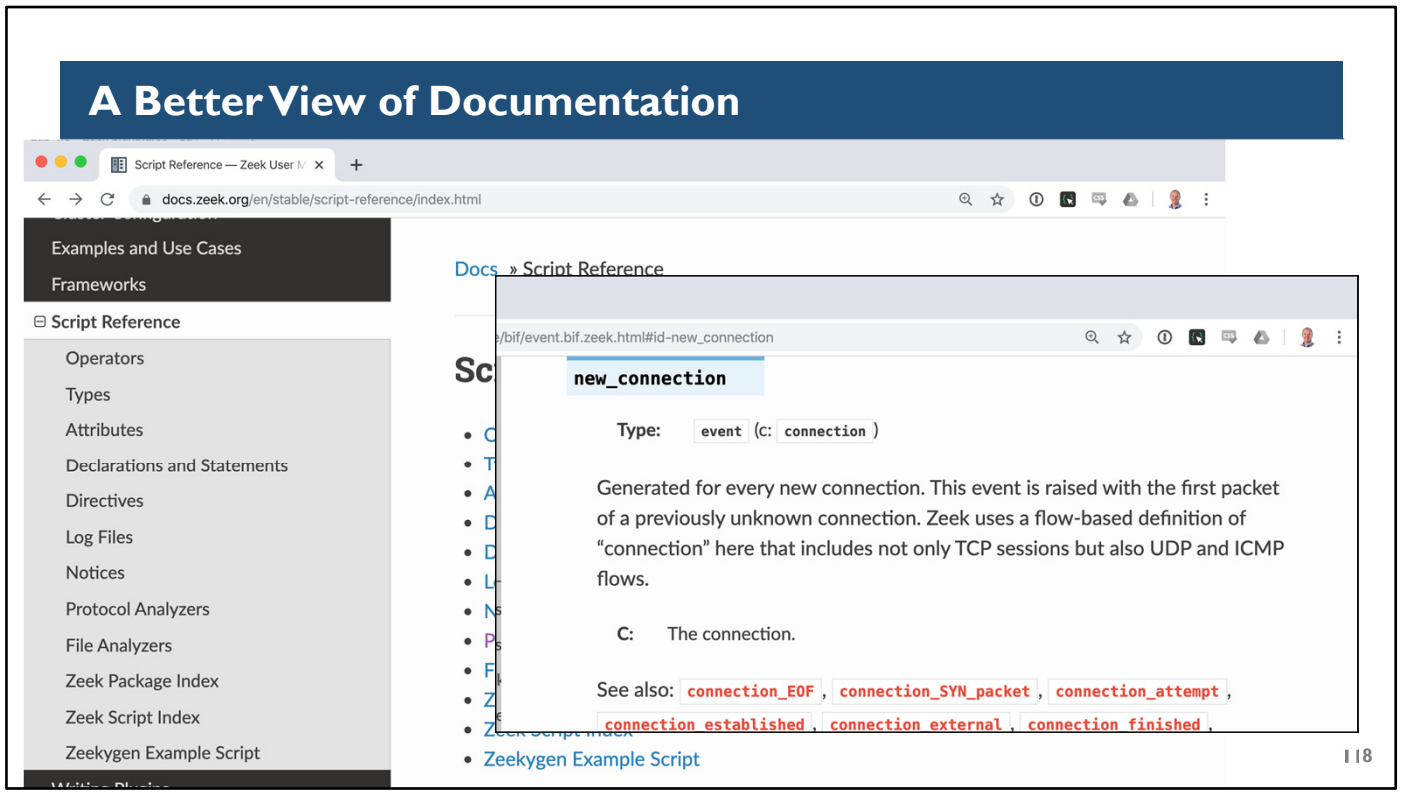
- Events define trigger conditions and an entry point for your customized scripts.
- Found in:
  - $PREFIX/share/zeek/base/bif/event.bif.zeek
  - $PREFIX/share/zeek/base/bif/plugins/Zeek*bif.zeek
  - $PREFIX/share/zeek/base/protocols/*protocolname*/main.zeek

zeek_init:	connection_SYN_packet:	mobile_ipv6_message:
teredo_packet:	tcp_contents:	protocol_violation:
udp_request:	icmp_error_message:	ssl_client_hello:
login_failure:	ftp_request:	ssh_client_version:
smtp_data:	mime_content_hash:	http_all_headers:
netbios_session_message:	dns_AAAA_reply:	dhcp_offer:

Sampling of some native events

There are two main ways to figure out what utility functions and events are available within Zeek. The first way, shown in the slide, is to locate the Zeek installation and examine the .**bif** files. *BIF* stands for "Built-In Function." You might notice that many of these BIFs are actually being loaded from scripts. That's fine; Zeek is designed to be extensible. The point is that scripts and analyzers that Zeek provides to you, whether they are compiled in or are actually scripts, are viewed as “built in” since you can make use of them anywhere, within any of your scripts.

While perusing these files can be very informative, it is probably not the most efficient way to consume the documentation. In fact, the data that you are reading in these files is used to automatically generate web-based documentation.



The far friendlier way of consuming the documentation is using your web browser. While you can instruct Zeek to build the HTML documentation when installing Zeek, most developers simply use the documentation published at **zeek.org**.

From here, you can read about variable types, how to create functions, how to create arrays and sets, how to find events generated by protocol analyzers, and more. Let's use some information from these two sources of documentation to create our first script.

A Simple Script

```
event zeek_init()
  {
   print ("Started zeek");
   }
```

```
sans@sec503:~/tmp/zeekscripts$ zeek -r cmdexe.pcap zeekinit.zeek
Started zeek
```


One of the events that Zeek makes available is **zeek_init()**. This event will be triggered whenever Zeek starts up. Similarly, there is an event called **zeek_done()** that will be called when Zeek finishes processing. Let's try to use one of these for a first attempt at a script.

Suppose we create a file that has a simple script, **zeekinit.zeek**. Scripts supplied in Zeek end with the extension **.zeek**. However, this isn't required; our script simply has the event name **zeek_init** found in the Zeek documentation. This identifies the triggering event for the script to be invoked if Zeek's processing encounters it. As mentioned, scripts can have parameters passed to them, denoted between the parentheses following the event name. The **zeek_init** event has no parameters. We place a Zeek print statement that says "Started zeek" as our single line of code.

Next, we run Zeek using the name of the Zeek script to run for reading in **cmdexe.pcap**. Although we don't need a pcap for this script, we include it to represent a more conventional situation. As you see, we get the output of "Started zeek."

To perform the command on the slide, execute the following:

zeek -r cmdexe.pcap zeekinit.zeek

Inform about a New Connection

```
sans@sec503:~/tmp/zeekscripts$ more zeekif.zeek
```

```
event new_connection(c: connection)
   {
     if (c$id$orig_h == 192.168.11.62 && c$id$resp_p == 80/tcp)
            print fmt("New Connection => orig: %s %s resp: %s %s",
            c$id$orig_h, c$id$orig_p, c$id$resp_h, c$id$resp_p);
     }
```

```
sans@sec503:~/tmp/zeekscripts# zeek -r http.pcap zeekif.zeek


New Connection => orig: 192.168.11.62 19086/tcp resp: 173.194.73.106 80/tcp
```


Now that you have a general idea how to run a script and understand the output that it generates, let's write a script that is more practical and has a bit of logic involved. According to the comments in **event.bif.zeek**, the **new_connection** event is described as follows: "Generated for every new connection. This event is raised with the first packet of a previously unknown connection."

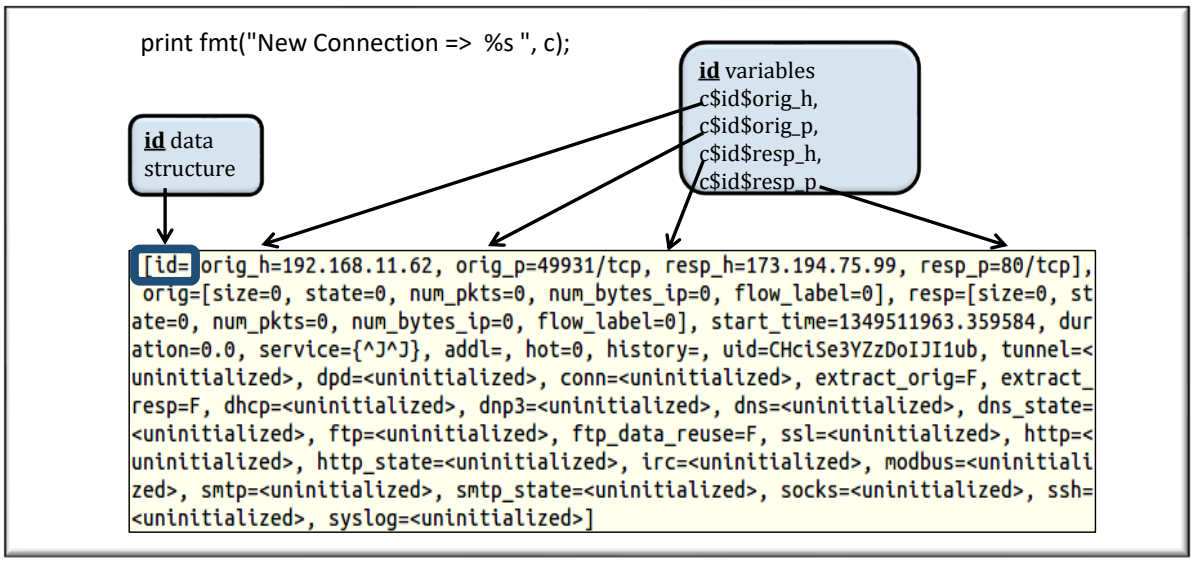
We will postpone a discussion of the variable names used in a script.

We want to find a new connection with the source IP address of 192.168.11.62 and a destination port of TCP 80. Upon finding a connection, we use a formatted print statement much like the ones used in C and other programming languages, where the first part describes the text output and the output format of the associated variables. The "%s" denotes string output for the source IP/port and destination IP/port of the variables that follow in the second part of the formatted print statement. When the script is run using **http.pcap** as input, we see the output of the script on the bottom of the slide.

To perform the command on the slide, execute the following:

zeek -r http.pcap zeekif.zeek

Connection Data Structure



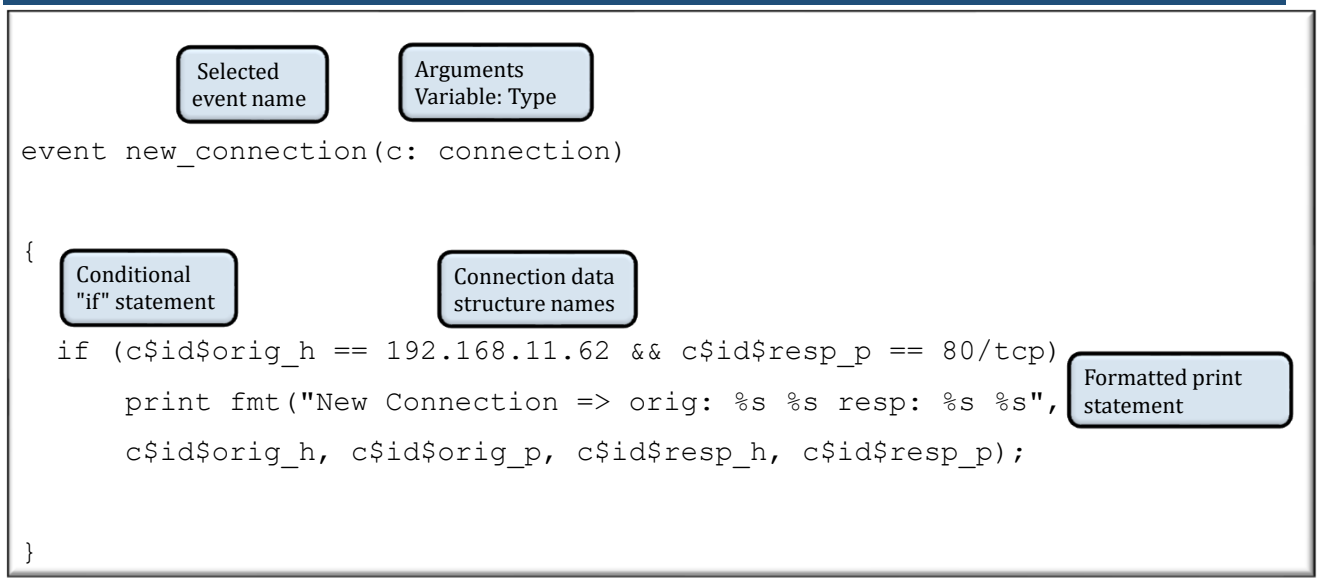
The concept of a Zeek connection is something you need to understand when examining output and writing your own scripts. This means it is necessary to understand Zeek's connection data structure. The script on the previous slide generated the output seen in this slide; it is a formatted print of a new connection known in Zeek as the variable **c**. The script lists the **c** data structure seen in the slide.

A connection is a nested data structure much like we saw in the event **signature_match** script parameter **state**. In order to identify a given variable, you must begin at the base structure **c** and then identify any other nested structures. We see that the **id** data structure contains the variables to identify the source and destination IP addresses and ports. For instance, **cidorig_h** identifies the source IP address. As you will recall, "$" serves to dereference or delimit each of the nested data structures as well as any variable in the final nested data structure.

To see the output on the slide, execute the following:

zeek -r http.pcap zeek-printc.zeek

Let's Examine the Script

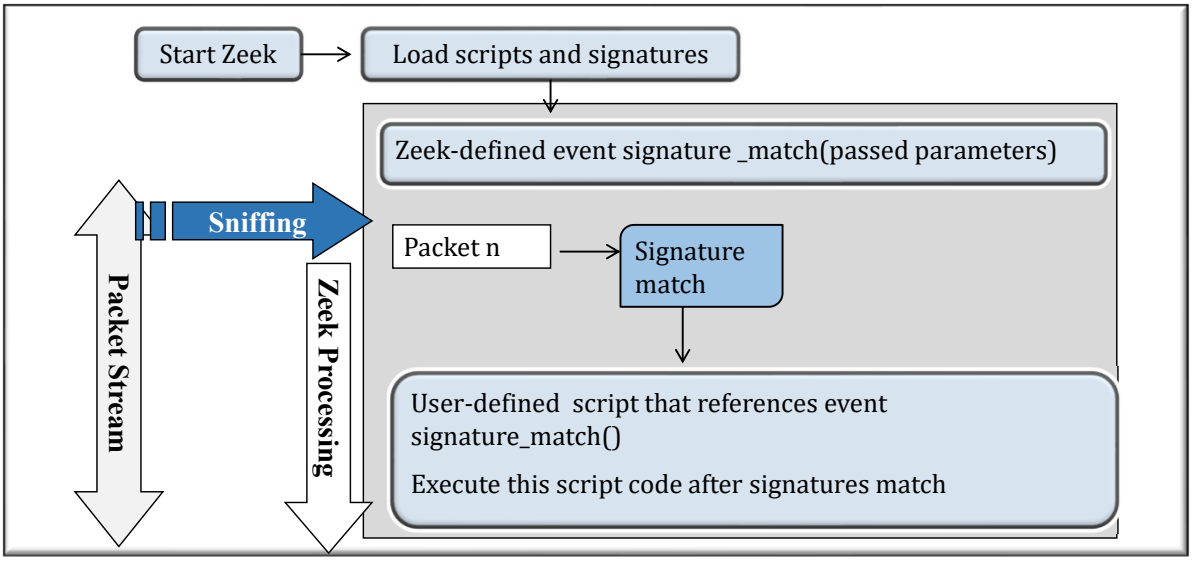


The **new_connection** event passes connection data in an argument named **c** with a Zeek type of **connection**. This is detailed where the event is defined in **event.bif.zeek** and in the documentation. The above script checks the connection to see whether the source IP address **cidorig_h** is 192.168.11.62 and the destination port **cidresp_p** is TCP 80. First, how did we know the names of the variables that Zeek associates with the source IP address and destination port address? There is a file named **$PREFIX/share/zeek/base/init-bare.zeek** that contains the names of all of Zeek's variables associated with any connection.

You'll notice the format of each variable used in this script begins with **cid**. The **c** represents the connection data passed to the function; the **id** represents Zeek's unique identification, UID, for this particular connection. The **c** connection is the base data structure, followed by an **id** data structure that is followed by all the relevant fields. The fields for identifying any new connection are the source IP known as **orig_h**, the source port known as **orig_p**, the destination IP address known as **resp_h**, and the destination port known as **resp_p**.

When we find a new connection with the source IP address of 192.168.11.62 and a destination port of TCP 80, we use a formatted print statement to display the output.

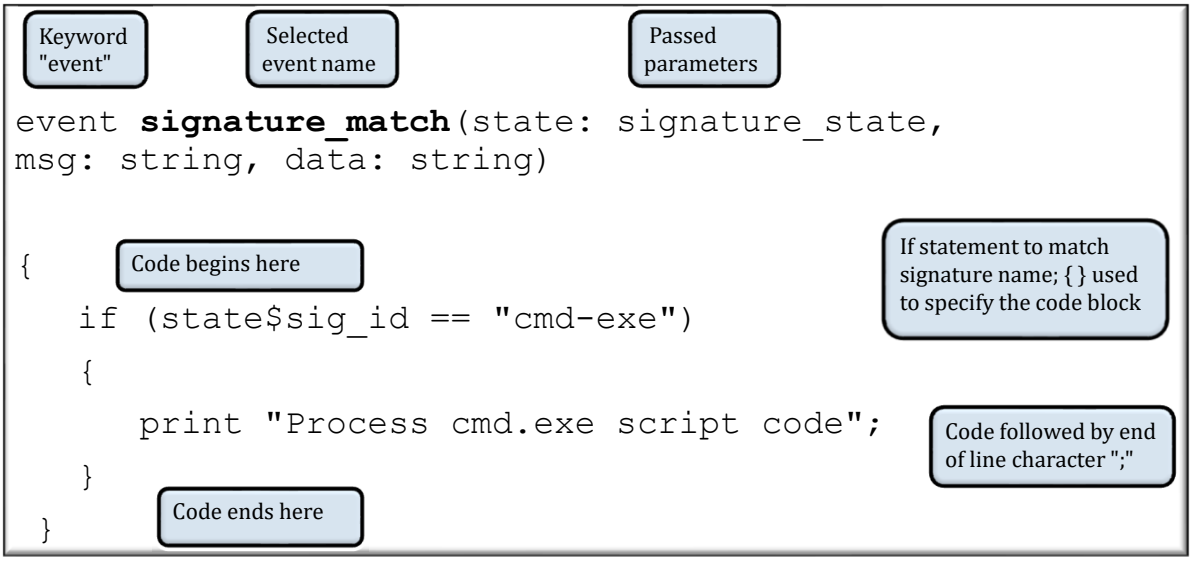
Trigger Script Execution after Signature Match



The concepts of events and associated scripts can be somewhat confusing, so let's try to give the discussion some gentler context. We are going to execute a Zeek script that we write that will be triggered when a specific signature finds a match. As mentioned, Zeek loads some default scripts upon being invoked. One of these scripts is defined as an event named **signature_match**, which is automatically triggered when a signature match occurs. By default, there is no accompanying user-defined script that is triggered by this match or event. Zeek has its own scripts to perform signature-matching processing, such as logging data to **signatures.log**.

But suppose you wanted additional processing to occur when a particular signature found a match in "Packet n" above. We would have to write our own code for the script that would begin with the declaration **event signature_match(*passed parameters*)**, where *passed parameters* are the names and variable types passed; we'll look at those in an upcoming slide. Zeek would execute our code after any signature match.

Format of Our Script



Let's look at the code and format of our script to process upon a signature match. As you are aware, you need to identify an existing Zeek event that will cause your script to trigger—in this case, **signature_match**. This is the event that occurs after a signature matches. It causes Zeek to look for any code with **event signature_match**.

This is accomplished by supplying the keyword **event** followed by the actual name of the event you selected to trigger your script. Any arguments that are to be passed to your script are enclosed between the parentheses. The **signature_match** event has three defined passed parameters: **state**, **msg**, and **data**, which are Zeek variable types of **signature_state**, **string**, and **string**, respectively.

Zeek uses the left and right brace characters to group a block of code; in this example, the beginning and end of the code associated with the event processing. Other examples of related code are those executed after conditional and loop statements. The print statement format is much like those used in other languages. The semicolon is used to identify the end of a line of code.

The script first checks for the unique signature identification, the name we gave to our signature in the file **windows-banner.sig** expressed as **signature cmd-exe**, before we execute the code. We need to test that the signature name that triggered this script is the one we created that we named **cmd-exe**. How did we know to use the variable name **state$sig_id** for the signature identification field? We'll discuss this on the next slide.

Documentation for Zeek scripting can be found at http://www.zeek.org.

Another option for learning syntax and scripting is to examine the scripts that come with Zeek. Like learning a spoken language, the best way to really become familiar with this language is to actually *use* it. You will have the opportunity to do so in the coming exercises.

State Data Structure

```
event signature_match(state: signature_state, msg: string, data: string)
 {
   if (state$sig_id == "cmd-exe")
   {
      print fmt("Structure of event signature_match parameter 'state' ===> %s",
       state);
   }
 }
```

state$sig_id

Entire **state** data structure

```
bro -r inbetween.pcap -s windows-banner.sig sig-event-printstate.bro

Structure of event signature_match parameter 'state' ===> [sig_id=cmd-exe, conn=[id=[orig_h=192.168.11.17, orig_p
=1770/tcp, resp_h=192.168.1.8, resp_p=4444/tcp], orig=[size=129, state=4, num_pkts=3, num_bytes_ip=168, flow_labe
l=0], resp=[size=0, state=4, num_pkts=2, num_bytes_ip=92, flow_label=0], start_time=1466083298.222038, duration=0
.052061, service={\x0a\x0a}, history=ShADa, uid=CW0MXF3hduOeHMT5Nk, tunnel=<uninitialized>, dpd=<uninitialized>,
conn=<uninitialized>, extract_orig=F, extract_resp=F, thresholds=<uninitialized>, dhcp=<uninitialized>, dnp3=<uni
nitialized>, dns=<uninitialized>, dns_state=<uninitialized>, ftp=<uninitialized>, ftp_data_reuse=F, ssl=<uninitia
lized>, http=<uninitialized>, http_state=<uninitialized>, irc=<uninitialized>, krb=<uninitialized>, modbus=<unini
tialized>, mysql=<uninitialized>, radius=<uninitialized>, rdp=<uninitialized>, sip=<uninitialized>, sip_state=<un
initialized>, snmp=<uninitialized>, smtp=<uninitialized>, smtp_state=<uninitialized>, socks=<uninitialized>, ssh=
<uninitialized>, syslog=<uninitialized>], is_orig=T, payload_size=36]
```


Let's see how to discover that **state$sig_id** is the variable for the signature identification field. It turns out that the variable named **state** is a *nested data structure* in Zeek's parlance. A nested data structure has embedded data substructures. Let's dump the variable **state** using a formatted print statement. This new print statement replaces the one used in the script in the previous slide to be executed upon a signature match.

Look at the *data structure* that was printed. Zeek denotes each data structure with a name followed by "=", which is followed by all the variables in that structure enclosed in "[" (left bracket) to begin and "]" (right bracket) to end. For instance, you find the **conn** data *substructures* contained within the **state** signature data structure. In order to identify a given variable, you must begin at the base structure (in this case, **state**) and identify any other nested structures. The **sig_id** field is denoted as **state$sig_id**; the "$" serves to dereference each of the nested data structures and any variable in the final nested data structure.

As you will see, the variables and data structure names associated with a given protocol are often a mystery, unless you look at Zeek's associated event file.

To perform the command on the slide, execute the following:

zeek -r inbetween.pcap –s windows-banner.sig sig-event-printstate.zeek

Run Our Script

```
sans@sec503: more sig-event.zeek
```

```
event signature_match(state: signature_state, msg: string, data: string)
 {
   if (state$sig_id == "cmd-exe")
   {
      print "Process cmd.exe script code";
   }
 }
```

```
sans@sec503: zeek -r inbetween.pcap –s windows-banner.sig  sig-event.zeek
Process cmd.exe script code
ls *.log
conn.log    notice.log    signatures.log
```


We discussed the practicality of creation and execution of Zeek in a separate directory when running signatures in readback mode. The same advice applies to running scripts.

We run Zeek using the **inbetween.pcap**, invoke the **signature windows-banner.sig** as we did before, but also make Zeek aware of our new script named **sig-event.zeek**. As you see, the message "Process.cmd.exe script code" results, showing that the script was triggered by the signature event match. Some of the logs that were produced are the same as those generated when running a signature with no script.

This is a demonstration of how to trigger a script from a matching signature. The code executed is not especially productive, but let's ease into the intricacies of the code after presenting some theory first.

To perform the command on the slide, execute the following:

zeek -r inbetween.pcap –s windows-banner.sig sig-event.zeek

This Looks Hard!

You might be thinking:

- I'm not a programmer!
- This looks hard!
- Is this really necessary?

Reserve judgment until after the next section.

- After seeing what we can do with this scripting in the next section, everyone decides that even if this looks hard, it's absolutely worthwhile!

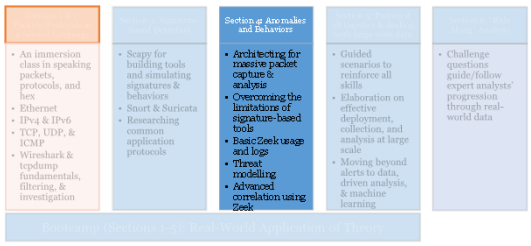
We're about to complete a very easy lab exercise. Before we do that, it seems important to acknowledge what you might be thinking at this point, "This looks really hard!" You might also be thinking, "I'm not a programmer!" That's completely OK.

Just as we asked you to trust the process at the beginning of the course (and we assume that looking back, you can see the value of spending so much time in understanding packets and protocols so deeply), we ask that you reserve judgment on Zeek scripting until after you see an example of the true power of this capability in the next section.

Zeek Scripting Exercise

Workbook exercises "Zeek Scripting 1"

This page intentionally left blank.



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

Introduction to Zeek

Zeek and Signatures

Zeek Scripting Basics

Threat Analysis

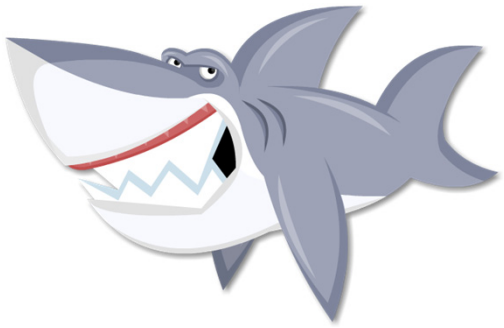
Scripting Correlations

Scripting Behavioral Anomalies

Spicy

Let's Generalize – Threat Modeling

- The Problem:
  - We're getting phished.
  - Phishers are always getting better at avoiding philters.
- What to Do:
  - Prevention is great… when you can.
  - You can't react until you detect.
  - How can we detect in near real time?



If you recall, we came to the realization while studying signature-based approaches that looking for the exploit is often less practical in the long term than looking for the effects of the exploit. Even then, how many different bad payloads could we find that indicate that a compromise has occurred? In fact, we would argue that this is *precisely what's wrong with signature-based detection*. We wait for a bad thing to happen and then write a specific rule to find that particular attack. We try to make it general enough that it isn't easily bypassed, but these signatures tend to be very narrow. We think this is a losing approach.

A far better, though more challenging, approach is to stand back and talk about what the *behavior* looks like. This is different. We're not trying to identify what a *specific* attack looks like. Instead, we consider a number of specific attacks and attempt to generalize those to techniques and behaviors.

This mindset is challenging with a signature-based IDS like Snort or Suricata. When using a network intelligence framework like Zeek, though, this works wonderfully! It's also extremely practical because it fits perfectly with what Defense-in-Depth actually means…. More on that in a moment.

(Image credit: https://www.deviantart.com/legendaryfrog/art/Cartoon-Shark-407934108)

Signatures?

- Signature detection is OK …
  - …but evading signatures is what attackers do…
  - …and it assumes that you *will* detect…
  - …and you can't respond until you *do* detect.

To be clear, we aren't saying that you should go back to your office and tear down your Snort or FirePOWER infrastructure. Those tools are wonderful, and you should keep using them! What we are suggesting is that you need to supplement your signature-based tools, which know how to look for *known* attacks, with a tool that you can use to describe potentially malicious *behavior*. What's great about this is that you can do threat modeling and write detection scripts long before any attacker has thought up a way to use that technique in an attack! This allows us to transition from being purely reactionary to at least being proactive.

With the case that we are about to explore, we're not saying that you *won't* get phished; nothing that we are about to do tries to stop the phishing email. Instead, you will at least *know* that you got phished, and you have the potential to know it and be able to react to it *far sooner* than you otherwise would!

Signature Alternative?

- Defense-in-Depth isn't just layers of defenses.
  - It's operating under the assumption that your systems *will* be compromised.
  - This has led to microsegmentation, nanosegmentation, and zero-trust networking.
- How would we act if we assumed we would be compromised?
  - We'd spend more time figuring out what "compromised" looks and acts like...

Be more “cat"...

Be more curious!

Granted, what we are proposing to you in this section is likely a new way of thinking about things. Still, we said that this fits with Defense-in-Depth, which is the security approach that most organizations claim to follow. What's interesting is that, while most organizations do *claim* this, many don't act like they *believe* it.

If you follow Defense-in-Depth, you are admitting that you *will* be compromised. You don't know when. You don't know which systems. You don't know by whom. This leads to creating layers of defenses. Unfortunately, experience has shown that, failing to actually *believe* that we will be compromised leads to our engineers and administrators buying and installing a great many *protective* or *preventive* controls (things like firewalls and configuring ACLs), paying little to no attention to the need for *detective* and *reactive* controls. What does this mean for our enterprise? It means that we are often assuming that our protective systems will prevent bad things from happening… but when they fail to do so, we have little or no capability to detect and begin a reaction to that compromise.

Signature-based detection is a symptom of this. Signature-based detection grew out of the existing antivirus market in the mid-1990s. At the time, as the network monitoring industry got started, a few experts were analyzing packets and watching for anomalous behaviors. Vendors, sensing a market opportunity, wanted to create tools, but it was (and still is) very difficult, especially in the 1990s, to automate detection of anomalous behavior. What could be done? We could simply take the existing model in antivirus tools and apply it to network activity! This creates a product quickly that requires ongoing support (signature updates) and can be run by someone with little or no training! Unfortunately, this really isn't enough today for a security-conscious organization.

Threat Modeling: Mary, the Mail Server

Let's do some threat modeling, then. To do so, let's introduce our cast of characters.

First, we have Mary the Mail Server. Mary is sitting behind our firewall on our protected network, accepting and relaying mail for our users.

Threat Modeling: Evil Eddy – Gone Phishin'!

On the left-hand side, we have Evil Eddy. Evil Eddy, the bane of the internet, has decided to go phishing within our network.

It has been pointed out that Evil Eddy bears a striking resemblance to another SANS Instructor, Mark Baggett. We assure you this is purely coincidental. Mark is a really nice guy. ☺

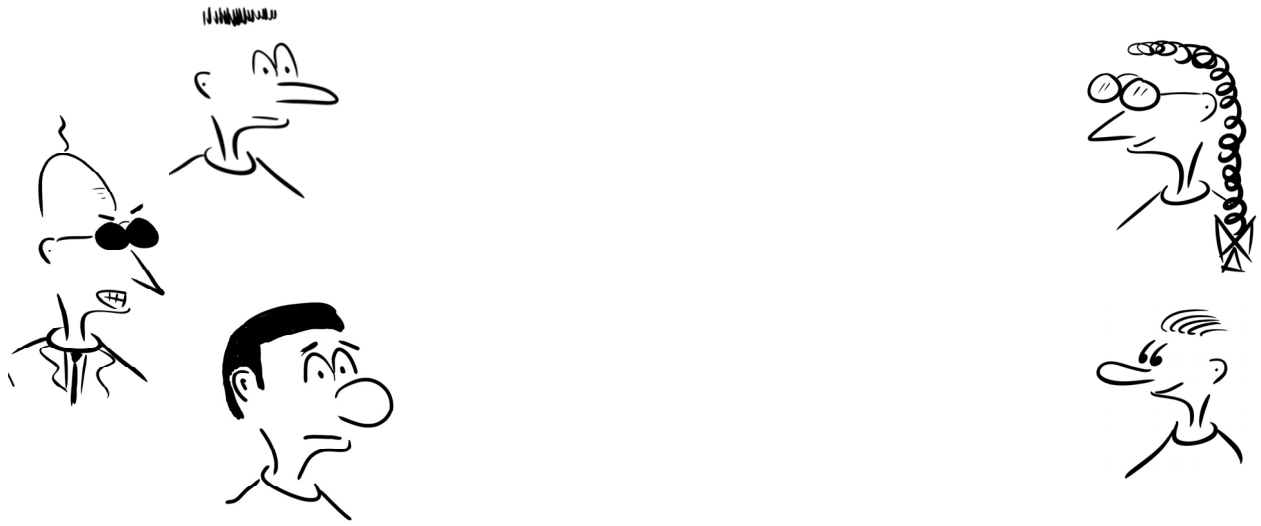
Threat Modeling: Sam the Salesperson

More specifically, Eddy has his sights set on Sam. Sam is a user in our Sales Department. He's one of those users who, no matter how much user awareness training you send him to, he can't seem to stop himself from clicking on links in emails.

Threat Modeling: Wally, a Co-opted Web Server

When planning his phishing campaign, Eddy decided he would like to blame some other people for the attack to avoid being implicated himself. To this end, he has co-opted Wally, an otherwise innocent web server out on the internet. Eddy will host his malware payload here.

Threat Modeling: Doug, a DNS Server

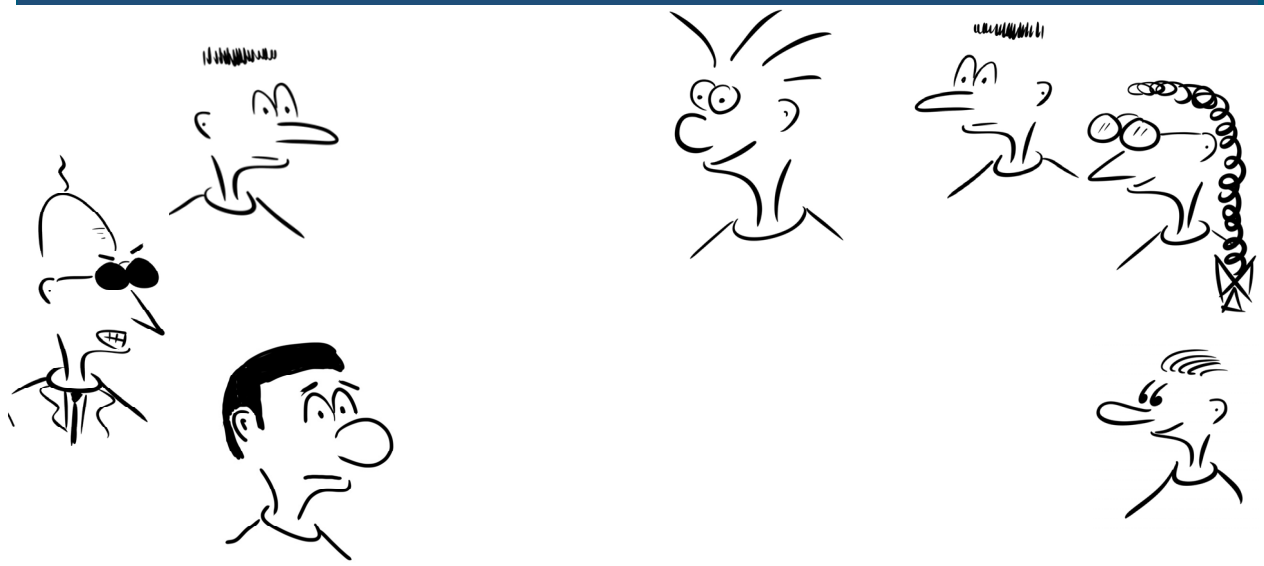


Eddy also needs a way for Sam to know where to go. While he could put a bare IP address into the email, that would look a little bit suspicious. Instead, he's going to use a domain name that he either owns or "Ownz." Doug will serve this DNS zone.

Threat Modeling: Dan, Our DNS Server

Of course, for Sam to get anywhere, he's going to need to do DNS lookups. Whenever he needs to do so, he's going to talk to Dan, our DNS server.

Threat Modeling: Bob, Our Fearless Analyst

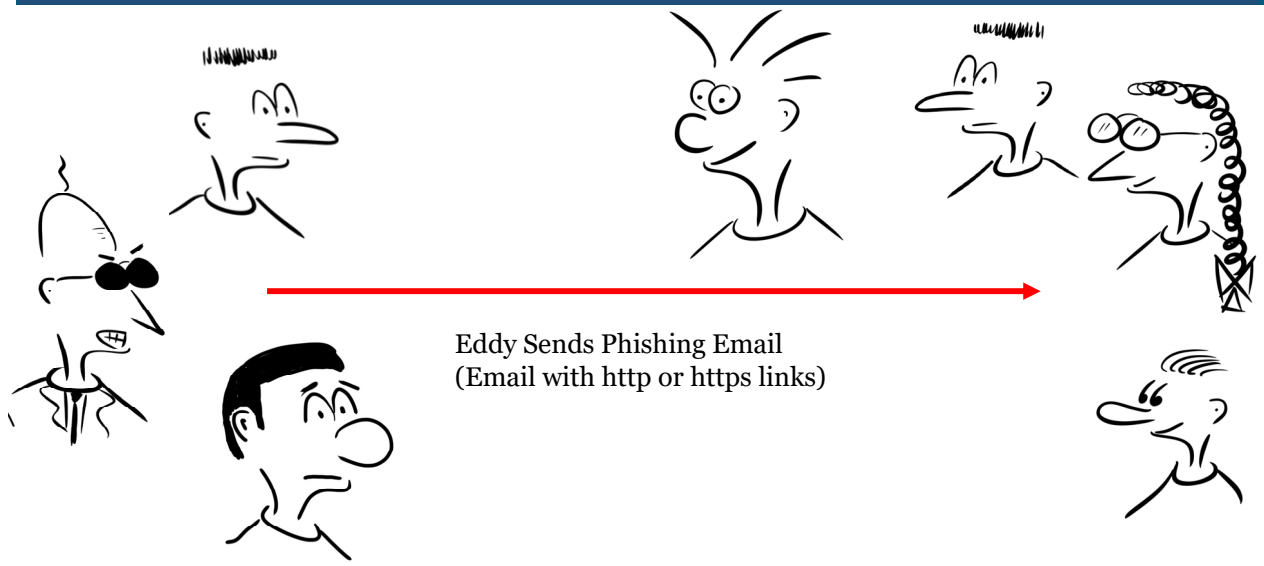


Finally, we have Bob, our fearless analyst who monitors our network night and day.

In this particular scenario, we are going to state that Bob's view of the network traffic is limited. Notice that he is facing away from Sam, Mary, and Dan. For the purposes of our analysis here, we will say that he has *no vision* of any internal communication between these hosts but can only see traffic entering and existing the network.

We have chosen this constraint because, especially on very large and dispersed networks, this may most closely simulate what you will have in place already.

Threat Modeling: Casting the Line

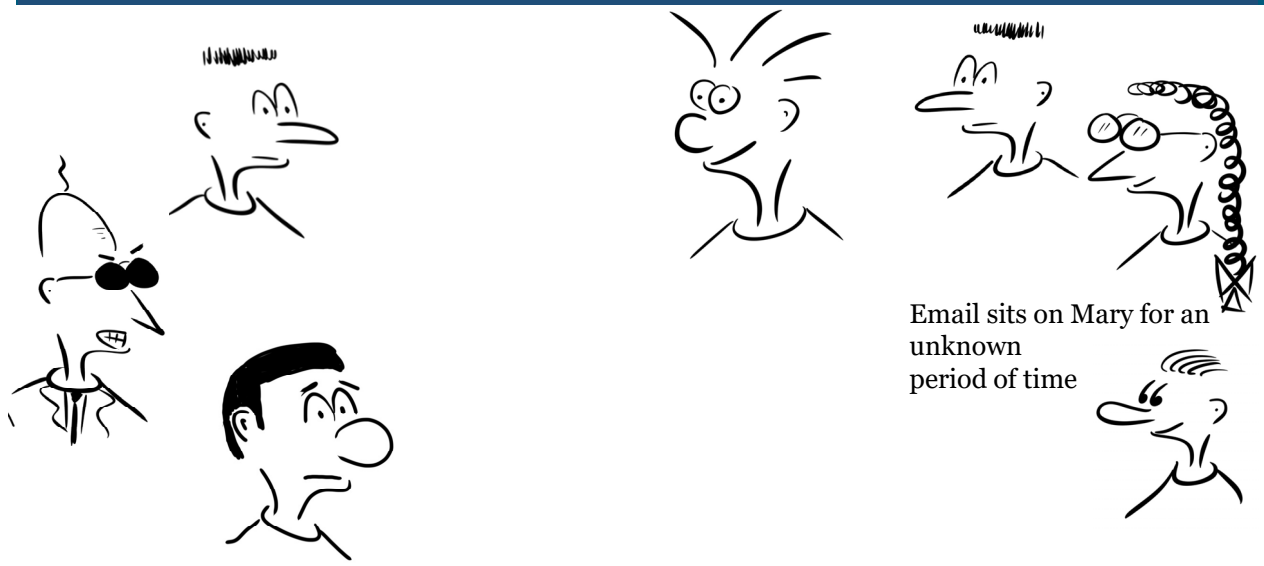


With our cast of characters laid out, we're ready to start. Eddy begins by crafting an email to Sam (and likely many other internal users), which he sends to Mary. During our analysis process, we spend time thinking about what phishing emails tend to look like. Certainly, they can have attachments in them; these could be zip files, executable files, PDF files, etc. More commonly, however, they will have one or more links to a site. That site could be a fake copy of a real website, seeking to steal credentials. It could also be hosting JavaScript-based malware, some other drive-by, or possibly a malicious download.

With this script, we aren't going to try to find *every* kind of phishing. For this exercise, we will focus only on phishing attacks that include links, which are the most common type.

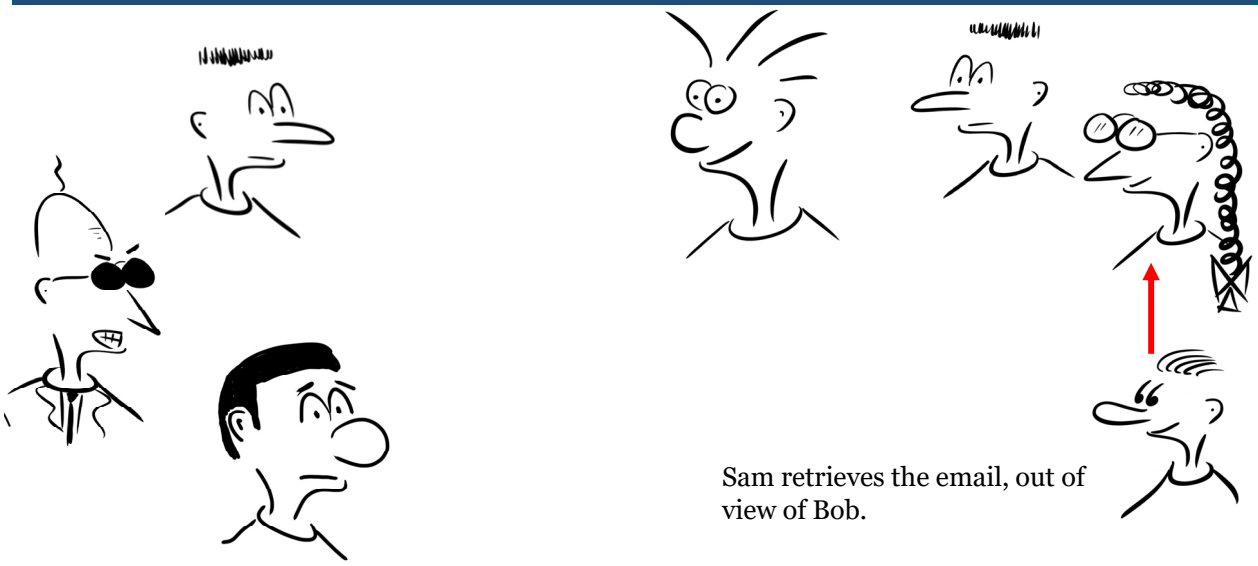
Please note that Bob has the opportunity to see this email pass by.

Threat Analysis: Phishing (1)



The email that Eddy has sent will now sit on Mary for an unknown period of time. Sam could choose to read his email in a few seconds, minutes, hours, days, or weeks. We have no real way of knowing. Whatever the case, time passes….

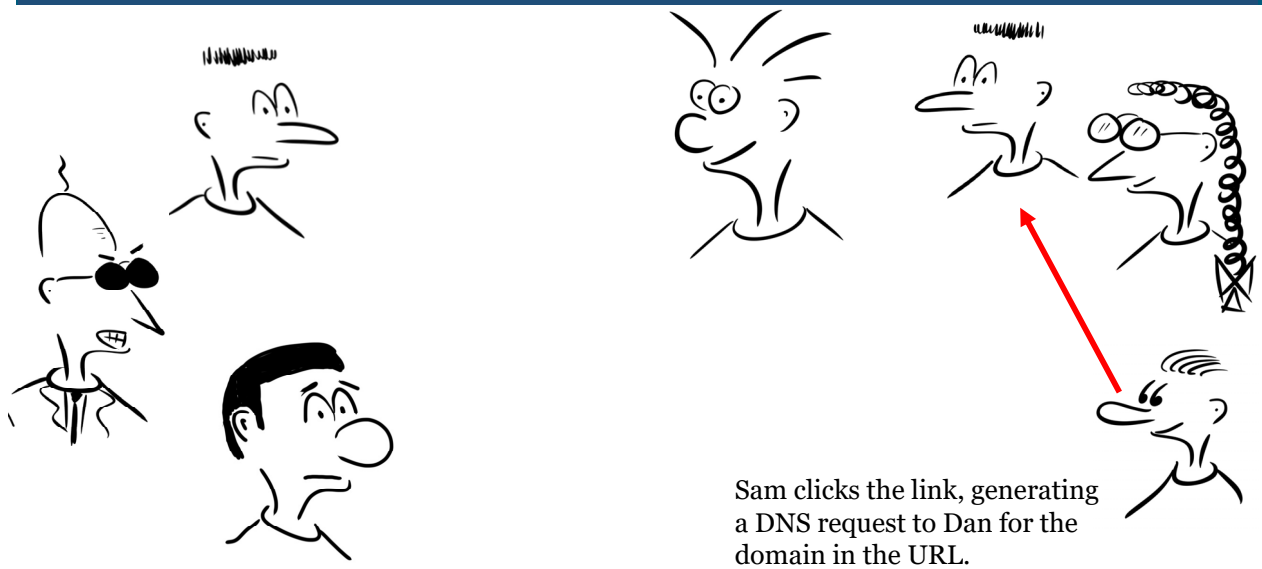
Threat Analysis: Phishing (2)



Eventually, Sam retrieves his email. Just because he has retrieved it, this doesn't mean he's actually looked at it yet. That email might sit in his Outlook client for a week before he actually deals with it. Regardless, this activity occurs out of Bob's view; he has no way to observe this activity.

Time passes….

Threat Analysis: Phishing (3)

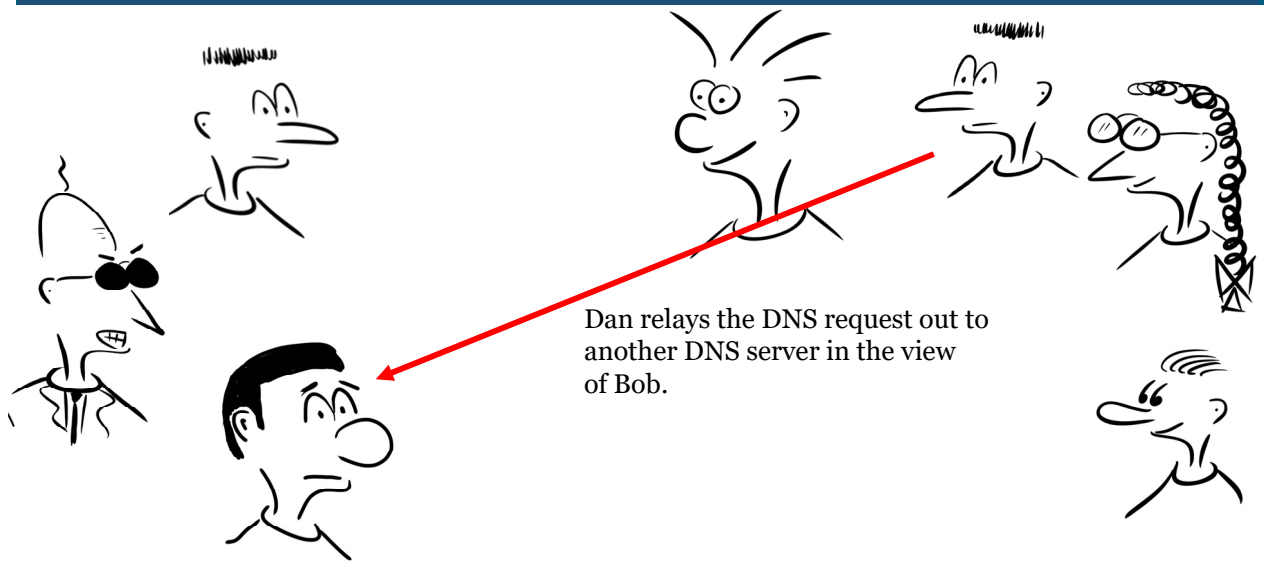


Sam clicks the link, generating a DNS request to Dan for the domain in the URL.

A few days later, perhaps, Sam opens up the phishing email. For whatever reason, he unwisely chooses to click on the link in the email. This will *not* trigger a connection to Wally… not yet. Sam doesn't know where Wally is. Instead, this will trigger a DNS request to Dan, trying to find Wally's IP address.

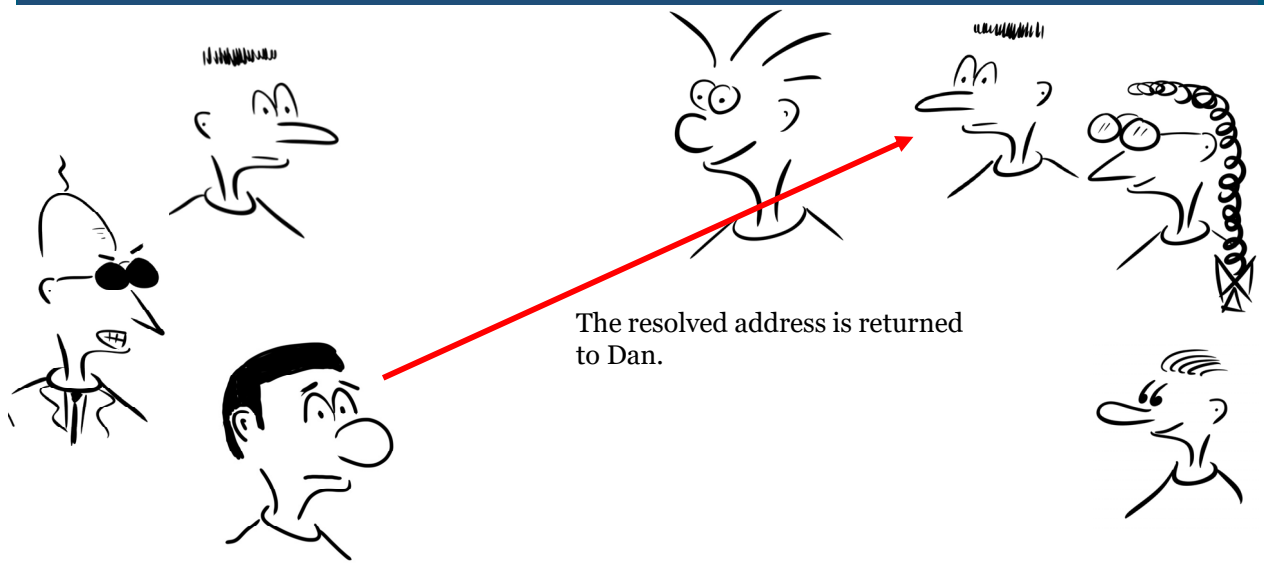
This request occurs out of the view of Bob.

Threat Analysis: Phishing (4)



Dan, having received a DNS lookup from Sam, will relay that request out to the authoritative server for that domain. In this case, that's Doug. Bob can absolutely see this request, but as far as he knows, it's Dan who is asking. Even though this is interesting, since it's related to the email, he doesn't know who the original requestor (Sam) is.

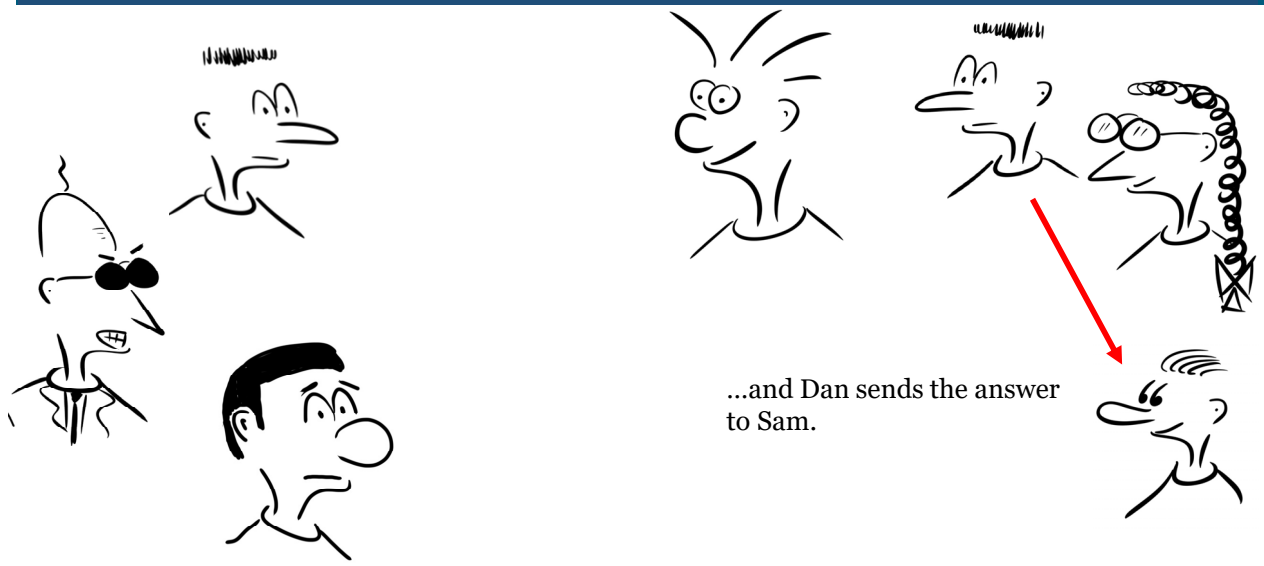
Threat Analysis: Phishing (5)



Eventually, likely after just a second or less, Doug will send back an answer. This DNS response will contain an A or AAAA record along with the original query information, Wally's name.

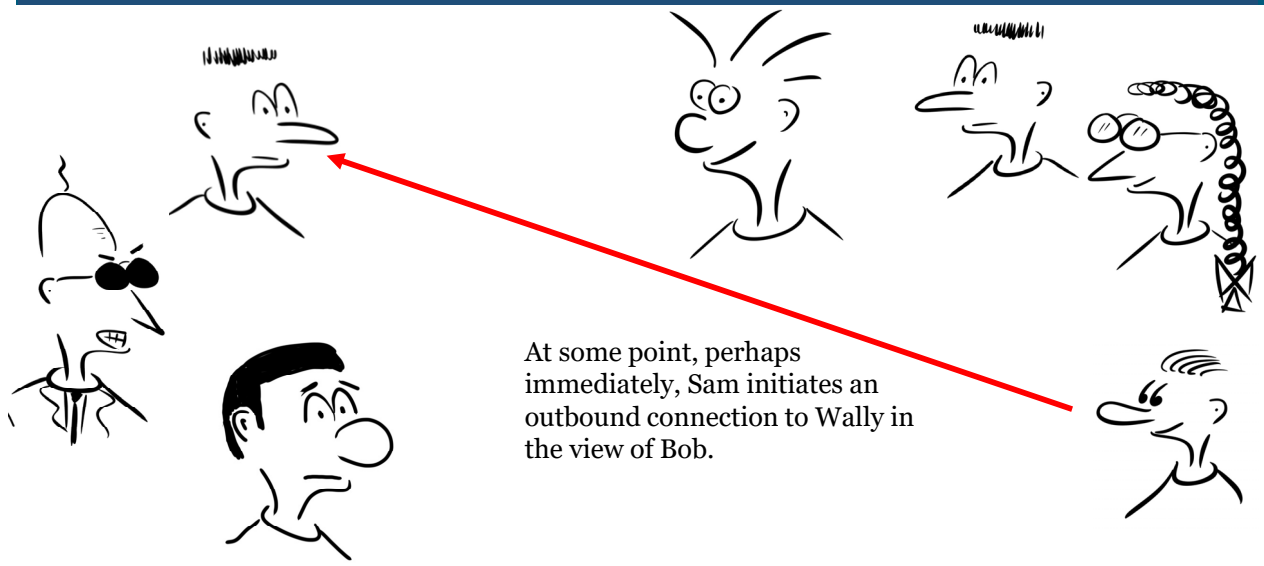
Bob, of course, can see this response coming back.

Threat Analysis: Phishing (6)



Dan immediately sends this response to Sam. This response is, again, out of Bob's field of view. Sam does have everything he needs to connect to Wally, though!

Threat Analysis: Phishing (7)



Either immediately or at some point in the future, Sam now attempts to establish a connection outbound to Wally. If you've been following our story closely, your shoulders likely just got tense because this is where the bad stuff will happen.

Bob can see this outbound connection to Wally.

How Does This Help?

- We know we're looking for this *behavior*:
  - Email containing one or more URLs
  - Future DNS request looking for the domain name from the URL(s) that was (were) in the email
  - Future outbound connection attempt to the address that the domain from the URL resolved to…
  - …and there can be more than one.
    - Multiple internal users will likely *not* generate multiple outbound DNS requests.

Now that we've modeled the behavior, let's take a moment to take stock. If we wanted to create a detection capability, we would need to find emails that have URLs in them. If we find URLs, we should probably take note of the fully qualified domain name in the URL.

If we kept track of those URLs, we could begin monitoring our DNS resolution activities. While we may have many thousands of resolutions, we now have a fairly short list of names that have shown up in URLs in emails. If we see one of these names pass by, it would make sense to take note of the IP address or addresses associated with it in the DNS response.

Armed with this information, we now have a list of IP addresses that are potentially involved in dering malicious payloads or in serving social engineering attacks. Better yet, we now have a way to identify the individual user rather than the nameserver! Additionally, while only one DNS request will be seen by Bob, if other users click that same link, we will also be able to find *those* users!

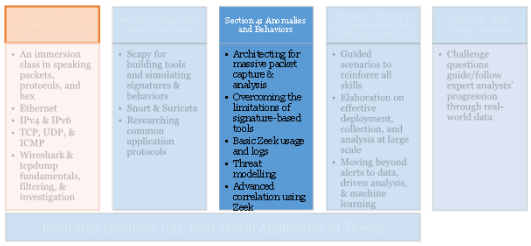
This all sounds good, but what can we do with it?

Yes, But How Does This Help?

- Signature detection can't do this.
  - Signature detection can look for the sender or the subject, not the behavior.
- Behavioral detection *can* do this.
  - Zeek can do it!
  - Let's see how...

First, make sure you clearly understand that signature-based detection absolutely cannot find this activity. Certainly, I could write a signature to alert whenever a URL is seen in an email, but that's as far as it goes. There is absolutely no way to feed that information to subsequent signatures for DNS lookups and no way to feed *that* information forward to outbound connection attempts.

Using Zeek, however, we absolutely can do it. Let's see how.



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

Introduction to Zeek

Zeek and Signatures

Zeek Scripting Basics

Threat Analysis

Scripting Correlations

Scripting Behavioral Anomalies

Spicy

Pseudocode

Nonprogrammers can find solving this problem to be overwhelming.

- Break the problem down into manageable pieces.
- One of the key ways we do that is through pseudocode:

Step 1: Locate URLs seen in emails.
Step 2: Extract the fully qualified domain names (FQDNs) from the URLs.
Step 3: Remember the FQDNs.
Step 4: Look for DNS responses for queries referring to FQDNs from step 3.
Step 5: Look for outbound connections to the addresses found in step 4.
Step 6: Report any outbound connections.

Before we embark on writing a script to solve this problem, let's start with a good foundation. Students with a programming background may automatically imagine all the steps required to find our desired connections. If you have no programming experience, you might just feel overwhelmed. There is a very simple way to overcome this.

The proper thing to do is to begin by describing all the steps required to solve the problem. We've done this in the slide by providing *pseudocode* for a solution. Pseudocode is just a plain language description of the steps that are required to solve the problem. This allows us to take what can feel like a big, overwhelming problem and break it into very simple and solvable steps. In a very real sense, you can think about this as you might a very difficult-looking equation in mathematics. Solving the entire problem can seem quite daunting, but if you solve very small pieces of it and continue sequentially, you can often break the difficult problem down into many simple problems.

Scripting Correlation: Selecting a Zeek Event

The defined Zeek event **mime_entity_data** gives access and allows manipulation of SMTP content

Variables passed are:

c a **connection** type for connection data like IP addresses

length for a **count** for the number of bytes of **data**

data a **string** representation of the MIME content

```
event mime_entity_data (c: connection, length: count, data: string)
{

}
```


We need to build a correlation script that will subscribe to three different events. The first event will allow us to examine email content. The second will find DNS address responses. The third looks for outbound connections.

For our first event, the **mime_entity_data** is perfect. Doing just a few seconds of research allows us to identify that any time a MIME entity, which would include SMTP and POP emails, passes by a Zeek worker, the **mime_entity_data** event will be generated. This makes available the actual content of the email message. This will make the passed-in variables accessible to the code we will write. The format is that all variable names, followed by a colon and followed by its Zeek-defined type, are listed between the parentheses that follow the event name. Each variable and type pair is separated by a comma from any others.

We have already discussed the use of the variable **c**, which is a Zeek-defined type of **connection**. A **connection** type represents a Zeek data structure containing variables and values that define different aspects of the connection, such as ports and IP addresses. The variable **length** is the length of **data** that is passed as a variable too. The Zeek-defined type **count** is a positive number. Finally, **data** in this reference is defined as "The raw data of the complete entity," meaning all the email content passed to the event. It has a type of **string**.

Scripting Correlation: Find URLs in Emails

```
global domains_in_emails: set[string];
global addresses_from_links: set[addr];

event mime_entity_data (c: connection, length: count, data: string)
{
  local urls = find_all(data, /https*:\/\/[^\/]*/);
  if(|urls| == 0){ return; }
  for(url in urls){
       add domains_in_emails[split_string(url, /\//)[2]];
  }
}
```


Let's put together the pieces logically. Ignore the two lines that begin with the word *global* for now. Instead, take note of the **mime_entity_data** event.

Within the event that we have written, we are finding all the instances of text strings that begin **http://** or **https://** and we stop collecting text when we reach the next forward slash. In other words, if the URL were **http://www.f4ceb00k.com/malware.php**, our **find_all** call is extracting *only* **http://www.f4ceb00k.com**.

The next line uses the absolute value markers to determine the size of the *urls* set. This is the set of URLs that were found using **find_all**. If there aren't any, the function exits. If there are….

Look at the line that begins "**for(…**". Can you take a guess what that is doing? Don't think, "I'm not a programmer!" Just try to *read* it. Does it make sense to you that it is going to loop through the list of URLs we have extracted? In fact, for each one of those URLs, it's going to execute the following line. That line is, admittedly, the most complex line in the script. This adds the URL to a variable, defined at the top to be of **global** scope (which is literally global, meaning any script running in Zeek can access its content). This variable contains the set of URLs that were found.

The complex part of it is the **split_string** stuff. Don't stress about it; this complex-looking stuff is simply pulling the fully qualified name out, stripping away the **http://**.

One last point of interest: We have defined **domains_in_emails** to be a set of strings. **Set** here has the mathematical meaning, that is, a list of unique elements. In other words, if you try to add the same name to the list multiple times, the list will never have more than one copy. This is a very wise thing to do to limit memory consumption.

Scripting Correlation: Global Variables

Global variables permit *any* script/code in Zeek access to them

```
global domains_in_emails: set[string];
```

A set comprised of string elements to store extracted URL names from phishing links

```
global addresses_from_links: set[addr];
```

A set comprised of IP addresses to store resolved URL link names

First, some **global** variables are defined. As previously mentioned, this permits all three user-defined events that we will create to have access to them.

We have defined the first **domains_in_emails** to be a **set** of strings. **Set** here has the mathematical meaning, that is, a list of unique elements. In other words, if you try to add the same name to the list multiple times, the list will never have more than one copy. This is a very wise thing to do to limit memory consumption. This variable is used to store string elements that are the extracted URL names found in the data, perhaps from phishing links.

The second global variable is **addresses_from_links**. It is a set comprised of IP addresses of resolved extracted URL links. We will not use this particular variable in our **mime_entity_data** event code. But remember that our intention is to perform our correlation using three Zeek events that appear in a single file. Therefore, we need to define all variables that are global at the top of the physical file that contains the code for all three events.

Scripting Correlation: Find DNS Resolutions for URLs

```
event dns_A_reply (c: connection, msg: dns_msg, ans: dns_answer, a: addr)
{
  if(ans$query in domains_in_emails){
    add addresses_from_links[a];
  }
}
```


Whew! That last function was really complex. The great news: That is absolutely the most difficult part of this script.

This event subscribes to **dns_A_reply**, which is generated whenever an address record is returned from a nameserver. Within that A record, the **ans** variable will contain an element called **query**, which is the name that was looked up. The **a** variable will contain the address that was returned that maps to that hostname.

Can you figure out what the **if(…** section is doing? Have a look at it before reading the rest of this page.

Hopefully, you can read and interpret what it says. "See if the queried name is in the set of domains_in_emails. If it is, add the address to the set addresses_from_links."

Scripting Correlation: Find Outbound Connections

```
event connection_SYN_packet (c: connection, pkt: SYN_packet)
{
  if(!(c$id$resp_h in addresses_from_links)) { return; }
  if(c$id$resp_p == 80/tcp) {
    print fmt ("Phishing related: HTTP connection from %s to %s", c$id$orig_h, c$id$resp_h);
    return;
  }
  if(c$id$resp_p == 443/tcp) {
    print fmt ("Phishing related: TLS/SSL connection from %s to %s", c$id$orig_h, c$id$resp_h);
    return;
  }
  print fmt (">>> Phishing related: connection to port %d from %s to %s", c$id$resp_p, c$id$orig_h, c$id$resp_h);

}
```


This is it! The home stretch! We're now looking for outbound SYN packets (though we could certainly look for **new_connection** events or any other event that feels relevant). This event, when triggered, will check to see if the destination address is in the set of addresses that were in DNS lookups… and those DNS lookups are for hostnames that showed up in links that were in emails!

What does this mean??? This means that *we just correlated all the way from a URL in an email to an outbound connection from Sam to Wally!* The best part is that we never wrote a signature. This script will work for *any* phishing email that implements this behavior.

Results

- We now have near real-time detection!
  - Now we can react…
  - How long is it taking you to begin incident handling on successful phishing today?

```
sans@sec503:/sec503$ analyze external | zeek -r - /sec503/ShowMeThePackets/Zeek/phishing.zeek

Phishing related: HTTP connection from 170.129.1.50 to 89.46.105.77
Phishing related: HTTP connection from 170.129.1.50 to 89.46.105.77
…
```

Script: https://github.com/dhoelzer/ShowMeThePackets/blob/master/Zeek/phishing.zeek

Here we just have an example of our script in action. This script, in its entirety, is available to you in the ShowMeThePackets GitHub repository.

There are still a few gaps, however.

Gaps/Improvements

- Won't that generate false positives?
  - Sure it will... but humans are creatures of habit.
    - Add a global set that contains permitted domains.
    - Add to it over two weeks or so.
    - After that, you may have a handful of new URLs each week.
  - Do you have a policy that says, "Thou shalt not click on links?" If so, Job Done!
- What about memory over time?
  - First, you can store these in a cluster-wide broker data store.
  - Second, you can mark these sets with something like this:

```
global domains_in_emails: set[string] &write_expire=2weeks;
```


Let's consider the gaps. The first is, "Won't this generate false positives?" Possibly. That depends on your policy. If clicking on links is forbidden, the script might be fine as is. On the other hand, if users are permitted to follow *some* links, this is a matter of some additional coding and some patience. If we were to add a global set that is used to track permitted hosts, we can use that to add in permitted sites. Since people's behavior tends to be very habitual, you will find that after about two weeks, you will only need to deal with a handful of new URLs every week, which is entirely manageable.

The other gap is this: You might be wondering, "If this remembers things forever, what happens if I need to restart Zeek? Won't I lose that data? Or what if Zeek runs for months! Will I run out of memory?"

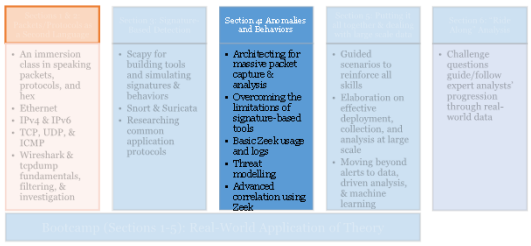
Great questions. Again, just a few small tweaks are required. In previous versions of Zeek, adding the **&persistent** attribute to the sets instructed Zeek to save them when exiting and reload them when restarting. Unfortunately, this attribute was released just prior to the 3.0 release in favor of Broker data stores. Interacting with data stores is a bit more complex, but doing so allows us to track this data over a very long period of time.

For the second problem, the concern over running out of memory, there are other attributes that can be used to limit how long Zeek remembers things. Adding **&write_expire** with a time frame instructs Zeek that, if the entry in the set has not been written to *or read* within that time frame, it should be expired out of the set. Easy!

Zeek Scripting Correlations

Workbook exercises ”Zeek Scripting 2"

This page intentionally left blank.



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

Introduction to Zeek

Zeek and Signatures

Zeek Scripting Basics

Threat Analysis

Scripting Correlations

Scripting Behavioral Anomalies

Spicy

Using Zeek Properly

We believe most organizations are not using Zeek to its full potential.

- Most just use it for rich log data.
- They try to perform all correlations at the SIEM/SEM.
- We believe it's better to correlate as close to the data as you can!

The possibilities are endless.

- We've done two basic correlations across time and sessions.
- Behavioral anomalies can be trivial to automate.
- Longer-term, larger-scale statistical analysis isn't any more difficult.
- We can also use scripts to feed other processes—like machine learning (tomorrow).

It is our hope that after seeing the example of detecting indications of phishing attacks and creating the DNS anomalies script in the lab exercise, you have come to the conclusion that Zeek scripting seems to be a worthwhile time investment. We also hope that you are coming to realize why we take the position that most organizations aren't using Zeek to its full potential.

Most organizations that use Zeek, or the commercial offering Corelogic, simply install it and point its logs at a SIEM. Many will add in things such as the **Threat Intelligence Framework**, a set of free scripts that can import threat intelligence feeds and begin to create additional logs to alert you to activity involving hosts and networks with poor reputations. This is fine as far as it goes, but Zeek is capable of far more.

In the slides and then in the lab, we have looked at or created two simple correlation scripts. The two examples are especially interesting because they illustrate correlations that would be *incredibly difficult* to perform with a SIEM! In this section, we will discuss together several useful scripts and analyses that we can script Zeek to perform. Some of these are most useful when engaged in incident response, while others are useful for long-term continuous monitoring and alerting.

As we discuss these, continue to think about other ways that Zeek could be used to feed other processes within your organization. For example, we will briefly discuss how we could feed data from Zeek into Machine learning processes in Book 5. The possibilities are endless.

Simple Behaviors

Think about the transactional nature of most protocols.

- Based on this and our knowledge of how systems behave, we can design simple yet useful indicators for our network.

Let's consider a few:

- Unusual HTTP activities
- Unusual HTTPS/TLS activities
- Unusual TCP behavior from hosts
- Long connections
- Large amounts of data
- Unusual numbers of connections

With the extensive knowledge that you now have about how network and application protocols behave, you are in an excellent position to think about and describe expected behaviors. You are well aware of how HTTP, DNS, and other types of applications, in addition to the underlying protocols, will behave during normal communications. You can even imagine how other applications must behave. For example, while TLS connections for HTTPS will usually be transactional, a VPN running over port 443 will not be and is very easy to identify.

Based on this knowledge, then, we will outline six different behaviors over the next several slides. After doing this, we will spend the remainder of this section working through lab exercises that create working scripts to identify each of these behaviors.

HTTP Script

HTTP: small request, larger response

- Wouldn't this apply to HTTPS as well?
- This rule of thumb is broken by uploads, video streaming, HTTPS-based VPNs, etc.

Script Overview:

- Test to see if the amount of data sent by internal hosts is greater than the amount of data sent by external hosts.

Our first example is very simple. We are applying the idea to HTTP and HTTPS, but you can extend this to other protocols. For example, we would expect a DNS query to typically be smaller than the answers that are returned. If we found that the queries were regularly larger than the answers, this would cause some concern and be worthy of a notification!

If we apply the rule of thumb that HTTP and HTTPS requests will generally be smaller than the responses, we can create a simple behavioral alert. Our script would simply generate alerts any time it detects HTTP or HTTPS connections where more data is sent outbound than is returned. This, by definition, is data exfiltration, though it isn't automatically bad. Still, it is worthy of a log notice!

TLS Script

HTTPS/TLS: Servers will generally serve one certificate to you.

- Hosting multiple sites breaks this rule.
  - How likely is it a user is hitting multiple sites on such a host?
- TOR and other ”privacy” applications break this rule
- A successful machine-in-the-middle attack?

Script Overview:

- Create a variable to track certificates from individual hosts
- Report when a host is seen serving more than X number of certificates.

Next, consider how HTTPS or TLS servers behave. We can expect that when a client system connects to a server over port 443 (or TLS in general), the server will present a single server certificate. What if, over a short time span (one hour or one day perhaps?), that server presents more than one certificate to clients in our network?

If the server is hosting multiple TLS sites, this could be completely appropriate. How likely is it, however, that our users just happen to be connecting to multiple sites hosted on the same server? Are there other possible explanations for this?

One possibility is that we are looking at a TOR connection, which is well known for using many different certificates over a short period of time. Another possibility is a successful machine-in-the-middle attack. In this case, an attacker could be attempting to act as a transparent TLS proxy to the real server!

To detect this, we could create a script that somehow keeps track of which certificates are seen associated with which servers. The server could then alert us if more than a specified number of certificates are seen from any one server.

TCP Script

TCP: Hosts will typically use a standard set of options/settings.

- These can change based on negotiation, but we can expect the SYN to be mostly standard.
- Large changes could indicate a new host at that location or some other significant change.

Script Overview:

- Capture protocol fields:
  - Options? Window size? TTL perhaps?
  - Remember them for each host.
  - Report if/when they change—possibly update after a change.

A more challenging possibility is to examine behavioral aspects of TCP connections originating from our systems. We know that operating systems will tend to create TCP connections in consistent ways. We would expect that an endpoint would always generate a SYN in a particular way, with a static window size, particular TCP options, and option values (window scale, for example).

What if we found these values changing on an endpoint? This could indicate many things. Two of the most obvious are that there is now a different client at the IP address or that the client is running some unusual piece of software that has reconfigured these defaults… perhaps a piece of scanning software!

How can we find these? We could write a script that tracks the various attributes of TCP SYN packets for individual clients that reside within our network. One we have "learned" what these are, our script could begin to create notices whenever they are seen to change.

Long Connections

Zeek will only log to the **conn.log** when a connection ends.

- A long-term connection could be an indication of a tunnel or C2.
- I don't want to wait until it ends to alert!

Script Overview:

- Identify connections that have been active for more than a specified time.
- Periodically alert when connections exceed this threshold.
  - Report incrementally as the connection persists.

One of the practical issues when using Zeek is that it will only generate a connection log entry when a connection has ended. What if there were a DNS "connection" that went on for three hours? Clearly that would not be normal. Would you be happy to wait until that connection ended to discover that it occurred, or would you want to have some notice that the connection had been up for more than some threshold amount of time?

Looking for long-term connections is a very useful continuous monitoring activity and a script that we would recommend you immediately begin to run in production. How? One of the scripts that you will shortly create will periodically check for how long a connection has been up and incrementally generate notices as the connection crosses certain thresholds.

Large Data Flows

Zeek will only log to the **conn.log** when a connection ends.

- An exfiltration of a great deal of data is very concerning.
  - Exfiltration does not automatically mean "bad."
- I don't want to wait until it ends to alert!

Script Overview:

- Identify connections that send more than *X* bytes outbound.
- Periodically alert when connections exceed this threshold.
  - Report incrementally as the data flow continues.

A related activity that we might want to detect is large data transfers. These connections might not be very long, but if there is a great deal of data transferring outbound, we might want to know sooner than the end of the connection! We might even want to set a threshold, at which point the connection will be automatically torn down.

We won't go to the point of tearing the connection down in our script, but we will write a script that alerts every time a connection crosses a predefined threshold number of bytes and then continues to alert as the connection sends more and more data. This gives us a wonderful way to detect potential exfiltrations that haven't generated any other alerts in our signature-based or DLP systems.

Numbers of Connections

User workstations will tend to have an average behavior—servers more so.

- A workstation establishing more than some threshold above the average?
- Could indicate various activities:
  - Torrent traffic?
  - Scanning?
  - C2 network?

Script Overview:

- Count the number of hosts any host talks to.
- Work out the average and report hosts that significantly exceed this average.

The final script that we will look at, and the most challenging, will attempt to examine the normal behavior of our systems. Generally, all workstations in your environment will have an average number of external hosts that they interact with. Similarly, servers will have the same type of average, though we would expect it to be far lower unless we are looking at a DNS, mail, web, or a similar external-facing server.

Would it be interesting to find hosts that are speaking with more than the average number of hosts? Could this be indicative of activities such as BitTorrent and other peer-to-peer file sharing networks? Could it be a host that is engaged in outbound scanning? Could it be a compromised host that is connecting to a fast-flux C2 channel? Could it be a sign of other HR-related issues?

We will create a script that keeps track of how many individual hosts any internal system talks to. The script will also work out the averages and report either all hosts or the hosts that are significantly above the average.

A script like this can be useful running in real time, but our script in the lab will focus on IR activities.

This Next Lab

This next lab is very extensive.

- Take it at your own pace.
- We will be creating scripts to solve some of the problems just outlined.

But I'm not a programmer!

- Multiple approaches:
  - "I get this!" Look at the problem description and go write the script!
  - "I sort of get this" Use hints as needed to reveal portions of the script.
  - "I'm a bit lost" Use the "Phone a Friend" hint right away.
- Phone a Friend is a mostly completed script that you can copy and paste.
  - You need only make a few adjustments to have a working solution!

Armed with these ideas for useful scripts, we are ready to work on the final lab for the main body of our coursebook. In the lab, you will have the opportunity to work through the creation of several scripts and to experiment with more advanced scripts.

Since we appreciate that everyone taking the class is at a different level of experience, especially when it comes to scripting, the labs are written in a very detailed way. There are, effectively, three approaches to each.

If you have programming/scripting experience, we would encourage you to read the description in each section and challenge yourself to use the Zeek documentation to create a script that accomplishes each of the tasks. If you get stuck along the way, use the hints as needed.

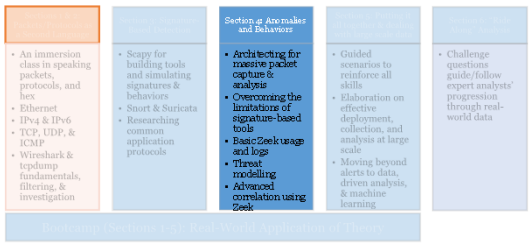
If you understand what is happening but have only a small amount of scripting experience, you may wish to make much more extensive use of the hints. For example, the first hint in each lab will give you an overall framework for the events that are required to solve that lab, with each successive hint adding more code to that script.

If you have no scripting experience at all, you will want to scroll to the final hint in each lab section, which is labelled "Phone a Friend." This hint will give you an almost complete script in which you need simply correct a few things or add a few values. You can simply copy this script and paste it into a file on your VM to solve the lab.

Scripting Behavioral Anomalies

Workbook exercises ”Zeek Scripting 3"

This page intentionally left blank.



- Architecture
- TLS
- Zeek
- IDS/IPS Evasion

Anomalies and Behaviors

Introduction to Zeek

Zeek and Signatures

Zeek Scripting Basics

Threat Analysis

Scripting Correlations

Scripting Behavioral Anomalies

Spicy

What About New Protocols?

We may need to work with protocols for which there is no support.

- Previously, you would have been required to use BinPAC and write C++ code to implement your own protocol analyzer.
  - This was not fun, nor was it easy.
- Robin Sommer set out to create a tool to solve this problem.
  - A major goal was to make the language as close to Zeek's scripting language as possible.

Spicy!

- A Zeek-like language for creating protocol analyzers

One problem that you may face is that there is a protocol in use in your organization that is somewhat unusual. This unusual protocol is perfectly normal for you, but since it isn't widely used outside of your industry, very few (if any) security tools have support for it. Yet, you need or want to perform monitoring and alerting for this protocol.

While we could write really complicated rules using **byte_test** and the related **byte_** options using Snort, wouldn't it be wonderful if we could generate events out of Zeek for this protocol? If you say, "Yes, please!" then you will be very interested in something that Robin Sommer, one of the Zeek developers and a member of the leadership of Corelight, has been working on for the past few years.

Adding a protocol dissector to Zeek has always been possible, but to do so you had to learn a lot about the internals of Zeek and then write your protocol dissector in C++. Nothing about this process was easy or fun. Robin decided he could do better.

His idea was to create a language and compiler that would allow you to define a new protocol analyzer using a language that was the same as, or at least very close to, the same scripting language that Zeek implements. What he came up with is Spicy.

Consider These Examples

DNS

```
type Header = unit {
    id    : uint16;
    flags : bitfield(16) {
            qr: 0;
            opcode: 1..4;
            aa: 5;
            tc: 6;
            rd: 7;
            ra: 8;
            z:  9..11;
            rcode: 12..15;
        } &bit-order = spicy::BitOrder::MSB0;

    qdcount: uint16;
    ancount: uint16;
    nscount: uint16;
    arcount: uint16;
};
```

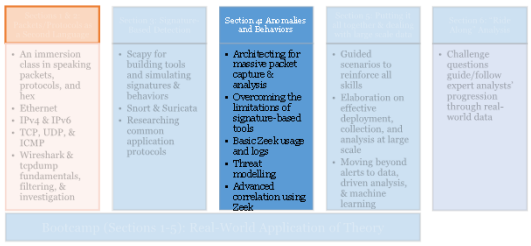
DHCP

```
public type Message = unit {
    op: uint8 &convert=Opcode($$);
    htype: uint8 &convert=HardwareType($$);
    hlen: uint8;
    hops: uint8;
    xid: uint32;
    secs: uint16 &convert=cast<interval>($$);
    flags: uint16;
    ciaddr: addr &ipv4;
    yiaddr: addr &ipv4;
    siaddr: addr &ipv4;
    giaddr: addr &ipv4;
    chaddr: bytes &size=16;
    sname: bytes &size=64
&convert=$$.split1(b"\0")[0].decode();
    file_n: bytes &size=128
&convert=$$.split1(b"\0")[0].decode();
    options: Options;
};
```


We aren't going to spend any time in class or in labs writing a protocol analyzer in Spicy. There are two reasons for this. First, we find that most students at this point in the class are absolutely exhausted. Add to that the fact that most of them are struggling to understand basic Zeek scripting, adding to this creating a protocol analyzer, and we are likely attempting to cross a bridge too far.

The second reason is more practical. Spicy has recently been released for production use with Zeek 4. Unfortunately, the project is still undergoing some rapid changes. Our experience has been that, if you want to use the Docker install of Zeek 4 with Spicy, everything works fine. However, if you are building from source (which many experienced analysts prefer to do), things are not so good. Spicy will generate a number of errors during its testing prior to install. If you override these and install anyway, everything appears to work, but the Spicy analyzers never generate any events. Oh well. We're certain this will be resolved. For now, if you want to experiment, the Docker version is the least frustrating way to get started. ☺

Have a look at the slide for a moment. What we are showing you here are two examples of what the Spicy language looks like. As you can see, it is also C-like and very much like the Zeek language. On the left, we can see the definition for a DNS header. Based on our experience, we can easily recognize the 16-bit query ID and the various flags, followed by the number of queries, answers, authorities, and additional records. Wow! Of course, it's not as clear as the header diagram, but converting from the diagram to that text doesn't look impossible. On the right, we have the header definition for DHCP. Can you puzzle out the meaning of most of the fields just from that code?



- Packet Crafting for IDS/IPS
- Wireshark Part III
- Application Protocols with Snort and Suricata
- IDS/IPS Evasion

Signature-Based Detection

Insertion, Evasion, and Denial of Service

In Book 4, we really begin to make direct application into the field of intrusion detection. Everything that we have covered so far has really been intended to provide you with a very firm grasp of network protocols and header dissection. While we have started to make some steps toward finding evil through BPF filtering, in this book we will work to master signature-based tools. To do so, we will make use of the most widely deployed IDS in the world: Snort (or the commercial equivalent, FirePOWER). We will also spend some time much later examining Suricata, another very popular open-source IDS that was forked from the Snort project in approximately 2010 but which has become an entirely distinct and powerful product in its own right.

Before we jump into the world of signature detection, however, we are going to do a bit of research into a packet crafting tool, Scapy, and round out our Wireshark abilities with some file extraction and forensic analysis work!

Introduction

- Landmark paper "Insertion, Evasion, and DoS: Eluding Network Intrusion Detection" by Thomas Ptacek and Timothy Newsham
- Many of the issues discussed have yet to be implemented or are exceedingly difficult to address in modern IDS/IPS solutions.
- IDS cannot know for sure if destination host will receive/react to a packet
- Insertion: IDS accepts a packet destination host rejects
- Evasion: Destination host accepts a packet IDS rejects

This 1998 paper is a landmark. Everything in it *still matters today*. Many of the attacks still work *with no changes!*

There is a seminal landmark paper written in 1998 called "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection." In it, the authors, Thomas Ptacek and Timothy Newsham, discuss attacks that can elude detection by the IDS (there were no IPS solutions at the time) by using methods of sending traffic that causes the IDS and the destination host to view packets sent differently. The paper is an excellent treatise on the different conditions that can cause an IDS to improperly analyze an attack. The authors conducted several different tests against the IDS solutions of the day to prove their theory. And although your initial instinct might be to dismiss this as ancient history, many of the techniques discussed in their paper have yet to be addressed in modern-day IDS/IPS implementations. Also, some of the issues that exist are problematic to address because of the difference in the ways that given operating systems handle them.

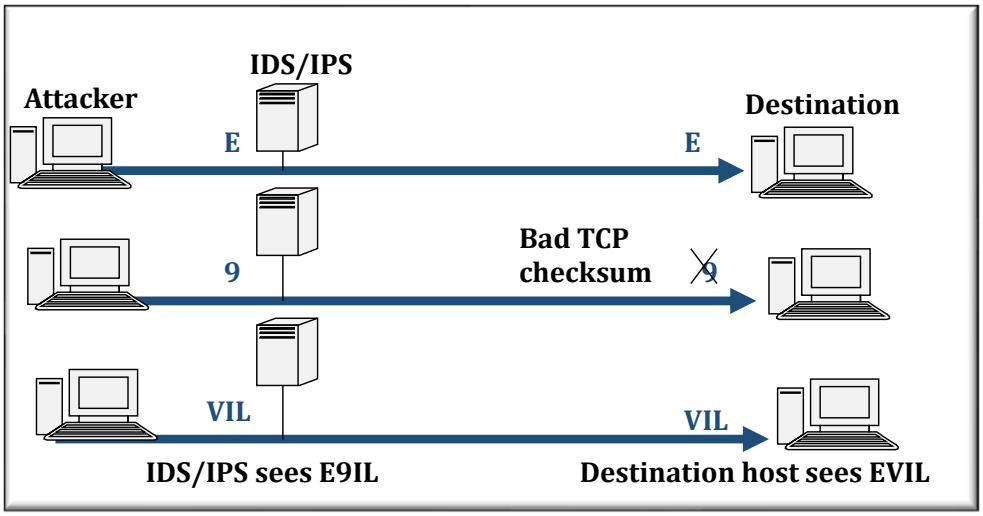
Along with the denial of service of an IDS, the paper basically discusses individual attacks that confuse the IDS. The first is known as *insertion*. This is where the attacker sends traffic to the destination host, where one or more of the packets will be accepted or seen by the IDS, yet they never reach the destination host, or if they do, the destination rejects them as faulty. The IDS and the destination host see different traffic or interpret it differently.

A second attack is known as *evasion*. This involves the same idea of sending traffic, yet this time the destination host sees all the traffic that the IDS does, but the destination host evaluates the packet differently than the IDS. Perhaps, the IDS discarded one or more packets that the destination host accepted. Again, the IDS and the destination host see the traffic differently.

Although the paper assigns a different name to each of these attacks, today we typically tend to refer to them with a single label of "evasion" because this is all-inclusive, and the end result for both is that there is a false negative.

Reference:
You can find this paper at:
http://insecure.org/stf/secnet_ids/secnet_ids.html

Insertion: IDS Sees Data That the Host Drops

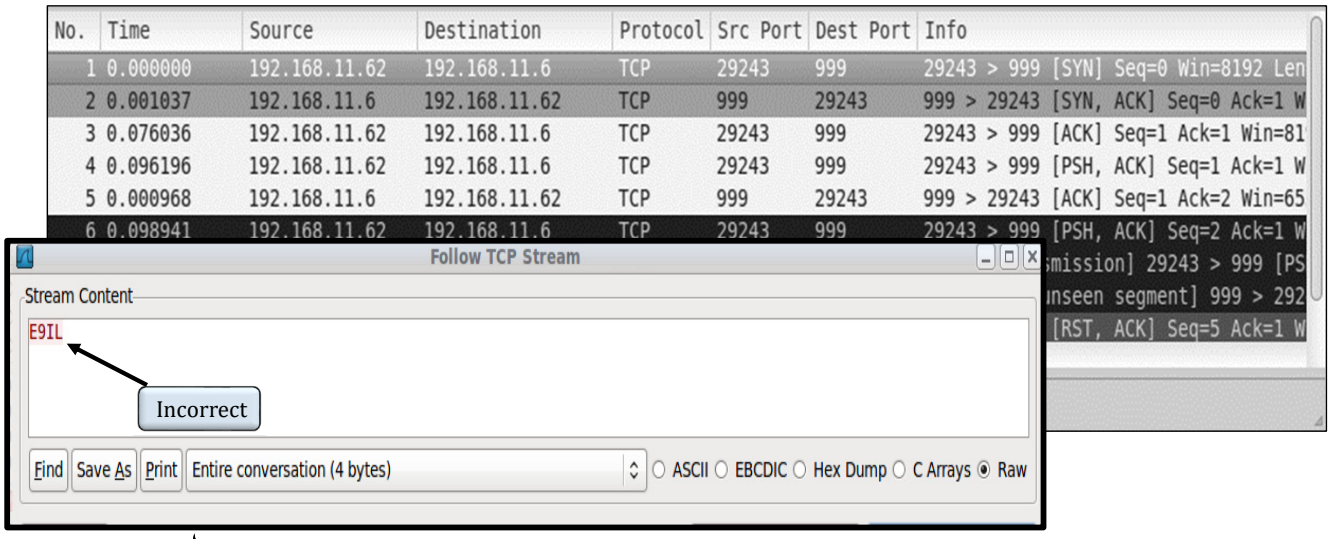


Let's examine how an insertion attack may work. Say that the IDS/IPS looks for signatures that may indicate some kind of problem or notable traffic. Suppose one of those signatures looks for traffic with a content of "EVIL" as a sign of some malicious activity. It is possible for the attacker to elude notice of the IDS/IPS if she can make the IDS/IPS accept a packet that the end host will not accept or will never see.

Let’s assume in this exchange that the three-way handshake has successfully been completed between the hosts. Next, the attacker sends three different packets destined for the target host, each with one or more characters in the payload. The first packet is a normal one that contains the letter "E", which both the IDS/IPS and the end host receive, examine, and accept. A second character of "9" is sent that has a bad TCP checksum. As you recall, checksums validate the integrity of the packet headers/data, and if not correct, the packet should be discarded. Say that the IDS/IPS sees this packet, does not validate the TCP checksum, and blindly accepts the packet as a valid part of the stream of characters sent to the destination host. The destination host receives the packet, validates that the TCP checksum is incorrect, and discards the packet. The attacker has managed to insert a character that causes the IDS/IPS to fail to recognize a real attack or action against the end host.

Finally, a third packet is sent with a payload of "VIL". The "V" has the same TCP sequence number as the previous segment that carried the payload of "9". The IDS/IPS ignores this because it has already acknowledged the payload of "9". It acknowledges the subsequent "IL" only. The destination host acknowledges the "VIL" because it dropped the previous segment with the payload of "9”. The outcome is that the IDS/IPS sees the payload as "E9IL”, whereas the receiving host correctly reassembles the payload as "EVIL". The attacker has managed to elude detection with this insertion attack because the IDS/IPS fails to see the true payload of "EVIL".

Insertion Attack Example



Let's simulate the session on the previous slide using a netcat listener on port 999 of host 192.168.11.6 and use Scapy to craft the traffic. The three-way handshake is established, and a payload of "E" is sent in the fourth packet. The server acknowledges this in record 5. Record 6 contains the payload of "9", but you can find a TCP checksum error if you expand the packet details pane with TCP checksum validation enabled in Wireshark. Record 7 reuses the sequence number from record 6 to send the "V" because that sequence number was never acknowledged by the receiving host. It consumes two additional sequence numbers for the "IL". Wireshark interprets this segment as a TCP retransmission because of the reused sequence number. Host 192.168.11.6 then acknowledges the 3 bytes sent.

As proof of the possibility for an insertion attack, look at the way that Wireshark reassembled the stream "E9IL". We've succeeded in duping Wireshark because it accepts the "9" on the segment with the bad TCP checksum. It should discard this segment, yet it does not.

There is an explanation for Wireshark's failure to discard this segment. Wireshark highlights a broken checksum value in the packet details pane when configured to validate TCP checksums. This is accompanied by an error about an incorrect checksum, the correct checksum value, and a possible explanation for the error, caused by "TCP checksum offload?" Checksum offloads transfer the checksum computation process from the host to the network interface card, much like TCP Segment Offload did for TCP segmentation. Consequently, this means that there is the possibility that the checksum will be corrected/provided by the NIC, and the TCP segment should not be deemed broken or discardable at this point.

To see the output, enter the following on the command line:

wireshark insertion.pcap

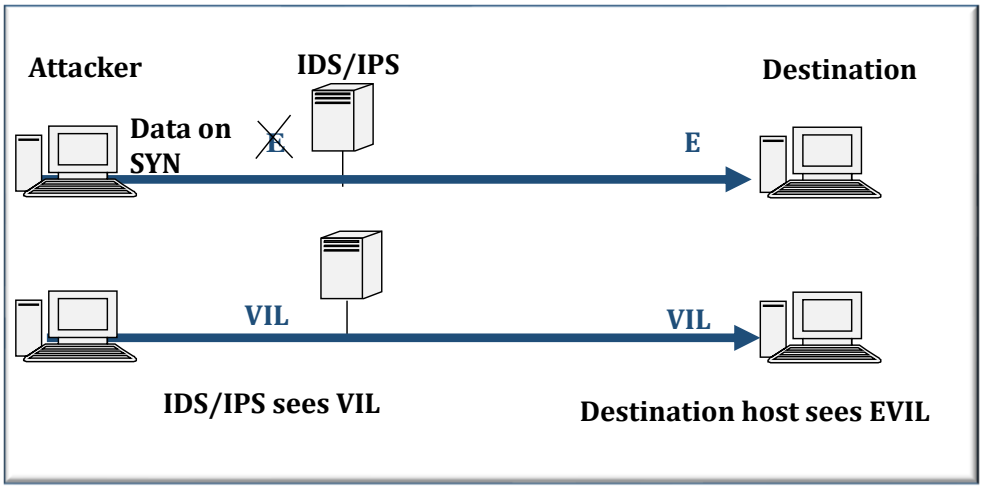
Select Analyze → Follow TCP Stream to see the reassembly.

What we witness is that Wireshark has made a decision to include the segment with the TCP checksum error in its reassembly. This isn't a logic flaw, just an interpretation choice. Wireshark is a magnificent tool, but as you see, it is not always perfect.

You may notice that there is no acknowledgement from the receiving server immediately after packet 6. Can that be used as conclusive proof that the receiving host discarded the segment with the bad TCP checksum? Not necessarily; this could mean one of two things. The first is that it was never accepted and acknowledged. The second is that packet 7 was sent so quickly after packet 6 that the receiver could have acknowledged the aggregated data of both packets in packet 8. It is impossible to tell from looking at the captured traffic alone. The only way to validate this is by examining what the destination host receives. Netcat displays the reassembled data as "EVIL"; therefore, Wireshark misinterpreted the situation.

We issued a caveat at the beginning of Section 1 about Wireshark, tcpdump, or any tool that does interpretation of traffic. The operative word is "interpretation" because, like an IDS/IPS, the tool is as accurate as the code and dissectors upon which it is built. That leaves you as the arbiter of the truth.

Evasion: The IDS Misses Data That the Host Accepts



In the case of evasion depicted in this slide, the destination host sees or accepts a packet that the IDS/IPS does not evaluate correctly. We look at another TCP session with a payload of "EVIL" sent to the target destination host. If the attacker can send the traffic in such a manner that the IDS/IPS discards or does not correctly evaluate a packet that the end host accepts, this eludes detection.

A possible scenario for this attack is sending data on the SYN connection. Although not typical except for in TCP Fast Open connections, which we discussed in Section 2, sending data on SYN is valid per RFC 793. Most operating systems do not accept data on SYN; however, there are some that do. Any data on a SYN connection should be considered part of the stream after the three-way handshake has been completed.

So, say we have a first packet that arrives on the network with a SYN packet destined for our target host, and it has a payload of "E" in the SYN packet. The IDS/IPS looks for payload only after the three-way handshake has been completed, so it totally misses that there is any data. The destination host receives the same packet and knows to store the "E" for the subsequent TCP stream after the three-way handshake is completed.

We then have the packets that complete the three-way handshake, each with no data in them, as expected. And, finally, we have a normal packet with the letters "VIL" as the payload destined for the target host.

The result is that the IDS/IPS sees a segment with a payload of VIL and a missing TCP sequence number from the "E" that it did not acknowledge. This content is not evaluated until the missing TCP sequence number arrives. The destination host, however, sees consecutive TCP sequence numbers, reassembles the stream as "EVIL", and accepts and acknowledges this malicious payload.

IP Layer Evasions

- Fragmentation overlap
- TTL variations
- IDS/IPS consequences with successful IP layer evasion:
  - Possible failure to detect any malicious traffic transported over IP layer

What if an attacker sends fragments with overlapping data? Which is the real data? The original data received or the overlapping data? Compounding these questions is that different operating system TCP/IP stacks either accept the original fragment data or favor the new data. Some operating systems take parts of the payload from both the original and overlapping data, depending on where the fragments overlap.

Another attack involves the Time to Live field. What if the attacker has done some reconnaissance on the topology of the network and surmises that the IDS/IPS is a hop or so away from the target destination host? If the attacker can set an initial TTL to expire at the next hop after the IDS/IPS, the attacker can successfully accomplish an insertion attack because the destination host will never see the packet.

A successful IP layer attack potentially has catastrophic consequences. Depending on how an IDS/IPS processes packets, it may be possible that all malicious traffic sent using a successful IP layer evasion technique will not be detected. If we look at the way Snort processes traffic it receives, it reassembles packets layer by layer before passing the parsed packet pieces to the detection engine to compare the traffic against a set of rules. Using the fragment overlap scenario (if packets are crafted to dupe the IDS/IPS into accepting a different set than the receiving host), all rules that examine traffic transported over IP (pretty much all of them) can be evaded.

Essentially, an evasion becomes more consequential and potentially more harmful the lower the layer on the TCP/IP model it pertains to. An application evasion threatens a specific application. A TCP evasion affects all traffic riding over TCP. An IP evasion encompasses all traffic with IP as the network layer.

TCP Evasions

- Bad TCP checksum
- Cause IDS/IPS to miss session beginning or prematurely terminate watching the session
- TCP sequence overlapping
- Abnormal TCP flag settings such as used in TCP Fast Open
- Manipulate TCP timestamp values
- Consequences:
  - Possible failure to detect any malicious traffic transported over TCP layer

There are many more ways to evade detection with TCP than IP because it is such a complex protocol. As we've discussed, it is imperative for an IDS/IPS to validate TCP checksums; otherwise, it may accept packets that the destination host rejects.

Evasion techniques include causing a failure of the IDS/IPS to detect the beginning of a TCP session or causing it to prematurely stop detection of a session. When we examine Snort rules, we'll see that Snort and other IDS/IPS solutions must look for malicious TCP traffic in the context of an established session. This means identifying the session establishment that the end host accepts and terminating the scrutiny of the session when the end host terminates the session. Although this seems like a trivial task based on obvious criteria, it is not straightforward at all. We discussed the four-way handshake, which is a variation of the three-way handshake that caused issues for IDS/IPS solutions in determining the session establishment. The TCP Fast Open also causes TCP reassembly issues.

Returning to the issue where the IDS does not validate TCP checksums, suppose a RST segment is sent with a bad TCP checksum. The IDS/IPS is duped into believing that the session has terminated, whereas it remains open because the receiving host drops it. Any malicious content sent thereafter will most likely evade detection.

After an IDS/IPS detects that a session has been established, it must reassemble the content of possibly many different segments. An attack that causes it to examine a segment that the destination host does not can cause issues. For instance, overlapping TCP segments that occupy the same TCP sequence numbers introduce ambiguity. Also, different operating systems may honor the original or overlapping segment, causing even more difficulty for correct IDS/IPS assessment.

In this section, we'll examine an evasion due to the evolving use of TCP Fast Open that has data on the SYN packet of a Fast Open subsequent session. The TCP timestamp option values are another technique that can be employed for evasions. The sender's timestamp value must be either equal to or greater than the last chronological acknowledged segment; otherwise, the segment should be discarded.

Many different situations and unique operating system interpretations can cause the IDS/IPS to evaluate a segment with manipulated timestamp values differently than the receiving host does. Yet, most IDS/IPS solutions do not even attempt to deal with TCP timestamps because either the developers are not aware of the evasion potential or because dealing with them is so difficult.

A single successful TCP evasion technique can have dire consequences for detecting malicious traffic. Any malicious payload transported over TCP may not be detected. Think about the many protocols that ride over TCP: HTTP, SMTP, and more. It is estimated that more than 80% and upward to 90% of internet traffic is HTTP. If you were to look at the default set of Snort rules, you'd find that approximately 90% are for attacks transported over TCP. Snort is not unique in its preoccupation with TCP traffic; every IDS/IPS solution must provide similar attention to TCP. You can see how a successful TCP evasion may have severe consequences.

Imperative for IDS/IPS to Validate Checksums

- Suppose IDS evaluates first two segments but doesn't validate checksums
- Then suppose IDS ignores second two segments because they overlap
- Receiving host gets both sets of segments and accepts exploit because checksum is valid

Send two segments with broken TCP checksum to established session

```
10.4.11.138.19538 > 10.4.10.64.80: Flags [P], seq 1, length 10
                    XXXXXXXXXX
10.4.11.138.19538 > 10.4.10.64.80: Flags [P], seq 11, length 18
                    XXXXXXXXXXXXXXXXXX
```

Broken TCP checksum

Broken TCP checksum

Send two overlapping exploit segments right after with good TCP checksum

```
10.4.11.138.19538 > 10.4.10.64.80: Flags[P], seq 1, length 10
                    GET /EVIL-
10.4.11.138.19538 > 10.4.10.64.80: Flags[P], seq 11, length 18
                    STUFF HTTP/1.1\r\n\r\n
```


Perhaps you wonder about the necessity and significance for an IDS/IPS to validate checksums. Let's take a specific example where we corrupt the TCP checksum so that it is not valid. This particular example may be more pertinent for an IDS because an IPS may be more likely to just drop an overlapping packet that is not a retransmission.

Say there is an already established TCP session to a web server in the network that the IDS monitors. For the purpose of this example, assume that "EVIL-STUFF" represents malicious activity. We send "EVIL-STUFF" in two separate segments, splitting the payload in the middle of what is most likely the IDS rule content of "EVIL-STUFF".

The reason that we split the payload between packets is that the IDS must reassemble the two segments to trigger an alert. It is possible for an IDS to treat exploit content found in a single segment differently than those found in multiple split segments because it has to reassemble those individual segments. For instance, an IDS may alert on a single payload of "EVIL-STUFF", even if the checksum is not correct or the TCP sequence numbers are wrong for the session, just because it sees exploit content. It's like catching the low-hanging fruit. In other words, it is better to generate a false positive than a false negative.

First, we send the two segments that have the same payload length and TCP sequence numbers of the ones we use for the exploit segments. But these two segments act as kind of a decoy because they have invalid TCP checksums and an innocuous payload of "X" characters. If an IDS does not validate TCP checksums, it does not alert or block these segments. But the destination host that receives these segments discards them because it validates the TCP checksums.

Now let's finish the attack by sending segments with overlapping sequence numbers with exploit content and valid TCP checksums. If the IDS believes that it has already evaluated segments with identical TCP sequence numbers and content, it may ignore these segments. It is extra work for an IDS to re-evaluate traffic it has already seen, so some ignore overlapping segments. It is possible for it to permit these exploit segments to reach the destination web server undetected. This is just one possible scenario of attempting to evade an IDS that fails to validate checksums.

Fortunately, most IDS/IPS solutions do checksum validations. Make sure any other software or product you employ, such as a SIEM that does session reconstruction, performs it too; otherwise, it too is evadable.

TCP Insertions: Timestamps

- Send a TCP segment with multiple timestamp options
  - If the first timestamp is too old, the IDS drops the packet.
  - If the first timestamp is OK, the IDS accepts the packet.
  - The hosts we have tested all properly drop the packet due to an old timestamp.

TCP, TS=1234	TCP, TS=2123 TS=1000	TCP, TS=3123
EV	Junk Data	IL

Let's give you two other "zero-day" insertion attacks. These will draw on your extensive and hard-won knowledge of TCP.

In the first, we are going to try to convince the IDS to accept data that the host does not. Remember that with PAWS, every TCP segment will have a timestamp option. The timestamp should increase for each subsequent segment that is sent. The receiving host knows that, if the timestamp is too old, it should drop that segment to avoid corrupting the stream.

What if we send three TCP segments? The first contains "EV", the first two characters in our attack string. The next segment has two timestamp options in it; the first option indicates a valid timestamp for the session, while the second timestamp is too old. In our testing, Snort and FirePOWER will improperly accept this data, inserting it into the analysis stream, relying only on the first timestamp in the options. All the receiving hosts that we tested, however, will properly drop the segment. Even though using two timestamps doesn't feel "valid," it turns out that the hosts can handle it appropriately. Finally, we send "IL", again with a valid timestamp. This means that the target has seen "EVIL“, while the IDS has seen "EVJunk DataIL". Attack successful!

TCP Insertions: Urgent Data

- Send a TCP segment with URG set and pointer > data
  - Yes, setting URG is anomalous.
  - Many applications will drop all data from *this* sequence number to the offset.
  - IDS will "insert" the urgent data up to the offset, allowing an insertion attack.

TCP	URG, pointer 32k	More TCP packets	More TCP packets	TCP
EV	32k...	of...	junk...	IL

The second example of a modern attack is a bit less covert. In this case, we're taking advantage of the lack of support in most applications for urgent data handling over TCP. Recall that we saw an example of one application, netcat, that has no support for this, so it simply acts as though the data was never sent.

The reason this attack isn't especially covert is that we will turn on the URG bit, which is like waving a flag. However, remember that an IDS will typically only store the packets that *trigger* the alert. At least in our opinion, it seems likely that the majority of analysts, if they even look, will simply dismiss it as an odd probe of some kind. This is especially true because the real attack isn't *in* the packet with the URG bit set!

Remember that when the URG bit is set, the Urgent Pointer indicates where the urgent data ends. Also, recall that this is a 16-bit field, allowing us to point well beyond the end of this packet. What if we sent a segment containing "EV", followed by a packet with URG set and a pointer that is, say, 32k in the "future." That means that as many as 22 or more packets will be required to send that urgent data. Finally, after that 32k of "junk" data goes by, we send "IL".

While this is application specific, chances are that the application actually *processes* "EVIL", while the IDS is processing "EV32k of junkIL", allowing us to evade the signature again.

History of IDS TCP Traffic Evaluation

- Evaluation of TCP traffic in the context of three-way handshake
- Early IDS TCP traffic evaluation not performed in context of three-way handshake
- DoS attacks by tools known as Stick and Snot overwhelmed analyst because of false positives
- Caused enhancements in IDS such as Snort:
  - Examine malicious payload in "established" session only (after three-way handshake)

Early in their evolution, IDS solutions were susceptible to denial-of-service attacks from tools such as *Stick* and *Snot*. These tools were aimed at Snort, which at the time had rules for TCP traffic that looked at payload only, not in the context of an established session after the three-way handshake. For instance, say that there was a rule to look for the content "foobar" in TCP traffic. At the time, this meant that any TCP segment containing foobar would cause Snort to alert. In reality, if a host received a lone PUSH segment containing foobar, it would return an RST because there was no established session. In essence, the Snort alert was a false positive.

The authors of Stick and Snot realized this and created tools that would craft TCP packets with content matching the many Snort TCP rules containing those content strings. The intent was to make Snort fire on all those many rules, thus overwhelming the analyst—a kind of DoS of the analyst, so to speak. Stick and Snot accomplished this DoS; however, the consequences were greater than just this. They exposed a critical weakness in Snort and just about every other IDS at the time.

In response to this, Snort was fortified to examine traffic only in the context of an established session, or after the three-way handshake. Snort was no longer fooled and didn't care that rogue segments with malicious payload were sent when not in the context of an established session. Snort examined the TCP streams for the initial SYN, SYN/ACK, and ACK to identify an established session. And all was fine in the land of Snort—well, mostly, as long as the session establishment followed conventions.

As we've seen, the four-way handshake and TCP Fast Open break with convention. This makes it more challenging to identify the beginning of an established session, potentially causing TCP evasions.

DoS Mitigation Creates Potential Insertion/Evasion

Snort's team took an additional tack to address stick and snot.

- Previously, Snort would generate an alert for every signature that was present.
- A hard-coded limit was added in the source code: no more than 5 alerts.
- Documentation claims you can set this to higher values, but in practice no more than 5 alerts can be generated.
  - Which alerts are generated can be prioritized, but they will effectively be random after that.

Can we leverage this for an insertion/evasion?

- Could we create a single packet/stream that will trigger many alerts?
- Could we leverage this to run an attack that will be detected but won't be logged?

The shift to requiring that content matches occur only within established sessions does seem to be a reasonable adjustment, though there are consequences. We'll look at one of those consequences on the next slide. Before we consider that, however, there is another tactic that the Snort developers took to mitigate this problem.

The Snort team realized that, even with an established session, it might be possible to create a single packet or stream in an established session that would match dozens of rules. They realized that this would be bad for Snort, effectively creating an amplification attack where we generate many logs for a single event.

In an effort to proactively mitigate this, the Snort team added a hard limit to the maximum number of alerts that can be generated for a single packet. You will not find this anywhere in the Snort documentation! In fact, the Snort documentation indicates that you can adjust the default limit of 3 to any number you'd like (Snort 2) or to a maximum of 32 (Snort 3). In practice, Snort will never generate more than 5 alerts on a single packet. If we have 20 rules that match, which rules will generate alerts? There are configuration options to set some priority for this, but assuming that the rules all have the same priority, the resulting alerts are effectively random.

Why random? Consider that this mitigation actually creates another evasion opportunity. Attackers can obtain the rule feed as easily as defenders. Could an attacker design an attack that will reliably generate 5 alerts that have nothing to do with the real attack, allowing the actual attack to slip by unnoticed? Yes! This is why the alerts are randomized. Still, the greater the number of rules a packet can trigger, the lower the likelihood the real attack generates the alert. This is effectively both an insertion (creating alerts that are irrelevant) and an evasion (the real attack doesn't alert) at the same time.

Snort and TCP Fast Open

- Can Snort detect the presence of the content "passwd" found in the URL of the SYN on the second TCP Fast Open session?

Rule:

```
alert http (msg:"Found passwd"; flow: to_server, established;
            content:"passwd"; sid:10000123;)
```

Testing the rule:

```
sans@sec503:/sec503/Demos$ snort -r tcp-fastopen.pcap -q \
      -c /sec503/Exercises/Day3/etc-snort/snort.lua -A alert_fast

17:07:13.347730 [**] [116:58:1] "(tcp) experimental TCP options found"...
```

No alert!

Another consequence of this DoS mitigation is that we do not inspect data on the SYN. Let's see whether Snort will find the content of "passwd" in the SYN payload of the Fast Open session with port 55534. This is almost certainly a sign of malicious activity.

The Snort rule seen at the top of the slide looks for TCP traffic from any source host where the HTTP service is in use. A message of "Found passwd" is displayed when an alert fires. The rule examines traffic with "passwd" in the payload only in the context of an established session. Finally, we assign a Snort ID number that appears in the output.

How does Snort do? Not well. While Snort reports that there are "experimental" options in use, it fails to detect the "passwd" string!

Snort Detection of Shellshock in DHCP

```
05/06-18:22:04.040957  [**] [1:31985:3] OS-OTHER Malicious DHCP server bash
environment variable injection attempt [**]
[Priority: 0] {UDP} 192.168.43.254:67 -> 255.255.255.255:68
```

Snort Rule

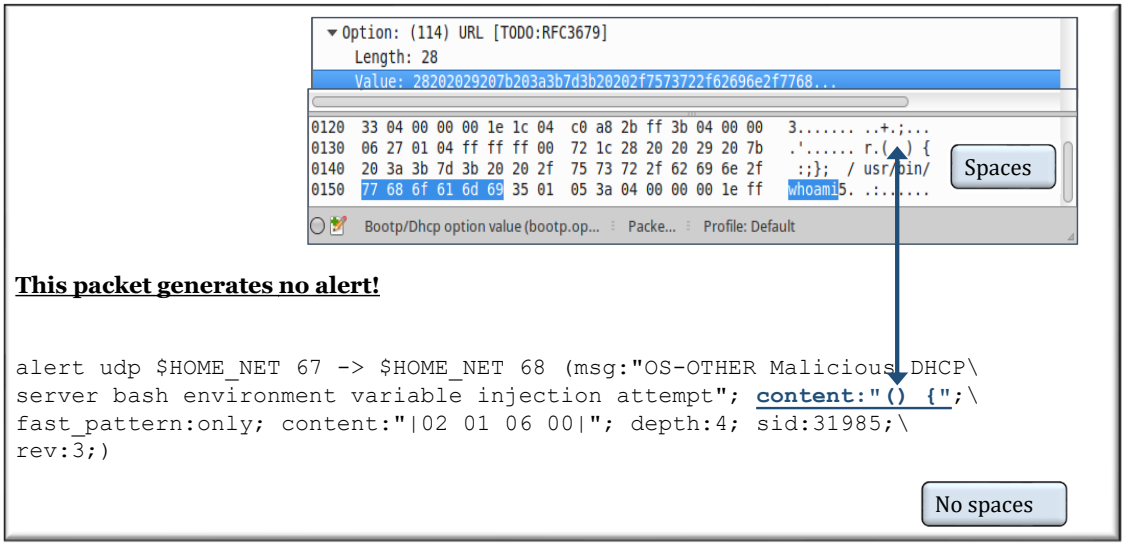
```
alert udp $HOME_NET 67 -> $HOME_NET 68 (msg:"OS-OTHER Malicious DHCP
server bash environment variable injection attempt"; content:"() {";
content:"|02 01 06 00|", depth 4; sid:31985;
rev:3;)
```

Consider this community Snort rule for the detection of Shellshock over DHCP. The upper part of the slide runs Snort, displaying the output on the screen, reading in **shellshock-dhcp.pcap**, and using the rule defined in the bottom part of the slide to detect the attack. This rule works just fine to detect exploits that use the proof-of-concept code with no modifications.

Let's look at the rule content only. The first content looks for `() {` for signs of Shellshock. The second content is hexadecimal, as enclosed between the pipe ("|") sign. This content looks for a DHCP boot reply (0x02) with a hardware type of Ethernet (0x01), a hardware address length of 6 (0x06), and with no hops (0x00) to identify a packet that may qualify for examination. The first content value is our focus. It turns out, as we'll see on the next slide, this is not a sophisticated rule and is easily evaded.

This has bigger consequences than might be immediately apparent. If a fast open option is used, not only will we miss the data on the SYN, but it will also shift all the data in the stream to the left! This means that our carefully written content rules will now be looking in the wrong places for *everything*!

Snort Shellshock Simple Evasion



The Snort rule to find the Shellshock content looks for `()␣{`. This community rule is unsophisticated because it is trivially evaded with whitespace, for instance, between the parentheses. The Scapy code that emulates a compromised/malicious DHCP server was amended to contain several spaces between the parentheses. Now, when the same attack is sent (but with the added space), the rule does not alert. Snort supports the use of regular expressions that would help in this instance. However, not only do we know that using a regular expression makes a rule very expensive, but there are also many ways to obfuscate or encode the signs of Shellshock, making it nearly impossible to write a rule to cover all manipulations of the rule content used to identify it.

Do not lose sight of our point. While we are using Shellshock to demonstrate the problem of evading signatures, the problem is not limited to Shellshock. While there are signatures that are not easily evaded because the vulnerability has very strict requirements for the payload being dered, the vast majority can be encoded and manipulated in endless ways. This means that evading signatures, for a knowledgeable and capable attacker, is trivial.

You can equate this to the state of antivirus tools. These are also, for the most part, just signature-based tools. Taking an existing piece of malware and recompiling it with different optimization options will render that malware undetectable. Similarly, making small changes to the code of the malware can have the same effect. This is why we see dozens of versions of a particular malware emerge over time.

Continuing in this line of thought, think about how frequently your organization updates malware signatures. Do you do so weekly? Daily? Or perhaps *hourly?* How frequently do we update our detection signatures? Far less frequently.

To be fair, updating your detection signatures hourly is unlikely to have any real value. This isn't because network attacks don't change rapidly; instead, it's because the organizations maintaining the feeds do not update them nearly as frequently as antivirus signatures!

Application Layer Evasion

AMEXEXPRESS_FORM JUNE ENCYP.html

```
<script>
<!--
document.write(
unescape("%3C%21DOCTYPE%20html%20PUBLIC%
/xhtml1/DTD/xhtml1-strict.dtd%22%3E%0A%0
/1999/xhtml%27%20xml%3Alang%3D%27en%27%3
rofile%20Update%20%7C%20American%20Expre
22%20content%3D%22width%3Ddevice-width%2
ent%3D%27%20Forgot%20User%20ID%20or%20Pa
09%3Cmeta%20http-equiv%3D%22content-lang
ible%22%20content%3D%22IE%3Dedge%22%20/%
```

- What's happening here?
  - Data to the right is in the HTML attachment.
  - Data below is a decoded fragment.

```
var jsObj = document.getElementById('iNCmnJS');
document.write("<scrip"  "t type='text/javascript' src='"  jsObj.getAttribute('data-uri')  "'></scrip"  "t>");
```

- Multiple application layer evasions!
  - Many Web Application Firewalls and browsers (and IDS/IPS devices) will look for `<script… content` as a sign of "badness."

Here's another interesting evasion, purely in the application layer. The content in the slide comes from a phishing style attack. The email contained an HTML attachment, inside of which is the content in the upper right. That content is entirely composed of URL/hexadecimally encoded data that will be unpacked using JavaScript. A small fragment of the decoded data appears near the center of the slide. What's happening?

What we are seeing here are layers of evasion! The attacker has attempted to directly evade detection by encoding the entire content of the page, including the JavaScript, inside of a massive string that is decoded and presented in the main browser window. Within that text, we find another evasion! Rather than simply including the `<script…>` tag with JavaScript, the word "script" has been split across two strings. Why would someone do this?

This is done for precisely the type of obfuscation that this section is all about: fooling your security systems into believing that there's nothing to see. In this case, by writing the code in unexpected ways, the attacker is attempting to defeat signature matching in the WAF, the browser, and in the IDS/IPS. Generally, these tools will not attempt to actually interpret or execute the code, but instead will be performing string matching. As soon as an attacker renders the string in an equivalent but unexpected way, a viable evasion opportunity emerges!

Nontechnical Evasions

- Payload is in a foreign language that the IDS/IPS doesn't understand.
- Attack timed during holiday vacation or less attended/unattended period:
  - Even if you are a 24/7 shop, are your top-notch analysts working at 0300?
- Attack timed to intentionally or unintentionally be a smokescreen during times of heavy traffic:
  - Who wants to attack Amazon on Prime Day?!

Not all evasions are technical or deliberate. Think about the situation where some attack payload has been translated to a foreign language. If the site's attack target host understands that language yet the IDS/IPS does not, an evasion is likely to occur. This is a trivial evasion, and perhaps not even deliberate for the attacker, yet a potentially hidden and unknown aspect for the defended network.

In a Black Hat Brazil presentation titled "Lost in Translation," Joaquim Espinhara and Rodrigo Montoro tested some Snort MySQL rules that had a content of "Access denied to user." In this case, the MySQL server supported a different language than English, so an error message returned did not match the content that the rule contained, resulting in an evasion. This seems so obvious yet had never before been explored. You can find slides from this presentation at http://www.slideshare.net/spookerlabs/lost-in-translation-blackhat-brazil-2014.

There have been a number of examples over the years of attackers, sometimes not even very advanced attackers, taking advantage of the fact that your employees go on vacation from time to time, especially around national and religious holidays. This can give them the time they need to penetrate your network or potentially spread much further within a network than they would normally be able to. Even without a holiday, attackers can choose a time that's not convenient for you. For example, are your best analysts working the overnight shift from midnight to 0800?

It could be that attackers just take advantage of how busy you are to conceal themselves in the masses. For example, if you had an attack you wanted to run against Amazon, wouldn't Prime Day be a great day to do it? There's so much other activity; the chances of you being noticed individually are much lower.

Target-Based IDS/IPS

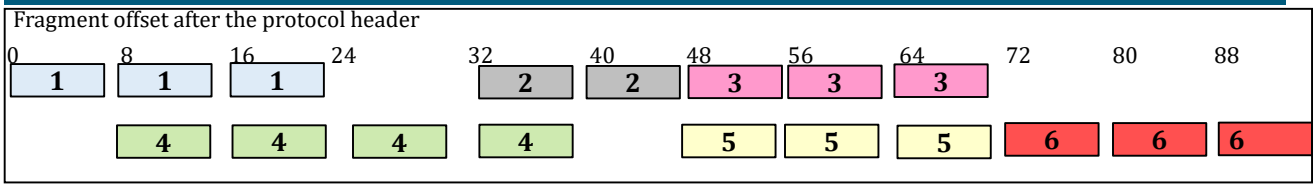
- IDS or IPS is aware of the operating systems running on the destination host.
- Can more accurately assess how a host reacts to stimulus traffic
- Does not help with all types of evasion/insertion attacks but can help IDS/IPS be more accurate
- We will look at IP fragment reassembly, but the same techniques can be applied to TCP reassembly.

One advancement in the attempt to deal with network and transport layer evasion and insertion attacks has been dubbed *target-based* intrusion detection or prevention systems. This is where the IPS or IDS is aware of some or all of the resident target or destination host behaviors in the network(s) that it is monitoring. A variety of ways exists to inform the IDS/IPS of the operating systems of hosts residing in the networks. It can be as simple, but painfully labor intensive, as having the administrator inform the IDS/IPS of the operating system identity or having a number of software packages (open-source and commercial) available to assist. For instance, what if you periodically had nmap run on a scheduled basis and inform that IDS/IPS of operating system identities? Or you may have a tool, such as p0f, that passively sniffs network traffic and attempts to identify operating systems.

These operating system identification tools are obviously not perfect, but they go a long way in helping the IDS/IPS assess whether traffic destined for the target is harmful. Say the IDS or IPS knows with a great deal of confidence that an Apache web server is the target for an IIS attack. There is no need to alarm the analyst over this traffic. If the IDS/IPS sends an alert, it can provide some kind of rating to assess the priority or danger of the attack. In this instance, it would be a low rating, if any is sent at all.

In the next few slides, we'll examine how this knowledge can be used to properly reassemble purposely overlapping fragments for proper interpretation. Knowing the operating system of a target host and the way it will react to a specific type of traffic can make the IDS or IPS more accurate in its assessment or interpretation of traffic.

Target-Based Fragmentation Reassembly Model



Fragment "1": Boxes labeled "1"	Offset 0–23
Fragment "2": Boxes labeled "2"	Offset 32–47
Fragment "3": Boxes labeled "3"	Offset 48–71
Fragment "4": Boxes labeled "4"	Offset 8–39
Fragment "5": Boxes labeled "5"	Offset 48–71
Fragment "6": Boxes labeled "6"	Offset 72–95

In 2003, Umesh Shankar and Vern Paxson released a paper titled "Active Mapping: Resisting NIDS Evasion without Altering Traffic." One of the sections discussed a "model" of overlapping fragments that could be sent as a stimulus to a given target host. They discovered that this same set of fragments would be reassembled five different ways by operating systems at the time. This has a lot of relevance when dealing with evasion or insertion attacks. For an IDS/IPS to reassemble the fragmentation as the target host, and assess whether it is malicious traffic, it has to be aware of the operating system of the destination host.

In the model of overlapping fragments discussed in the paper, six fragments are sent, each composed of partial fragments of 8-byte chunks. If you recall, the smallest length for a fragment is 8 bytes, just because that is how the IP header fragment offset value represents each fragment's displacement. Any fragment that is in the first row is an *original* fragment, whereas any fragment in the second row is an *overlapping* fragment. The original fragments are sent, and presumed to arrive, before the overlapping fragments.

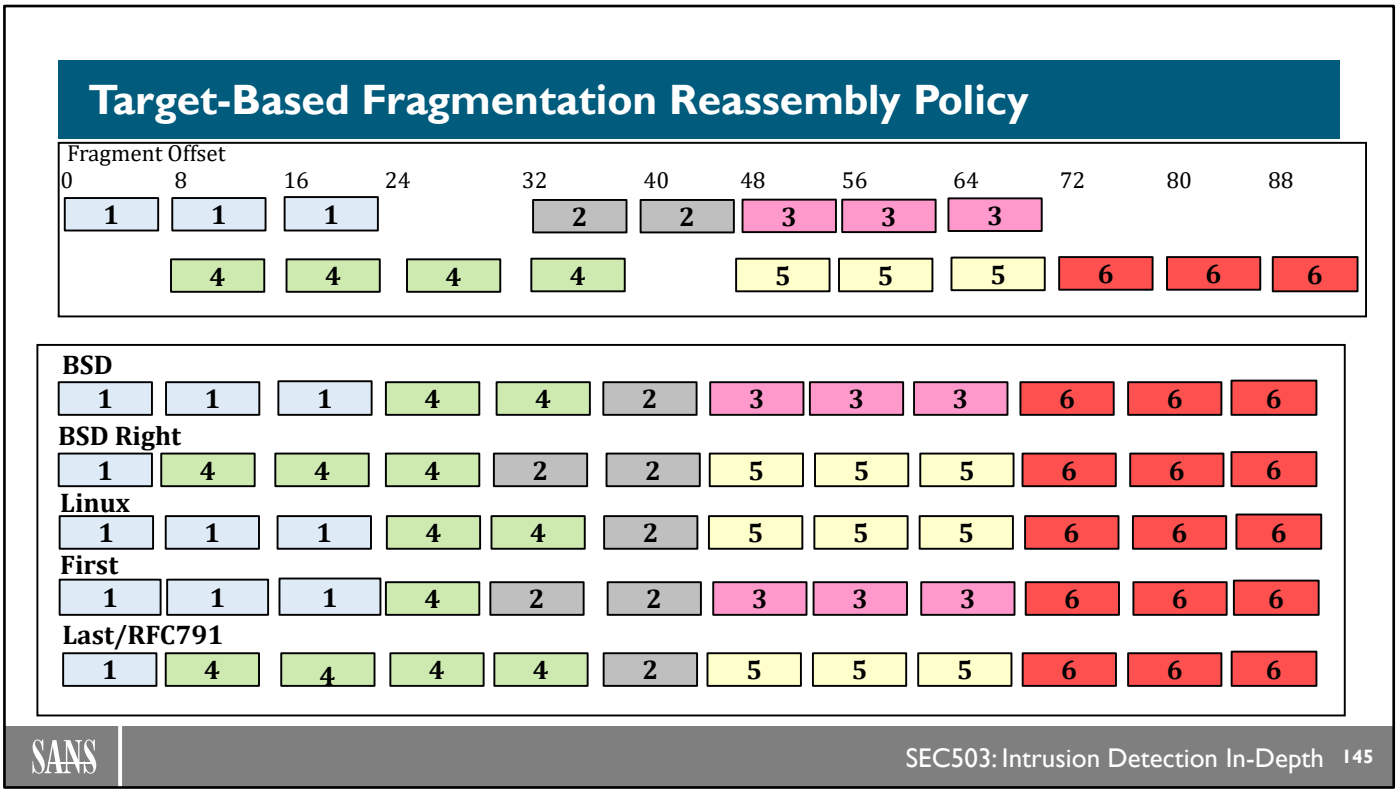
The small numbers at the top of the slide represent the offset of the fragments from the end of the protocol header. Data—whether it is a protocol header or actual data payload following the specified protocol header—that follows the IP header is considered to be part of the fragment. In our example, we use ICMP that has an 8-byte protocol header. The offset values shown are relative to the end of the protocol header. In the case of ICMP, an 8-byte protocol header begins the fragment. The payload that follows begins at offset 8. However, the diagram is a generic representation to accommodate, for instance, a TCP header, so it may begin at a different offset than 8.

Each rectangular box represents an 8-byte chunk associated with a fragment. For instance, fragment 1 begins at 0-byte offset from the end of the protocol header and has 24 bytes, or three 8-byte chunks. The content of each 8-byte chunk associated with the fragment in each of the six different fragment configurations contains the same value.

However, the values in each of the six different fragments are unique for determining the reassembly method used by the receiving host. We'll soon discuss how this works.

The fragments differ in content, sometimes in total length, and starting offset. The overlaps are fashioned so that there is a case of each of the following:

- A subsequent fragment wholly overlapping an original fragment with the same offset and length. (Fragment 5 does this with fragment 3.)
- A subsequent fragment partially overlapping and ending after an original fragment. (Fragment 4 does this with fragment 1.)
- A subsequent fragment partially overlapping and beginning before an original fragment. (Fragment 4 does this with fragment 2.)



As it turns out, at the time that the Shankar/Paxson paper was written, operating systems had five different means of interpreting the overlapping fragments. It is very likely that more modern operating systems may have more or a different reassembly of the overlapping fragments. Although dated, the paper contains a chart of which operating systems used each of the discovered policies, if you are interested.

Defense Against IDS/IPS Attacks

- Generic:
  - Malicious content placed in multiple fragments
  - Malicious content placed in 1-byte payload in multiple TCP packets
  - Solution: Snort, Suricata, Zeek, and others perform stream reassembly
- Target-based:
  - Overlapping IP fragment/TCP segment/QUIC/HTTP2 stream segments
  - Solution: Snort, Suricata, and others allow target-based IP/TCP reassembly
    - No one has anything covering HTTP2/QUIC reassembly oddities.
- Application Layer:
  - HTTP preprocessor allows matches against fully decoded and/or decompressed data
  - SMB, DNS, and other preprocessors for protocol anomalies

Evasions, by definition, cause false negatives. Snort is not susceptible to evasion attempts that split malicious payload among fragments or individual TCP segment packets. The **frag3** and **stream5** preprocessors reassemble fragments and packets into their original pre-fragmented and session states.

Marty Roesch, the original developer of Snort, was a pioneer in engineering Snort to be a *target-aware* IDS in an attempt to thwart evasions that capitalize on the unique ways that operating systems reassemble ambiguous packets. Take, for instance, an evasion attempt where an attacker uses overlapping TCP sequence numbers with different payload in a particular session. Suppose the original packet has a payload of "GOODSTUFF", while the overlapping packet has the payload of "EVILSTUFF", which represents some malicious content for which there is a Snort rule.

Let's assume that Snort accepts and evaluates the original packet payload of "GOODSTUFF" and ignores the overlapping payload of "EVILSTUFF". No alert will fire. Now suppose that destination host operating system accepts the overlapping packet with the evil payload. Snort fails to detect this, and the receiving host sustains some undetected malicious activity.

Target-based detection allows the user to configure Snort to associate a **frag3** or **stream5** *policy* to apply to a destination IP address based on the destination host operating system. The policy identifies the unique TCP reassembly to be applied when Snort is performing evaluation of traffic to that host. In this case, Snort would be configured to apply an appropriate policy to favor the overlapping payload for the particular destination host instead of a default policy of favoring the original packet payload.

Application-specific protocol decoders can provide some defenses against insertion and evasions at the application layer. For example, the HTTP preprocessor allows you to create content rules that will be applied to the data after it has been fully decoded or after HTTP messages have been decompressed. Other application layer preprocessors look for anomalies within the protocol, but unfortunately they do not add useful content-matching options, such as the DNS and SMB preprocessors.

How Do I Do This?

Snort/Suricata – Configure it

- Both support more than a dozen reassembly strategies.
- No one tells you how to know how your system reassembles data.
  - It's not as easy as identifying the operating system!
  - Remember, some NICs do offloaded stream reassembly, complicating things.
  - Try out the TBR.py script from the class repository/VM.

Zeek/Corelight

- Absolutely no effort to address this problem
- "We don't see that on large, well-behaved networks."
  - Of course, you don't… Your tool can't find it!

The question, then, is, how to I mitigate these detection holes? For Snort and Suricata, there's great news. For the IP and TCP reassembly issues, we just must configure our tools properly. Both monitoring tools support more than a dozen different reassembly strategies. As easy as this sounds, there is a small problem; How do I know which strategy I should configure for any particular host?

Unfortunately, the answer is less straightforward. It is not as simple as identifying the operating system because our higher quality NICs support TCP stream reassembly offloading. This means that the segment reassembly (and handling of the overlapping segments) is being handled by the NIC and not the operating system.

To help with this, you might try out the **TBR.py** script that can be found in the Scapy directory on the course VM or in the ShowMeThePackets course repository. This script isn't fast, but you can run it against a host to try to ascertain what the reassembly strategy is.

Unfortunately, there are no other automatic solutions for the other evasions that we have discussed. Writing good rules and doing some threat analysis of new rules added to the rule feeds is a good start, but it is a constant process. For us, we do attempt to detect attacks, but we spend much more time thinking about what "compromised" looks like and we write signatures and correlations to find those. In other words, we recognize that signature detection can and will fail.

You likely noticed that we excluded Zeek from the above comments. As much as we love Zeek, the sad news is that Zeek has absolutely nothing built into it to do any kind of target-based reassembly. When we tried to reach out to the Zeek development team to discuss this and present some of our findings, their response was that they, "don't see that on large, well-behaved networks." Leaving aside the question of what, exactly, a well-behaved network is, there's a more obvious issue. It is painfully obvious that you will never see something that your tool has no capacity to identify!

The takeaway perhaps is that you shouldn't be running just Snort, or Suricata, or Zeek. Instead, you should be running a signature-based tool *and* a behavioral or correlation tool, for example, Snort *and* Zeek, but never just one of the two.

Review of Evasion Theory

- Many different attacks for evasion and insertion
- May be successful because IDS/IPS cannot know:
  - How all different hosts (TCP/IP stacks) react to a given packet
  - Differences in network segments to destination hosts
  - How all target applications work
- Defense against these attacks?
  - Host-based IPS
  - Target-aware IDS
  - Advanced protocol decoders/normalization

Many techniques can be used for insertion and evasion attacks against an IDS/IPS. And many will be successful just because the IDS/IPS may not know the behavior and response of every possible destination host TCP/IP stack to various attacks. It also takes time for an IDS/IPS to be "reprogrammed" after an unconventional method is used to establish a TCP session. This was the case initially for the four-way handshake, the advent of ECN flags in a SYN packet, and, most recently, TCP Fast Open.

Many facets of the TCP/IP stacks differ among operating systems. Although keeping track of a lot of this information may be feasible for the IDS/IPS, understand that as you require the IDS/IPS to perform more functions and duties, the slower the IDS/IPS will become in processing all traffic. It is a trade-off of functionality and speed.

In addition, for attacks that attempt to elude the detection by the IDS/IPS using application layer "obfuscations," more functionality is required for the IDS/IPS to detect these. You are requiring the IDS/IPS to understand the actual application and react as the application would. Many now include more advanced protocol decoders that understand and normalize a particular protocol, such as HTTP.

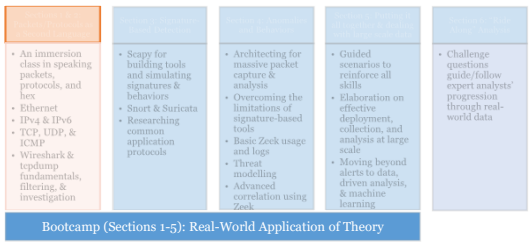
Requiring an IDS/IPS to perform all these functions is a tall order. Understandably, no IDS/IPS can possibly foresee every possible attack and detect it. Knowing all this, you see that it is impossible for the IDS/IPS to know the state of the network and all the behaviors of every destination host under its watch. So you have to recognize that an IDS/IPS is a best-effort solution. For that matter, no security software is 100% effective.

In some cases, a host-based IPS would be the best remedy to deal with attacks that try to elude the notice of the IDS/IPS. The host-based IPS sees and thwarts the activity because it can more accurately analyze it as the resident host does.

IDS/IPS Evasion Exercises

Workbook exercises "IDS/IPS Evasion"

This page intentionally left blank.



- Bootcamp

Section 4 Bootcamp

Going Further with Zeek

Applying Zeek and Your Scripts

This section's bootcamp score server questions rely entirely on Zeek

- Finding and correlating activities with Zeek logs
- Applying a few of the scripts that you created in the day's labs to real-world data

Before you start, discuss with your instructor ideas for scripts that would be useful during:

- Continuous monitoring
- Threat hunting
- Incident response

Don't be shy asking for help if you try to write a script of your own!!!

For this bootcamp, you will be logging in to the bootcamp score server and using the scripts created throughout the section to answer questions about data extracted from our real-world packet repository.

Before jumping into this, we'd like to invite you to engage in a discussion about possible scripts that you can imagine would be useful in your environment for monitoring, threat hunting, and incident response. If you are taking this class OnDemand, we would encourage you to listen to the recorded discussion from the class and invite you to share thoughts or ask questions of us in the Slack channel.

If you decide that you want to try to write a script to solve some of the problems in your own organization, please don't hesitate to ask your instructor for assistance!

Section 4 Bootcamp Exercises!

Real-world application of Zeek using the score server

This page intentionally left blank.

COURSE RESOURCES AND CONTACT INFORMATION

AUTHOR CONTACT
David Hoelzer
dhoelzer@enclaveforensics.com
BEST CONTACT: The Slack Channel!

SANS INSTITUTE
11200 Rockville Pike, Suite 200
N. Bethesda, MD 20852
301.654.SANS(7267)

#SHOWMETHEPACKETS
https://www.showmethepackets.com

SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.