GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

Interested in learning more?

Check out the list of upcoming events offering
"Security Essentials: Network, Endpoint, and Cloud (Security 401)"
at http://www.giac.org/registration/gsec

Risk Management with Automated Feature Analysis of Software Components

Author: Steven M. Launius, SteveLaunius@gmail.com

Advisor: Clay Risenhoover

Abstract

Organizations developing software need pragmatic risk management practices to prevent malicious code from contaminating their software. Traditional security tools for Static Code Analysis identify vulnerabilities, not the presence of backdoors exhibiting unintended actions. Application Inspector is a Microsoft tool released to the open source community that identifies risky features and characteristics of source code libraries. This research will evaluate the accuracy of feature detection in the Application Inspector tool and construct a risk model for automating decisions based on feature analysis of source code.

1. Introduction

Nefarious contributors can insert malicious source code at various steps in the production of software. An organization's developers, or third-party library dependencies, are potential avenues for introducing malicious code or backdoors. Security consultants, software development managers, and developers need practical solutions to address threats from insiders and within the supply chain. Many organizations prioritize new functionality over manual reviews of source code from third-party suppliers. Therefore, vetting source code for backdoors is not a common practice. Software security architecture focuses on reducing vulnerabilities in the fast-paced continuous delivery model known as DevOps: software development (Dev) and IT Operations (Ops).

Modern development methodologies and security practices are increasing the attack surface for backdoors. In DevOps, developers deploy small portions of code frequently by reusing source code from internal teams or third-party libraries, such as open source software (OSS), frameworks, or software development kits. The efficiency increase from this DevOps model comes at the expense of understanding how the code base operates (Flores, 2019). Additionally, DevOps provides little time for security reviews. A recent SANS Institute survey shows that many organizations use automated and manual application security practices, but most rely on manual testing and reviews (Bird, 2017). A similar Ponemon Institute survey found that most organizations in the financial services industry conduct an assessment after the software's release. Furthermore, even though respondents are concerned with the security of third-party software suppliers, most do not inventory or manage OSS (Ponemon Institute, 2019). The respondents from both surveys use security practices designed to identify vulnerabilities rather than undesirable behavior in their software. For example, automation tools for software component analysis target known flaws in third-party software libraries (Springett, 2020). Of all the security practices, manual source code review is one of the most effective means to identify malicious behavior in code, but it is also the most resource intensive. Insertion of backdoors into a software development lifecycle is a tactic that adversaries employ with damaging effects.

Steven M. Launius, SteveLaunius@gmail.com

Malicious code inserted into an organization's software product can have tragic consequences. One of the most devastating supply chain attacks, which exploited a trust relationship between an organization and a third-party software supplier, occurred in 2017 when a nation-state was blamed for inserting a backdoor into a Ukrainian company's software update to distribute a destructive malware named NotPetya. The resulting global damages are estimated at $10 billion (Greenberg, 2018). Cyber-criminals and nation-states also target OSS in supply chain attacks. A recent attack on a common OSS JavaScript package named *conventional-changelog* tricked organizations into unwittingly including crypto-mining source code in their products. The attackers inserted their malicious code by compromising the credentials for the developer's code repository (Howard, 2018). In another attack on OSS, adversaries added cryptocurrency stealing code into a popular Python package named *Colourama*. Developers unintentionally contaminated their software products after downloading this trojan source code from a typo-squatting website (Bennett, 2018). Insider threats come from developers intentionally or unintentionally altering functionality requirements of the software. In 2016, an OSS developer became upset and removed a widely-used JavaScript module breaking a massive number of other projects that depended on it (Tung, 2016). More recently, a disgruntled contract developer used a backdoor on an NHL-affiliated smartphone app to demand unpaid wages from his former boss using the app's messaging system that was viewable by all customers (Elliot, 2019). Whether backdoors are surreptitiously inserted into the supply chain by an adversary or unintentionally placed there by a developer, understanding the functionality of software through source code analysis can help organizations reduce risk.

An automated tool for analyzing source code features can help organizations implement a practical risk management process to prevent unintended effects from the features in their software products. Application Inspector is a Microsoft tool released to the open source community that identifies risky features and characteristics of source code libraries for many programming languages. This tool's primary use cases identify high-risk components or unexpected features and changes in a component's feature set over time (Acosta & Scovetta, Introducing Microsoft Application Inspector, 2020). This research will evaluate the accuracy of feature detection in the Application Inspector tool

Steven M. Launius, SteveLaunius@gmail.com

and construct a risk model for automating decisions based on feature analysis of source code.

2. Application Inspector

On January 16, 2020, Microsoft publicly introduced the Application Inspector tool as an OSS project. Like other organizations employing DevOps, Microsoft's development teams reuse source code from different internal developers and third parties. Source code reuse comes with an inherent risk of trust in the supplier of the code. While employees arguably deserve more confidence due to Human Resource practices and incentives, there may be no way to have similar screening for third-party suppliers. Microsoft produced Application Inspector to reduce this risk with a new static code analysis tool. The tool's source code analyzer identifies features and metadata that could be abused to perform undesirable actions in the software development lifecycle. A person manually reviewing the code could analyze and conclude its intentions. However, the tool provides analysis capabilities at scale for libraries with many files and millions of lines of code. The source code's characteristics are reported without judgment. The tool's use cases include identifying feature set changes between library versions and identifying high-risk features that should receive additional review (Acosta & Scovetta, Introducing Microsoft Application Inspector, 2020).

Application Inspector is a command-line tool with characteristics indicative of its maker. The principal Microsoft developers, Guy Acosta and Michael Scovetta, wrote this cross-platform tool for the .NET open source developer platform using the C# programming language. Microsoft publishes Application Inspector on the public code repository GitHub and NuGet, a Microsoft-supported mechanism for sharing .NET code. The tool's GitHub project site contains a Contributors section showing four developers, including the principals, actively supporting it. A wiki in the project site contains thorough documentation of the tool's features and underlying structures (Acosta, Home · microsoft/ApplicationInspector Wiki, 2020). The tool supports scanning a variety of programming languages, in similar or mixed language files, including: C, C++, C#, Java, JavaScript, Hypertext Markup Language (HTML), Python, PowerShell, and many more.

Steven M. Launius, SteveLaunius@gmail.com

The output formats for reporting include HTML, JavaScript Object Notation (JSON), or plain text. The tool's architecture and features relevant to this research will be explored in the following sections.

2.1. Key Features & Structures

The *analyze* command is the primary functionality of interest in this research as it enables the identification of high-risk features. This command recursively scans a given directory for supported programming language files, which may be in TGZ or ZIP compression and archival formats. There is a default set of over 400 rules defining search patterns to detect features within the source code of each file and characterize them using recognizable tag names.

Rules use a simple structure for analysis to identify features and assign them to tag names. A rule is defined in JSON formatted files with one or more string patterns to search, as shown in Figure 1. Each search pattern creates a regular expression (regex) that will match the string in the source code. Each pattern can have modifiers, conditions, or scopes to alter the regex behavior. Rules associate with an identifier, name, and tag. Tag names have a naming convention with common terms and a hierarchical structure to systematically identify generic and specific features of well-known programming language components.

Steven M. Launius, SteveLaunius@gmail.com

```
 1 □[
 2 ⊞  {
25 □  {
26       "name": "Network Connection: Socket",
27       "description": "Network Connection Socket",
28       "id": "AI032000",
29 □    "applies_to": [
30         "python"
31       ],
32 □    "tags": [
33         "OS.Network.Connection.Socket"
34       ],
35       "severity": "moderate",
36 □    "patterns": [
37 □      {
38           "pattern": ".bind(",
39           "type": "string",
40 □        "scopes": [
41             "code"
42           ],
43           "modifiers": [],
44           "confidence": "high",
45           "_comment": "Python socket"
46         }
47       ]
48     },
49 ⊞   {
70 ⊞   {
94 ⊞   {
127 ⊞  {
149 ⊞  {
180 └]
```

Figure 1. Example Application Inspector Rule in JSON Format

The tool can format reports in HTML, JSON, or plain text. Opening the HTML output in a web browser provides an excellent visual representation of results with a variety of default graphs and analytics. The JSON format is useful in automation environments like DevOps. Humans can read and write the JSON format, as shown above in Figure 1, as easily as computers can generate and parse it. The ability to quickly parse the results enables continuous deployment activities to incorporate this tool. Organizations wishing to establish security requirements for source code libraries can map detected features to them.

2.1.1. Command-Line Options

This research uses several options with the Application Inspector's *analyze* command for analysis. The -*s* option takes the full path to the directory containing the files for the targeted library. A custom rule file in JSON format can limit the tool's *analyze* command to specific rules when providing the file's full path in the -*r* or --*custom-rules-path* options. The -*i* , --*ignore-default-rules* options will exclude the default

Steven M. Launius, SteveLaunius@gmail.com

rules from the analysis. The *-c, --confidence-filters* options with a value of 'high,medium,low' includes all pattern matches regardless of its confidence rating. The author of a rule should set the confidence level of a pattern according to its precision or granularity. The *-o , --output-file-path* options take the full path and file name of the output file. The *-d , --allow-dup-tags* option and *-f , --output-file-format* option with a value of 'json' work together to list duplicate pattern matches for the same rule in a JSON formatted file. The *--single-threaded* option disables parallel processing.

2.2. Automated Tool Comparison

Using automated tools for static analysis of code is common during the implementation phase of the software development lifecycle. Static analysis tools integrate into an automated build process or a programmer's development environment (Bird, 2017). These tools use various techniques to catch mistakes in the code before execution occurs. Static analysis excels at identifying code errors and specific types of vulnerabilities as well as enforcing coding standards at scale (Morgan, 2018). There are dozens of either commercial or open source tools that can improve a developer's code quality.

Software Component Analysis tools address specific software supply chain risks from OSS and other third-party libraries. This type of analysis begins with an inventory of subcomponents, which is the first control in the 20 prioritized security controls from the Center for Internet Security and is vital to understanding the organization's assets that require protection. The focus of automated tools for software component analysis is on identifying and managing software composition, licenses, known vulnerabilities, and outdated components of third-party software. While understanding each component's purpose or function should be part of a Component Analysis strategy, a review of some of the available tools found this capability was missing (Springett, 2020).

2.2.1. Differences in Application Inspector

The Application Inspector tool disregards errors and flaws; instead, it provides an objective feature analysis of code libraries. The tool reports the presence of a feature and requires the user to determine whether it is expected. The principal developers proclaim

Steven M. Launius, SteveLaunius@gmail.com

their tool is not a replacement for manual security code review or static analyzer tools (Acosta, Home · microsoft/ApplicationInspector Wiki, 2020). However, automated validation of a code library's claimed objectives integrates well into DevOps' continuous delivery model.

3. Accuracy Testing

For security practitioners and developers to gain trust in the Application Inspector tool, this research will test the precision of its analysis feature. DevOps teams need assurance that the tool will detect features correctly before integrating a tool that could stop an automated process when identifying an unexpected or risky feature. This research will count the true and false positive results as well as true and false negative results from the *analyze* command for various code libraries. This type of quantitative testing is well-suited for accuracy tests since all possible outcomes are considered for a sample set of libraries.

3.1. Lab Environment

The simulation of a DevOps environment is achieved through a virtual machine (VM) lab environment. The host system specifications include ample hardware resources to support running at least one guest system with VMware's Workstation Pro v15. The guest system configuration represents either a developer's workstation or a build server for continuous deployment. The guest's specifications include one 64-bit CPU with two cores, 4GB of RAM, and 20GB of hard drive space running the Ubuntu v20.04 operating system. The Ubuntu installation includes its default packages for the Linux 5.4.0-29-generic kernel. The selection of Linux as the lab operating system is based on the popularity of this platform in a DevOps environment and the Application Inspector's cross-platform support.

Testing requires installing the Application Inspector tool, its dependencies, and a programming language on the guest system. Simple installation instructions in the *justRunIt.md* file on the tool's GitHub repository recommends downloading the tool and installing the .NET Core pre-requisites (Acosta, GitHub -

Steven M. Launius, SteveLaunius@gmail.com

microsoft/ApplicationInspector, 2020). The latest version at the time of testing the Application Inspector tool v1.2.11 was selected for this research. Downloading from GitHub repositories requires installing the *git* package. The pre-requisite .NET Core runtime v3.1.4 instructions for Ubuntu v20.04 from Microsoft's documentation page were followed to complete the installation (De Gorge, 2020). Finally, as the selected programming language for testing, Python v3.8.2 requires installing the *python* package. The Ubuntu administrative account, *root*, is needed for the installation of system-wide packages.

3.2. Testing Methodology

Accuracy testing for binary results from pattern matching lends itself to analysis with a confusion matrix. The confusion matrix is a data science tool for calculating the accuracy of a classification algorithm (pratz, 2019) (like pattern matching) that is based on the error matrix (Stehman, 1996). This experiment defines the two classifications as positive when *a pattern matches on either an identifier name or stored data* or negative when *a pattern does not match on either an identifier name or stored data*. A manual code review will determine whether the predicted pattern match from the tool's results is true or false. Accuracy will be calculated for each pattern as the sum of true positives and negatives divided by the sum of all possible outcomes:

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + True\ Negative + False\ Positive + False\ Negative}$$

The testing scope will consist of a sample of rules based on a common tag name. The *OS.Networking.Connection* tag is associated with a set of seven rules, summarized in Appendix A, and a total of ten regex patternsAppendix AAppendix AAppendix AAppendix AAppendix AAppendix AAppendix AAppendix AAppendix AAppendix AAppendix A. Manual review of the tool's *analyze* output for each test will confirm a positive or negative of each rule match in all of the library files. This analysis method marks a rule as true positive as long as one match was a true positive, even if other duplicate matches are false positives.

Steven M. Launius, SteveLaunius@gmail.com

3.2.1. OSS Test Libraries

The selection criteria of OSS libraries for testing in this experiment use the above methodology and scope. Library candidates will have primary features for the OS.Networking.Connection tag and, to minimize manual code review time, will contain a dozen or less Python source code files. Three OSS libraries were selected by searching GitHub for strings from the rules of the OS.Networking.Connection tag. Then filtering those results by the programming language type. The TFTPY, HRShell, and Human_curl libraries establish various network connections and contain less than twelve source code files. An additional custom script was created to test a possible false negative result, where its full source code is listed in

Steven M. Launius, SteveLaunius@gmail.com

Appendix B.

3.2.2. Test procedures

This experiment tests each OSS library twice with Application Inspector's *analyze* command. The first test analyzes the project's files without alteration. The second test analyzes the project's files after pre-processing them with *pyminifier*. The pyminifier tool minimizes Python source code files by removing non-executable code and unnecessary white space. Additionally, pyminifier can obfuscate individual identifier names by altering the original characters to make the names illegible (McDougall, 2014). Both minimization and obfuscation simulate attackers' techniques designed to make their malicious code more difficult to identify and review. The second test will evaluate the impact attackers' obfuscation techniques may have on the tool's accuracy.

The experiment will test the tool's *analyze* command with these options based on the following testing procedure:

1. Download OSS Library from GitHub.

```
git clone <URL to GitHub project file>
```

2. Analyze the OSS library with Application Inspector.

```
dotnet ~/ApplicationInspector_1.2.11/ApplicationInspector.CLI.d
ll analyze -s <path to OSS library directory> -d -f json -i -r
<path to custom rules file> -o <path to JSON output file> --
single-threaded
```

3. Manually review JSON results file to determine rule classification in the confusion matrix.

4. Minimize and obfuscate the OSS library's source code with pyminifier.

```
pyminifier --obfuscate <path to OSS library directory>
```

5. Analyze the altered OSS library with Application Inspector.

```
dotnet ~/ApplicationInspector_1.2.11/ApplicationInspector.CLI.d
ll analyze -s <path to OSS library directory> -d -f json -i -r
<path to custom rules file> -o <path to JSON output file> --
single-threaded
```

6. Manually review JSON results file to determine rule classification in the confusion matrix.

Steven M. Launius, SteveLaunius@gmail.com

3.3. Results

The overall test results show the tool has mostly high accuracy that can easily be improved with minor changes, but one false negative reveals a limitation. The results in Table 1 below show accurate results for all tested tag names except one, *OS.Network.Connection.HTTP*. This tag consists of two rules with three patterns, each with a different confidence rating of low, medium, or high. The pattern with a string of "https*:\/" and a low confidence rating is responsible for all false positive matches from this rule. In the first TFTPY test, this pattern matches an HTTP Uniform Resource Locator (URL) in a multiline comment. Pre-processing the TFTPY library with pyminifier in the second test (see Table 1) removes this false positive because pyminifier removes all comments from Python source code. However, the test results for the HRShell library pre-processed with pyminifier do not remove the false positive for this rule. This match resulted from an HTTP URL in a string for a Python function that displays it as output. Since HRShell is not requesting this URL, it is classified as a false positive match. This result is expected as the pyminifier tool does not alter strings assigned to a variable or passed as a parameter to functions.

Library Name	OS.Network.Connection.Miscellaneous	OS.Network.Connection.Socket	OS.Network.Connection.Http	OS.Network.Connection.RPC	OS.Network.Connection.General	Accuracy
TFTPY	TP	TP	FP	TN	TN	80%
TFTPY (pyminifier)	TP	TP	TN	TN	TN	100%

Steven M. Launius, SteveLaunius@gmail.com

HRShell	TN	TP	FP	TN	TN	80%
HRShell (pyminifier)	TN	TP	FP	TN	TN	80%
Human_curl	TN	TN	TP	TN	TP	100%
Human_curl (pyminifier)	TN	TN	TP	TN	TP	100%
Custom Script	TN	TN	FN	TN	TN	80%
Custom Script (pyminifier)	TN	TN	FN	TN	TN	80%

Table 1. Accuracy Test Results

Note: The abbreviation definitions in the above table are true positive (TP), true negative (TN), false positive (FP), false negative (FN).

Of the ten regex patterns tested, only the pattern assigned a low confidence rating produced false positives for all the tested rules. More than 400 default rules exist in this tool with over 800 pattern matches, and only 12 are assigned a low confidence rating. Therefore, analysis with Application Inspector should use the default confidence filters of high and medium and exclude the low confidence patterns. However, the false negative results represent a more severe limitation with Application Inspector.

3.3.1. Known limitations

Besides matching on multiline comments in Python, the false negative test results for patterns in the OS.Network.Connection.Http rule demonstrates a limitation of Application Inspector. A true positive pattern match for the string "import parse_http" was found during a manual review of the results from Human_curl, but no identifier names with "curl". The OS.Network.Connection.Http rules contain a string pattern for "curl", but only matches on word boundaries. Therefore, this pattern never matches even though there are dozens of identifier names containing the string "curl" in the source code for this library. The Human_curl library depends on the *pycurl* library, but it was not included during analysis because it is a standard Python library. This observation identified a possible false negative scenario where an HTTP connection is made, but the tool may not detect it.

Steven M. Launius, SteveLaunius@gmail.com

A test was devised to create a false negative scenario for an HTTP connection using a Python standard library. The rules in Application Inspector for HTTP connections do not contain patterns to match the *urllib* library, which is a standard Python library. A Custom Script (see

Steven M. Launius, SteveLaunius@gmail.com

Appendix B) was designed to create an HTTP connection using urllib. The HTTP URL string was broken into multiple strings and concatenated to avoid matching the regex pattern "https*:\/". This script was tested using the same process as the other OSS libraries. The output from Application Inspector found no matches for any of the OS.Network.Connection.Http rules. A summary of the known Application Inspector limitations is as follows:

1. Multiline comments are not ignored.
2. Rule patterns do not consider all standard libraries in the Python programming language.
3. Obfuscation of identifier names and strings can avoid pattern matching in custom libraries.

There are solutions to overcome each of these limitations. The first limitation is easily resolved by pre-processing source code files to strip comments. Second, contributing new rules with patterns that match more standard libraries for a particular programming language will improve the tool's accuracy. For example, the rule with ID AI032500 and tag of OS.Network.Connection.Http could be enhanced by changing the regex pattern to "import .*(http|requests|urllib)" or creating a new rule with a similar regex, which addresses not only the second limitation but also the third. A standard library call must be invoked somewhere in the source code regardless of obfuscation techniques. Therefore, additional rules identifying more standard libraries will increase accuracy.

3.3.2. Anomaly

Application Inspector's parallel processing capabilities produced different results. During this experiment, tests are run twice on the same OSS library. Once with the files unaltered and then again with source code files pre-processed by pyminifier. When a pattern match is not identified in comments of the source code, the match count for unchanged and pre-processed tests should be the same. The TFTPY library exhibits this characteristic. However, a manual review of both tests on the TFTPY library resulted in different match counts. This flaw was reproduced by testing the same OSS library

Steven M. Launius, SteveLaunius@gmail.com

multiple times. Adding the *--single-threaded* option for subsequent analysis tests produced consistent results.

4. Risk Modeling

This research will develop a use case and test a risk model for a practical implementation of Application Inspector. The use case consists of a security assessment for new libraries that developers create for an organization's internal web application. The web application is necessary for critical processing but does not contain customer or employee confidential information. The programming language of the web application is not significant as this research focuses on feature analysis. The developers are internal employees or contractors creating their code and reusing code from OSS or commercially licensed third-party sources to complete tasks on time. Application Inspector analysis could be performed by the developers, but should be performed by an independent, automated process. This process would automatically analyze new sets of source code to identify high-risk feature sets that should then have an independent manual review before deployment. Analysis results that are not high-risk should be trusted for automatic deployment.

The greatest impact in this use case will come from threats incorporating malicious code into the web application. Since new code is incorporated as sets of new source code files, Application Inspection will demonstrate analysis on a group of files to identify a feature set based on a threat model. Threat modeling will identify the most likely features necessary to accomplish the adversary's goal. Finally, testing of the model will demonstrate whether the risk model is useful for deciding when to perform a manual review.

4.1. Threat Model

Based on the risk model use case above, threat modeling will identify the most likely attack vectors and the techniques necessary for it to be successful. These techniques will be mapped to features in source code through Application Inspector's feature analysis of OSS libraries that simulate the same techniques. The primary features

Steven M. Launius, SteveLaunius@gmail.com

of the analysis will be grouped together to form a threat model for a particular technique. This research will focus on one attack vector and the most dangerous techniques for accomplishing an adversary's goal.

Common adversary objectives include preventing system operations or obtaining sensitive information that can lead to monetary compensation. The following malware types enable these objectives, and the organizational impact for the above use case is given for each:

- Ransom/wiper malware (high impact)
  Disabling a critical business web application can cost organizations thousands of dollars per day.
- Credential stealing (moderate impact)
  This type of malware is one step of a larger attack, often paired with Ransom/wiper malware to spread the infection.
- Data theft (low impact)
  The use case focuses on business-critical operations that do not involve sensitive data; therefore, the impact is low.
- Crypto mining (low impact)
  An increase in resource utilization will add up over time, but the system still functions, and data is not lost or stolen.

This research will use the NotPetya wiper malware as the example threat for modeling as it represents ransom/wiper malware that has the highest impact in this use case.

The MITRE ATT&CK framework is a knowledge base of adversary tactics and techniques. The ATT&CK Enterprise matrix will be useful in developing the threat model for this research (MITRE, 2020). More than 400 software tools and malware have been modeled by MITRE to capture known malicious techniques (MITRE, 2020). MITRE's NotPetya malware threat model has 13 unique techniques from the Enterprise ATT&CK matrix. Nine example OSS projects, listed in Table 2 below, can demonstrate all but one of the NotPetya techniques.

Technique Name	Example OSS Projects

Steven M. Launius, SteveLaunius@gmail.com

Credential Dumping	https://github.com/gentilkiwi/mimikatz.git
Data Encrypted for Impact	https://github.com/goliate/hidden-tear.git
Exploitation of Remote Services	https://github.com/ElevenPaths/Eternalblue-Doublepulsar-Metasploit.git
Indicator Removal on Host	https://github.com/limbenjamin/Invoke-LogClear.git
Masquerading	https://github.com/malcomvetter/CSExec.git
Rundll32	https://github.com/p3nt4/PowerShdll.git
Scheduled Task	https://github.com/NukelearGhost/SleepWake.git
Service Execution	https://github.com/malcomvetter/CSExec.git
Supply Chain Compromise	N/A
System Shutdown/Reboot	https://github.com/risoflora/system_shutdown.git
Valid Accounts	https://github.com/malcomvetter/CSExec.git
Windows Admin Shares	https://github.com/malcomvetter/CSExec.git
Windows Management Instrumentation	https://github.com/Robinatus/Windows-Network-Spy.git

Table 2. Example OSS Projects for Each NotPetya Technique

The Application Inspector tool analyzed each of the nine OSS projects to identify their feature sets. The characteristics across all projects that represent high-risk threat features are *Authentication.General* and *OS.Process.Dynamic.Execution*. However, a model with only two common functions would identify a large percentage of libraries as high-risk. Therefore, several threat models of the significant techniques modeled from example OSS projects will be more useful. These techniques from NotPetya are Credential Dumping, Data Encryption, and Exploitation of Remote Services. The following is a model of each technique with a unique set of features identified as tag names by the tool's *analyze* command of the corresponding OSS project:

Steven M. Launius, SteveLaunius@gmail.com

Credential Dumping

Mimikatz by gentilkiwi

- Authentication
- Authorization
- Microsoft.DLL
- Cryptography
- Data.DBMS
- Data.Deserialization

- Data.Sensitive.Credentials
- OS.ACL.Impersonation
- OS.FileOperation
- OS.Network.Connection
- OS.Process.DynamicExecution
- OS.SystemRegistry

Data Encryption

hidden-tear by goliate

- Authentication
- Authorization
- Cryptography
- Data.DBMS
- Data.Parsing

- Data.Sensitive.Credentials
- Data.Zipfile
- OS.FileOperation
- OS.Network.Connection
- OS.Process.DynamicExecution

Exploitation of Remote Services

EternalBlue by ElevenPaths

- Authentication
- Authorization
- Cryptography
- Data.DBMS

- Data.Parsing
- Data.Sensitive.Credentials
- Data.Zipfile
- OS.Process.DynamicExecution

Masquerading/Service Execution/Windows Admin Shares

CSExec by malcomvetter

- Authentication
- Authorization
- Microsoft.DLL
- Cryptography
- Data.DBMS

- Data.Parsing
- Data.Sensitive.Credentials
- Data.Zipfile
- OS.FileOperation.Delete
- OS.Network.Connection

Steven M. Launius, SteveLaunius@gmail.com

- OS.Process.DynamicExecution

Note: Tag names have been shortened to a substring that matches similar feature components of the same type, where possible, to group them.

Each of the four OSS projects produces a unique set of features for the NotPetya techniques. Mimikatz has a collection of thirteen features that model the Credential Dumping technique. Hidden-tear has a collection of ten features that model the Ransomware technique. EternalBlue has the smallest set of features, with eight, that model Exploitation of Remote Services. The CSExec project has a collection of eleven features that model Masquerading, Service Execution, and Windows Admin Shares (shown as Masquerading).

4.2. Risk Model Testing

4.2.1. Lab Environment

The same lab configuration, from the Accuracy Testing section above, is used to test the threat model on a selected set of OSS project libraries from GitHub. A sample of 20 OSS projects will represent library components that may be created or reused by developers for the web application from the use case. The selected samples will include various programming languages with an API, framework, or other functionality to aid a developer in completing an application. The complete set of samples can be found in Appendix C.

This test requires modification of the Application Inspector preferences to identifying the threat model feature sets in a custom HTML report. The custom report shows matching features (see Figure 2) as blue icons and non-matching features as grey icons. A sample of the HTML results is shown below in Figure 2.

Steven M. Launius, SteveLaunius@gmail.com
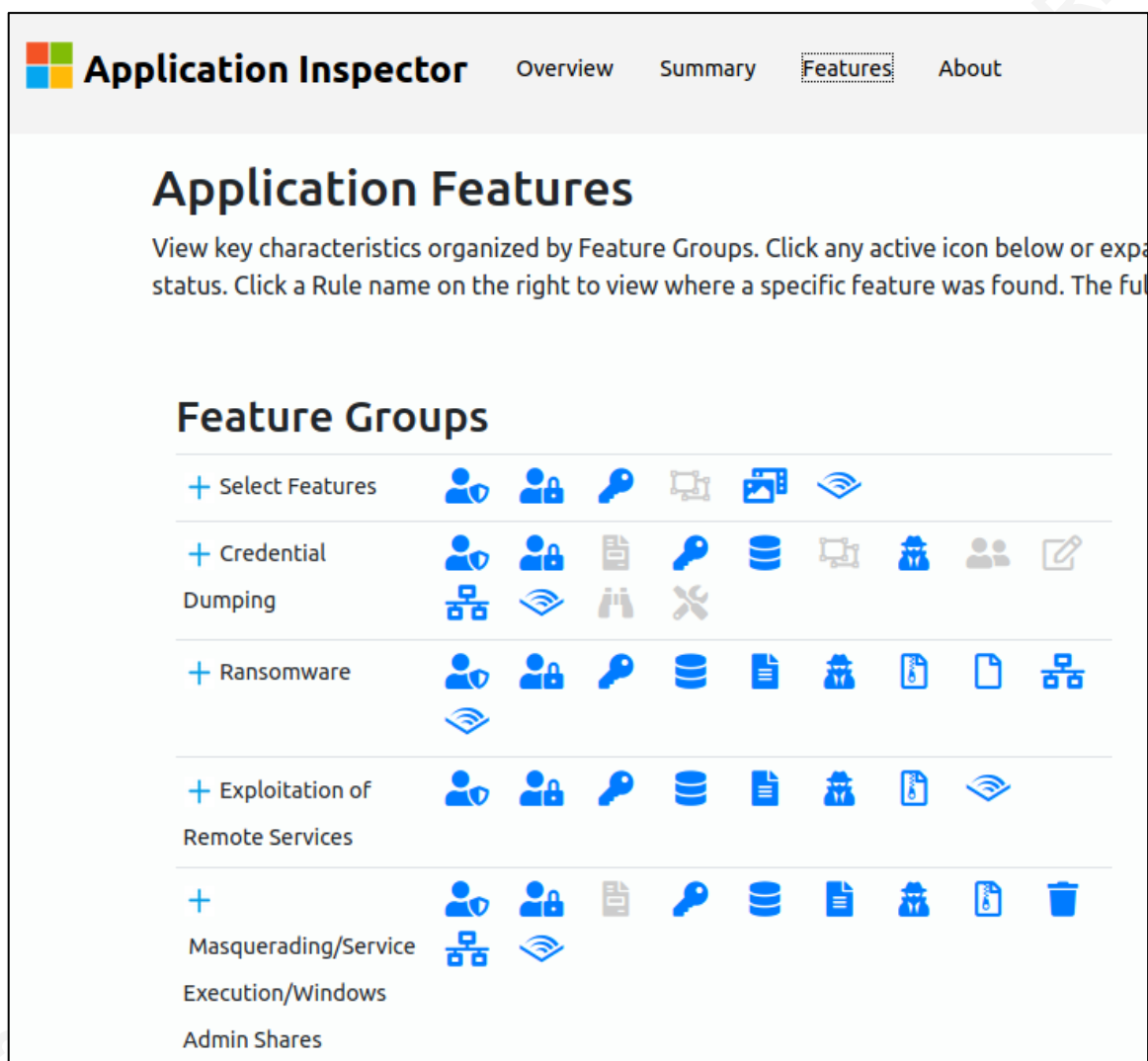
Figure 2. Sample of a Custom Application Inspector HTML Report

4.2.2. Testing Methodology

The number of matching tag names for each threat technique modeled will be analyzed and compared across the OSS projects. The configuration of the Application Inspector analysis options will output HTML and JSON files with results and use single threading to prevent anomalies identified in the Accuracy Testing section above. The following procedures will be followed for each of the OSS projects:

1. Download OSS project from GitHub.

```
git clone <URL to GitHub project file>
```

Steven M. Launius, SteveLaunius@gmail.com

2. Link the 'html' directory from Application Inspector's source to enable the correct display of the HTML report.

```
ln -s ../html/ <OSS project directory name>
```

3. Record the project size.

```
du -h --max-depth==1 ./<OSS project directory name>
```

4. Change directory into the OSS project.

```
cd <OSS project directory name>
```

5. Analyze the OSS library with Application Inspector.

```
dotnet ~/ApplicationInspector_1.2.11/ApplicationInspector.CLI.d
ll analyze -s . --single-threaded
```

6. Manually review the HTML report in Firefox to count the features matching in each technique modeled.

4.3. Results

The test results will identify the error rate of each threat model. Tests showing a 100% error rate indicate all features in a model's set matched, and represents a negative result since each of these projects are assumed not to contain malicious code. The useful models will have few if any, negative results. The size of each OSS project was calculated to correlate any trends in the volume of code analyzed. The percentage for each outcome is calculated as X of Y, where X is the number of matching features and Y is the total number of features in each model's set. The features were manually counted by reviewing the features for each OSS project in a custom HTML report created to display the threat model results. In Figure 3 below, the Credential Dumping results are given as an example.

Steven M. Launius, SteveLaunius@gmail.com

Figure 3. Sample HTML Report of Features in the Credential Dumping Threat Model.

The error rate for each threat models' results, as tested against all OSS projects, is displayed in Table 3 below.

Steven M. Launius, SteveLaunius@gmail.com

OSS Project	Size (MB)	Credential Dumping	Ransom-ware	Exploitation of Remote Services	Masquer-ading
ui_components	4.9	46%	80%	100%	73%
Hazel	11	54%	90%	100%	82%
zphiser	120	62%	90%	100%	82%
tabler	127	54%	80%	100%	73%
thingsboard	134	54%	100%	100%	82%
trilium	96	69%	100%	100%	82%
redash	37	62%	100%	100%	82%
livewire	12	54%	90%	100%	82%
PhpSpreadsheet	46	62%	90%	100%	82%
airflow	108	77%	100%	100%	91%
websockets	2.7	69%	100%	100%	91%
calculator	58	62%	100%	100%	82%
Ooui	6.9	69%	100%	100%	91%
redux.NET	8.6	46%	80%	100%	73%
fastapi	17	54%	90%	100%	82%
sheetjs	75	62%	100%	100%	90%
card	2.1	46%	80%	100%	73%
flask	11	62%	100%	100%	82%
autobahn-python	26	82%	100%	100%	82%
skatejs	15	70%	100%	100%	91%
Total Negative		**0**	**11**	**20**	**0**

Table 3. Error Rate in Risk Modeling Analysis

Steven M. Launius, SteveLaunius@gmail.com

The test results for two of the four threat models are entirely positive. The Credential Dumping and Masquerading models show that none of the 20 tested known-good projects matched all the features in their sets. Indicating these models would be useful to identify source code libraries that include features similar to the malicious software they were modeled after. Ransomware's model shows a little more than half of the projects contain the same feature set as the model. This model would produce many false positives if incorporated into a continuous deployment pipeline that could unnecessarily waste resources. There are no positive results for the Exploitation of Remote Services model, indicating this model would never be useful for identifying high-risk libraries. There was no correlation between the project's size and the percentage of features found in any model.

5. Recommendations and Implications

The Application Inspector is a unique static code analysis tool that can reduce risk in a DevOps environment by identifying features in source code. Feature analysis provides a summary of functionality found within millions of lines of code to quickly and automatically identify high-risk features. Threat modeling of adversaries targeting an organization can produce high-risk feature sets. Organizations employing many developers as contractors and incorporating third-party libraries can decrease the risk of malicious backdoors with this tool.

5.1. Practical Use

The accuracy testing results for Application Inspector's analysis are encouraging, but there is an opportunity for improvement. As a newer tool, Application Inspector has a reasonably high accuracy rate for the Python language. However, this research shows there are standard Python libraries unmatched during analysis that have features the tool is designed to identify. Additional rules to match all of the standard library features for Python, and other programming languages, will increase the tool's accuracy. Organizations can concentrate on creating rules for the programming languages found in

Steven M. Launius, SteveLaunius@gmail.com

their DevOps environment. Sharing these rules with the open source community would be a public service that benefits an organization's reputation in the DevOps community.

The test results from risk modeling indicate that the more features in a model's set, the more likely it is to identify malicious code. The more specific a threat model is for a particular adversary's malware, the larger is the set of features in the model. The test results indicate models with more features perform better and benefit organizations modeling threats targeting them or their industry. Models with more than ten features of Application Inspector's tags are more likely to catch the malicious code. Feature analysis can be a useful automated technical control to reduce the risk of deploying malicious code when incorporated into deployment pipelines or manually run by developers or auditors.

Threat models for feature analysis require testing and risk management. Information security professionals can develop specific threat models targeting an organization and test the error rate. The test results for ambiguous models, like Ransomware, is risk that leadership will decide to accept. Management must decide whether or not a model's resulting error ratio outweighs the likelihood an adversary will use this attack vector on their organization. Automated implementation of the threat model is more practical than performing manual code review.

When personnel resources limit manual code review, the Application Inspector tool provides automated capabilities for managing the risk of the source code features. The tool's JSON reporting lends itself to implementation through scripting in a continuous deployment pipeline. The tool enables DevOps teams to adhere to strict feature requirements from initial development that only permit authorized features to be deployed into a production environment. Additionally, this research presents a risk model for the tool to identify a specific threat technique from a set of features that DevOps teams can implement in a deployment pipeline or code versioning system. Developers or auditors could also run the tool, as necessary, with this risk model to identify specific threats within a set of source code files from a third-party library or other internal developers.

Steven M. Launius, SteveLaunius@gmail.com

6. Conclusion

Adversaries are taking advantage of third-party libraries and DevOps' speed to insert malicious code inside of software that an organization develops. Organizations often trust their third-party suppliers and internal developers to produce software that meet their required features. Therefore, management may choose to accept the risk from backdoors so they can compete in their industry. The Application Inspector tool can provide a practical method for reducing this risk. The tool's accuracy is good but it can be improved with additional rules to detect more standard libraries from the programming language version used by an organization. Additionally, the creation of threat models from adversary techniques can provide a set of high-risk features, if enough exists, that can uniquely identify the same technique in source code libraries. Organizations can implement the tool at scale within continuous deployment pipelines or use ad hoc to vet source code libraries.

Steven M. Launius, SteveLaunius@gmail.com

References

Acosta, G. (2020, April 10). *GitHub - microsoft/ApplicationInspector*. Retrieved April 20, 2020, from GitHub: https://github.com/microsoft/ApplicationInspector

Acosta, G. (2020, March 8). *Home · microsoft/ApplicationInspector Wiki*. Retrieved June 11, 2020, from GitHub: https://github.com/microsoft/ApplicationInspector/wiki

Acosta, G., & Scovetta, M. (2020, January 2020). *Introducing Microsoft Application Inspector*. Retrieved April 19, 2020, from Microsoft: https://www.microsoft.com/security/blog/2020/01/16/introducing-microsoft-application-inspector/

Bennett, J. (2018, October 31). *When Good Software Goes Bad: Malware In Open Source*. Retrieved April 19, 2020, from Hack A Day: https://hackaday.com/2018/10/31/when-good-software-goes-bad-malware-in-open-source/

Bird, J. (2017, October 24). *2017 State of Application Security: Balancing Speed and Risk.* Retrieved April 19, 2020, from SANS Institue: https://www.sans.org/reading-room/whitepapers/application/paper/38100

De Gorge, A. (2020, June 4). *Install .NET Core on Ubuntu - .NET Core*. Retrieved June 12, 2020, from Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/core/install/linux-ubuntu

Elliot, J. K. (2019, July 11). *Disgruntled employee hijacks AHL hockey app to settle an office score*. Retrieved June 9, 2020, from Global News: https://globalnews.ca/news/5484145/ahl-push-notifications-throat-punch/

Flores, B. (2019, June 20). *The looming threat of malicious backdoors in software source code*. Retrieved April 19, 2020, from SC Magazine: https://www.scmagazine.com/home/opinion/executive-insight/the-looming-threat-of-malicious-backdoors-in-software-source-code/

Greenberg, A. (2018, August 22). *The Untold Story of NotPetya, the Most Devastating Cyberattack in History*. Retrieved June 9, 2020, from Wired:

Steven M. Launius, SteveLaunius@gmail.com

https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/

Howard, M. (2018, October 9). *Who Cares if Supermicro Happened. Supply Chain Attacks are Real and It's Time to Pay Attention*. Retrieved April 19, 2020, from Sonatype: https://blog.sonatype.com/is-supermicro-real-who-cares.-supply-chain-attacks-are-happening-and-its-time-to-talk-about-it

McDougall, D. (2014, May 31). *pyminifier - Minify, obfuscate, and compress Python code*. Retrieved June 13, 2020, from GitHub: http://liftoff.github.io/pyminifier/

Migues, S., Steven, J., & Ware, M. (2019, September 18). *Building Security In Maturity Model (BSIMM)*. Retrieved April 19, 2020, from BSIMM: https://www.bsimm.com/download.html

MITRE. (2020, June 10). *ATT&CK*. Retrieved June 16, 2020, from MITRE: https://attack.mitre.org/

MITRE. (2020, June 17). *Software*. Retrieved June 20, 2020, from MITRE ATT&CK: https://attack.mitre.org/software/

Morgan, L. (2018, September 5). *5 ways static code analysis can save you*. Retrieved June 11, 2020, from SD Times: https://sdtimes.com/test/5-ways-static-code-analysis-can-save-you/

Ponemon Institute. (2019). *The State of Software Security in the Financial Services Industry*. Mountain View: Synopsys. Retrieved June 8, 2020, from Synopsys: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/software-security-financial-services-ponemon.pdf

pratz. (2019, September 8). *Decoding the Confusion Matrix*. Retrieved June 12, 2020, from KeyToDataScience: https://keytodatascience.com/confusion-matrix/

Security Current. (2017, November 19). *11 CISOs Say Open Source Software Can Be As or More Secure Than Commercial Software, With a Potential for Savings*. Retrieved April 19, 2020, from Security Current: https://securitycurrent.com/10-

Steven M. Launius, SteveLaunius@gmail.com

cisos-say-open-source-software-can-be-as-or-more-secure-than-commercial-software-with-a-potential-for-cost-savings/

Springett, S. (2020, January 16). *Component Analysis*. Retrieved April 19, 2020, from OWASP: https://owasp.org/www-community/Component_Analysis

Stehman, S. V. (1996, October). Selecting and interpreting measures of thematic classification accuracy. *Remote Sensing of Environment, 62*(1), 77-89. doi:10.1016/S0034-4257(97)00083-7

Tung, L. (2016, March 23). *Disgruntled developer breaks thousands of JavaScript, Node.js apps*. Retrieved June 9, 2020, from ZDNet: https://www.zdnet.com/article/disgruntled-developer-breaks-thousands-of-javascript-node-js-apps/

Steven M. Launius, SteveLaunius@gmail.com

Appendix A

OS.Networking.Connection Rules

The most pertinent attributes for each of the patterns associated with rules that have the OS.Networking.Connection tag name are summarized in the table below.

ID	Tag Name	Applies To	Pattern	Pattern Type	Mod- ifier	Confi- dence
AI031600	OS.Network.Con nection.Miscella neous		tftp|ntp\\.org|ntpupdat e|imap|snmp|ftps|sftp| ftp|nntp|smtp|telnet|ss h|pop3|gopher	regex-word	i	high
AI032000	OS.Network.Con nection.Socket	python	.bind(	string		high
AI032100	OS.Network.Con nection.Socket		socket	string	i	high
AI032500	OS.Network.Con nection.Http	python	import .*(http|requests)	regex		high
AI032600	OS.Network.Con nection.Http		https*:/	regex		low
AI032600	OS.Network.Con nection.Http		curl|wpget	regex-word		medi um
AI032900	OS.Network.Con nection.RPC	python	fastrpc|xmlrpc|SimpleX MLRPCServer|jsonrpc|rp c\\.server|client\\.rpc	regex		high
AI033410	OS.Network.Con nection.General		send.*message	regex		high
AI033410	OS.Network.Con nection.General		send\\(	regex		medi um

Steven M. Launius, SteveLaunius@gmail.com

Appendix B

Custom Script for Accuracy Tests

The following script was created in a Python file to exhibit a false negative for Application Inspector rules.

```python
import urllib.request

req = urllib.request.Request("htt" + "ps://www.google.com/")
result = urllib.request.urlopen(req)
print(result.read())
```

Steven M. Launius, SteveLaunius@gmail.com

Appendix C
Selected OSS Projects for Risk Model Testing

The table below contains OSS projects used in the risk model testing.

OSS Project	Primary Language	GitHub URL
ui_components	JavaScript	https://github.com/bradtraversy/ui_components
Hazel	C++	https://github.com/TheCherno/Hazel
zphiser	PHP	https://github.com/htr-tech/zphisher.git
tabler	Java	https://github.com/tabler/tabler
thingsboard	Java	https://github.com/thingsboard/thingsboard
trilium	JavaScript	https://github.com/zadam/trilium
redash	JavaScript	https://github.com/getredash/redash
livewire	PHP	https://github.com/livewire/livewire
PhpSpreadsheet	PHP	https://github.com/PHPOffice/PhpSpreadsheet
airflow	Python	https://github.com/apache/airflow
websockets	Python	https://github.com/aaugustin/websockets
calculator	C#	https://github.com/microsoft/calculator
Ooui	C#	https://github.com/praeclarum/Ooui
redux.NET	C#	https://github.com/GuillaumeSalles/redux.NET
fastapi	Python	https://github.com/tiangolo/fastapi
sheetjs	JavaScript	https://github.com/SheetJS/sheetjs
card	JavaScript	https://github.com/jessepollak/card
flask	Python	https://github.com/tensorflow/tensorflow
autobahn-python	Python	https://github.com/crossbario/autobahn-python
skatejs	JavaScript	https://github.com/skatejs/skatejs

Steven M. Launius, SteveLaunius@gmail.com