Augmenting Network Flows with User Interface Context to Inform Access Control Decisions

by

Zorigtbaatar Chuluundorj

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

December 2019

APPROVED:

Professor Craig A. Shue, Major Thesis Advisor

Professor Robert J. Walls, Thesis Reader

Professor Craig E. Wills, Head of Department

Acknowledgements

I want to express my sincere gratitude to my advisor Professor Craig. A Shue, who has supported me throughout my graduate education with skillful guidance and ample patience. He continuously conveyed the importance of persistence in regards to research and the scientific method. The trust he placed in my abilities provided the motivation needed to further my studies and not to give up. Without his help, this thesis would not have been possible. I would also like to express appreciation to my reader professor Robert J. Walls for his feedback on my thesis, and the insightful questions asked during presentations. His assistance has been vital to the improvements of the thesis.

I would like to thank the professors and graduate students of the Applied Logic and Security (ALAS) research group for their feedback and support in my development as a member of the scientific community. The discussion with my peers and their willingness to share their thoughts and knowledge regarding their area of expertise broadened my understanding of the diverse fields of computer science. The opportunities to present my thesis to the members of ALAS improved my public speaking capabilities. Furthermore, the social environment fostered in the ALAS laboratories helped me overcome the difficulties and the eventual stress resulting from the pursuit of higher education. Without the members of the ALAS research group, my graduate experience would have been less enjoyable and fruitful. I would like to extend my gratitude to the staff at the Computer Science department at Worcester Polytechnic Institute. Their efforts made the nonresearch parts of my education easy.

Last, I would like to thank my parents Chuluundorj Begz and Altanger Balgan, for their trust and support throughout my secondary and undergraduate education and for raising me. They are the reason why I was able to pursue my graduate education.

Abstract

Whitelisting IP addresses and hostnames allow organizations to employ a default-deny approach to network traffic. Organizations employing a default-deny approach can stop many malicious threats, even including zero-day attacks, because it only allows explicitly stated legitimate activities. However, creating a comprehensive whitelist for a default-deny approach is difficult due to user-supplied destinations that can only be known at the time of usage. Whitelists, therefore, interfere with user experience by denying network traffic to user-supplied legitimate destinations. In this thesis, we focus on creating dynamic whitelists that are capable of allowing user-supplied network activity. We designed and built a system called Harbinger, which leverages user interface activity to provide contextual information in which network activity took place. We built Harbinger for Microsoft Windows operating systems and have tested its usability and effectiveness on four popular Microsoft applications. We find that Harbinger can reduce false positives-positive detection rates from 44%-54% to 0%-0.4% in IP and DNS whitelists. Furthermore, while traditional whitelists failed to detect propagation attacks, Harbinger detected the same attacks 96% of the time. We find that our system only introduced six milliseconds of delay or less for 96% of network activity.

Contents

1 Introduction **6**

2 Related Work **8**

2.1 Intersection of GUI and Security 8

2.2 GUI Quality Assurance 9

2.3 Intrusion Detection Systems 9

2.4 Default Deny Policy 10

2.5 JavaScript and Web Requests 11

3 Approach **11**

3.1 Threat Model 12

3.2 UI Workflows as a Precursor to Network Activity 14

3.3 Design of Harbinger 14

3.4 UI Workflows and Network Activity 15

3.5 Monitoring the Graphical User Interface 17

3.6 Policy Generation 20

3.7 Data Generation 21

3.8 Browser 22

4 Implementation **23**

4.1 Application monitor 23

4.1.1 UI context 23

4.1.2 UI Activity 24

4.1.3 Inner Workings of the Application Monitor 25

4.1.4 Validity of UI Activity and Context 26

4.1.5 Bundling of processes and services 27

4.1.6 Browser Instrumentation 27

4.1.7 Compression 28

4.2 Network Monitor 29

4.3 Policy Engine 30

5 Experiment environment **31**

5.1 Application Under Test 31

5.2 Automatic generation of UI activity 31

6 Security Evaluation **32**

6.1 Policy Classes . . . 33

6.2 Policy generation . . . 33

6.3 Experimental Setup . . . 34

6.4 Experiment 1: Deterministic Legitimate Destinations with Command and Control Server . . . 36

6.5 Experiment 2: Deterministic Legitimate Destinations with Propagation Attack . . . 36

6.6 Experiment 3: User-Supplied Legitimate Destinations with Malicious Command and Control Server . . . 37

6.7 Security Evaluation Results . . . 38

6.8 Direct Attacks on Harbinger . . . 38

7 Performance Evaluation **38**

7.1 Network Monitor and Policy Engine . . . 38

7.2 Application Monitor . . . 40

7.3 Performance Evaluation Results . . . 42

8 Discussion and Limitations **42**

8.1 Generalization and Limitations . . . 43

8.2 Inference . . . 43

9 Conclusion **43**

1 Introduction

Cybercriminals make a profit by selling personal information on the Dark Web or using ransomware to hold computing resources hostage until the victim pays the ransom. As a result, enterprises are lucrative targets for malicious individuals because of their storage of a large amount of customer information and their ability to pay ransom with liquid capital [27]. To protect against such threats, enterprises use intrusion detection systems (IDS) and anti-virus software. Nonetheless, there are still monthly cyber-attacks on enterprise networks that affect the lives of millions of people. For example, data breaches of Equifax, First America, and Capital One each delivered more than 100 million people's personal information to the hands of malicious individuals [21, 5]. Furthermore, the data breaches combined with the fact that the average dwell time, the duration of time a malware has undetected network access until removed, was 191 days in 2017 [11], the adversary has ample opportunity for corporate espionage. Despite enterprises using IDS and anti-virus software and hiring proficient information technology (IT) staff, there is room for improvement in both detection and prevention of cyber-attacks. In this thesis, we describe the design and construction of an IDS tool, called *Harbinger*, which aims to improve the current detection rate of cyberattacks by using the User Interface (UI) context associated with network traffic and offer more information for security decisions about network connection.

The end-user's predominant form of communicating their intentions to an application is through interactions with a Graphical User Interface (GUI); as a result, we can conclude that GUI interactions act as a precursor to user-driven process and network activity and provide a rich source of contextual information. For example, a network connection from Microsoft Word to a printer would have a precursor UI context of the "Print" button. Harbinger monitors the GUI activities of the end-user and uses this information in conjunction with traditional network knowledge, such as IP address, port, and protocol to detect malicious network activity. We developed a GUI monitor which tracks the user's interaction with the GUI, such as which buttons were clicked, which GUI widgets are currently in focus, mouse clicks, keystrokes, and the associated application. Harbinger takes a fine-grained approach and is able monitor the behaviors of each application with its network activity. Furthermore, using the observed behavior, which takes UI information into account, we can restrict network connections to the intranet and the Internet.

In our previously mentioned "Print" example, the UI context "Print" is a prerequisite for network connections to the printer. As a result, if a malware inside the enterprise attempts to connect to the printer, it will fail unless the user presses the "Print" button at the same time. On the other hand, a traditional IDS will likely allow the network activity because the source and destination are trusted. We can say the same for network activity to the Internet. Harbinger is also be able to predict network connections previously not seen. For example, if the end-user clicks on a hyperlink www.example.com in Microsoft Word, a traditional IDS may flag the connection as an anomaly, but Harbinger will not because it will detect that the user has clicked on a GUI element with class type "link" and name "www.example.com."

Harbinger records the GUI activity of applications for IT staff to consult in case of a breach. This

information can also be more useful than simply having traditionally logged network activities. Traditional log activity only includes network information such as protocol and port numbers, and an example of this would be a Wireshark output. On the other hand, the log file Harbinger produces includes UI information in addition to the traditional network information. In case of a breach, the IT staff, with access to UI information, will be able to determine which application the user was interacting with and what interaction took place.

Current enterprise network security tools such as firewalls and IDS rely on signature-based or anomaly-based detection methods to inspect packet headers and payload to detect malicious network activity. Signature-based IDS looks for specific pre-defined patterns, such as byte sequences, in network traffic to detect attacks. In other words, the approach searches for malicious event spaces. Although a signature-based detection approach can detect known attacks, it cannot detect unseen attacks, for which there are no pre-defined signatures. Furthermore, with obfuscation techniques, such as payload encryption, the adversary can change the signature of their attacks. Therefore, this approach does not easily adapt to new threats and intelligent adversaries. Anomaly-based IDS monitors network activity and classifies the network behavior as either normal or anomalous, which can be thought of as searching for unmalicious event space. In order to detect anomalous or malicious behavior, the IDS first learns to recognize normal behavior, and most IDS employs machine learning for this purpose. Anomaly-based IDS holds an advantage over signature-based IDS because of its ability to detect new attacks for which there are no pre-defined signatures. However, the anomaly-based approach suffers from false positives because it might mistake previously unknown legitimate network traffic as malicious. Furthermore, both approaches are insufficient to stop mimicry attacks in which the adversary attempts to mask malicious traffic as benign by trying to mimic normal network activity. In the event of a breach, the forensic information from current tools is insufficient for understanding the attack; we can see this from the number of major data breaches and the average dwell time, a duration of undetected access an adversary has in a network. As a result, in this thesis, we describe design and construction of Harbinger which aims to overcome these problems by using UI information.

Previous research related to the intersection of security and GUI activity has employed course-grained approaches such as last time of use of the UI [13]. These are course-grained approaches, and an adversary can time their attacks to coincide with UI activity. Approaches which rely solely on traditional knowledge of network activities are also insufficient. These tools can be overcome when an adversary mimics legitimate network activity, hijacks benign software, or obfuscates payload by encrypting it. Furthermore, such tools falsely flag benign activity if it has not been seen before or appears to be an anomaly.

In this thesis, our research questions are: 1) *To what extent can the UI activity be used to predict network activity?* and 2) *To what extent can this information be used to improve network resource requests?* We passively monitor and acquire fine-grained UI contextual information surrounding a network request with traditional network information to find out if UI information improves network security analysis. To test our theory, we built an intrusion detection system which takes advantage of the UI and compared its true

positive and true negative rates to a traditional firewall.

2 Related Work

We group the related work into four categories: intersection of GUI and security, GUI quality assurance, intrusion detection systems (IDS), and JavaScript and web requests. We are building an IDS that leverages UI information, hence an understanding of IDS systems and research on the intersection of GUI and security is essential. Research concerning GUI quality assurance and JavaScript are not directly related to the topic of this thesis, but they give useful background. Research related to GUI quality assurance can provide insightful information regarding comprehensive and accurate monitoring of UI and research on JavaScript can provide information on instrumenting browsers or give a general background information concerning the topic. We require such information because browsers make automatic and irregular network requests without UI interaction, unlike regular desktop applications.

2.1 Intersection of GUI and Security

Previous research in the field focusing on the intersection between GUIs and security have suggested using mouse click and keyboard press counts to differentiate between users, using low system activity to infer higher-level GUI activity, or requiring the customization of existing GUI functionality to monitor the end-user. Binder [7] and Kwon [13] stated that malware operates in the background without interaction with the UI; therefore, they tried to correlate user activities with the network activity immediately succeeding it. However, they did not tie specific network destinations to UI activity or an application; as a result, their approach is susceptible to mimicry attacks [35]. HoNe [9] correlated a process with its network activity but did not consider UI activity preferring to focus on the process' metadata solely. These approaches have limited or no visibility into user activity of an application, thus making them coarse-grained. Harbinger overcomes these deficiencies by linking specific UI activity of an application to a specific network destination and limits the opportunities for an adversary to launch mimicry attacks. Bhukyaet *et al.* [4] studied mouse and keyboard usage to differentiate between users on a machine. They monitored mouse clicks and keyboard presses and used machine learning to build per-user profiles. They then applied the machine learning model to detect malicious individual attempting to masquerade as a company employee on a company machine. Even though they focused on insider threats, which is not in our threat model, Harbinger leverage user input because of its utility.

UI activity has also been applied to access control of system resources and file management at the end host. Polaris [34] tracked user activity to infer if a user wants software to access system resources such as the camera and the network card. They achieved this by replacing an application's GUI widgets with custom widgets and monitoring the user activity using them. Shirley *et al.* [31] monitored whether a software tries to gain access to files it has not created or does not have a history of accessing to detect malware. The approach

requires persistent storage of an application's access history of files since its installation for all application is on the system. These approaches take advantage of the UI to infer user intent concerning access control of system resources. However, they require the instrumentation or the replacement of existing functionality, and the latter requires the indefinite maintenance of an application's access history of files.

Lanzi *et al.* [14] did not directly monitor the user activity to infer user intention but inspected system logs and system calls to infer user actions, such as opening an application. The approach not only adds another layer of abstraction between user intention and the security decision but also has difficulty dealing with interleaved system log and calls and incurs high-overhead monitoring system activity. Other research has focused on matching UI input to network packets. Gyrus [12] used UI automation [18] to acquire user input and matches those input with outgoing network packets to make sure the packets were unmodified. However, Gyrus requires a VM host to implement secure overlays and assumes the packets are unencrypted. These approaches are course-grained or require the instrumentation of existing functionality; as a result, there currently are no approaches that can perform comprehensive GUI monitoring without significant instrumentation of existing functionality of the GUI, which would hinder enterprise deployment.

2.2 GUI Quality Assurance

While not directly related to the thesis, research related to GUI testing gives insightful information regarding the comprehensive and concise monitoring of GUI elements, activity, and the structure of the UI interface. Research in this field has focused on the automatic generation of test cases for quality assurance purposes. GUIRipper [16] used low-level function calls to reverse engineer the GUI structure and elements. Once GUIRipper discovers an executable widget, GUIRipper executes the widget and saves the invoked window for later, more in-depth exploration into the GUI. ReGUI [19] uses UI automation to acquire the structure and elements of the GUI and uses guided exploration to explore the structure of the GUI fully. GUISurfer [33] can generate the structural information regarding the GUI interface but requires the source code of the application. Unlike the previous approaches, Sikuli [6] is not able to automatically explore the GUI structure. Instead, it employs image recognition and a Python-based scripting language to enable the rapid prototyping of GUI test cases. These approaches are unable to explore the changes in the GUI structure fully. Many are resource-intensive or require access to the source code. This shows that full exploration of the UI for security purposes will be challenging. However, passive acquisition of UI information while in use may be possible.

2.3 Intrusion Detection Systems

IDSes monitor network or application behavior for malicious activity or policy violation and they are classified either according to their range in scope or detection mechanisms. IDSes systems can range in scope from a single end-user system to large networks. Host-based IDSes (HIDS) are located at the end-user's system and is an example of an IDS monitoring a single end-user's machine for malware. Network IDSes (NIDS) are located at the perimeters of an enterprise network and monitors network traffic for malicious activity.

Depending on the IDS's location in the computing environment, they can have an in-depth view of the end user's machine or a holistic view of the entire computing environment. IDSes also can be divided according to their detection approach, and the common variants are signature-based detection and anomaly-based detection. In this section, we explore research related to IDS systems.

Bro [26] and Snort [29] are NIDSs which filter streams of network traffic into a series of events and enforces policy scripts written in Bro policy language and Berkeley Packet Filtering commands, respectively. Bro and Snort are both examples of NIDSs located at the end-host system that employ signature-based detection mechanisms. Bro focuses more on building an NIDS system capable of handling large volumes of network traffic with real-time notification and puts particular emphasis on the importance of separating the policy engine from the rest of the NIDS. Bro states the separation results in smooth implementation and extensibility of network policy. Snort stated that the slow release of new attack signatures by security vendors creates a security risk. They aim to shore up the security risk by analyzing suspicious traffic at the application layer and create policy rules instead of signatures (or hash values) to mitigate the risk. Harbinger separates the policy engine and enforcement from the rest of the NIDS and uses a rule-based approach instead of hash values as signatures. Lee *et al.* [15] constructed a NIDS similar to Bro, but it employed machine learning algorithms to detect anomalies in network traffic. Yeung *et al.* [37] built a host-based IDS which employed dynamic and static machine learning approaches to detect anomalies on the end host by observing system calls and shell commands. While Harbinger does not take advantage of machine learning to build an anomaly-detection based IDS, we consider it as part of our future work. Garfinkel *et al.* [10] built a hybrid IDS isolated from the end host using virtualization. Garfinkel discussed the trade of between HIDS and NIDS. Garfinkel stated that HIDS and NIDS trades visibility for security, with HIDS having a better view than NIDS because it resided at the end-user's machine while being more susceptible to attacks by malware on that machine. To combat this, Garfinkel used virtualization to isolate the HIDS from the rest of the host system and claims that such a measure makes the HIDS more secure without severely hampering visibility. Harbinger does not use virtualization to isolate itself from the host system for security, but it does take the same precautions as other commercial security tools.

2.4 Default Deny Policy

In this thesis, we seek to move from default-allow to default-deny NDIS policy approach. In the default-deny approach, security tools deny program execution or network activity if they are not explicitly permitted. They commonly implement this approach using a whitelist, which contains a list of permitted applications or network behaviors. Whitelists have been used to deny malware execution on the end-user's system both in academic research and industrial use [11, 18]. While whitelisting program execution is different from whitelisting network behavior, research related to program whitelisting sheds light on what difficulties we may encounter while developing Harbinger.

Application whitelisting involves building an index of legitimate software that is permitted to be present

and active on the end-user's system. The purpose of the application whitelist is to protect the end-user's system from malware infection. The difficulty in creating whitelists lie in making a comprehensive whitelist and updating it regularly to keep up with changes in user behavior. Codeshield introduced Personalized Application Whitelisting (PAW) to overcome the difficulty in using a one-size-fits-all whitelist approach. PAW creates custom whitelist of safe executables and only allows them to run. They claimed this approach overcomes the inflexibility of a one-size-fits-all approach without requiring a security personal to configure the whitelist. The problem with executable whitelists is that keeping the whitelists up-to-date is difficult because the executable's hash value can drastically change with a patch. In network whitelisting, this is not a major issue because network activity should not change drastically without an update. However, network whitelisting presents its challenges in the form of user-specified destinations that cannot be known prior. We focus on solving this problem in this thesis.

2.5 JavaScript and Web Requests

While traditional application's network behavior tends to be deterministic and predictable, a browser's network behavior is erratic and nondeterministic. Web pages contain hundreds of links, interconnectedness to other websites, and JavaScript changes to the document object model (DOM). As a result, their behavior is more complex than desktop applications, and changes are frequent. We, therefore, explore research papers related to the dynamic exploration of JavaScript for quality assurance.

Research in this field focuses on monitoring the event space of a webpage, which is equivalent to monitoring for changes in the DOM. The DOM represents the webpage as a logical tree, which includes the hyperlinks. Therefore, monitoring for DOM changes alerts us to hyperlink changes. Artemis [2] is a JavaScript test case generator which uses a feedback-directed random test generation prioritizing code coverage through interactions with registered JavaScript events. Kudzu [30], similar to Artemis, combines random test case generation to explore JavaScripts event space with a model for reasoning about JavaScript string values and operations. Crawljax [17] uses a similar approach to explore the JavaScript event space to crawl dynamically created links. However, it is still not known whether Artemis and Kudzu can explore JavaScript event spaces fully. Crawljax further requires some invokable events to be user-specified.

3 Approach

Based on our observation, computer applications tend to have deterministic network behavior when there is no interaction from the user or if the user is interacting with the application in a predetermined manner. They establish connections to predictable destinations, which can be observed during application startup or periodic checks for updates if the network activity resulted from the background activity of an application. This is also true when the user is interacting with an application in a predetermined manner, meaning that the user does not provide input that might affect the network destination. These destinations are hard-

coded or are in a configuration file and, thus, deterministic. However, when users start providing network destinations, such as inputting URLs, nondeterministic network activity is introduced. Creating an effective whitelist is then difficult because such network activities are not in the baseline application behavior and have not been observed before. Whitelists, therefore, start interfering with user experience and legitimate network traffic by blocking user-specified network activity.

Prior work on whitelists have sought to integrate user interactions when making security decisions [7, 31, 13] to tackle the above problem. By understanding what the user intends to do, they sought to whitelist only those activities which the user authorized. However, malicious agents have become more intelligent, employing sophisticated mimicry attacks [35] blending in with legitimate activity. Due to the coarse-grained approach of previous works of allowing network activity immediately following user activity, they give malicious agents a chance to blend in.

In this section, we explain our threat model, the challenges we faced, and the design decisions made to overcome those difficulties to make effective whitelists. In our endeavor, we have leveraged UI activity to create dynamic whitelists that take into account both an application's baseline and user-generated network behavior. Our approach is more fine-grained than previous works able to monitor the user's UI activity precisely.

3.1 Threat Model

We assume that the adversary has already exploited a vulnerability in the host system and has user-level privileges only. We made the previous assumption because the first step of exploitation is finding a vulnerability and achieving a foothold inside the enterprise network, regardless of the end goal. As a result, we do not concern ourselves with the mode of exploitation for the initial phase of an attack, as the methods available to the adversary are numerous. For example, freely available exploitation tools, unpatched vulnerable software installed on the computing base, and zero-day exploits are among many possible avenues of attack. As a result, our threat model assumes a compromise will occur despite the best efforts by the defenders.s.

The assumption that the adversary only has user-level privileges is also in line with adversaries' real-world behavior [3] and the principle of least privilege (POLP) [8]. Once adversaries are inside the system, they usually have user-level privileges only at the beginning and work their way up to acquire administrative privileges with advanced techniques or further propagate within the enterprise network. To gain administrative privileges, adversaries usually download more capable malicious software from outside the enterprise. Such an attempt would be detected by Harbinger, as explained in sections below.

With user-level privileges, the adversary can perform any actions permissible to a user-level process, such as making arbitrary number of connections to any destination, starting and stopping other user-level processes, reading and writing to files, and anything else that does not require administrative privileges. The adversary will likely use these capabilities to make arbitrary outbound connections in the hopes of further propagating within the local area network (organization) or to an outside command and control center to

receive further instructions or download a more capable malware because being dormant inside the exploited machine is not conducive to profitable outcomes such as stealing data from the exploited machine. Since the adversary only starts with user-level privileges, he or she will be barred from activities that can tamper with administrative level processes, files, and the kernel; as a result, we consider the administrative account and the kernel to be in the Trusted Computing Base (TCB). This limitation of the adversary is not only realistic, but in line with the assumptions made by host-based firewalls and anti-virus software because these defense systems operate in the administrative level and can be terminated if the adversary acquires administrative privileges.

The adversary may be aware of our intrusion detection system and its inner workings, including the libraries and windows services Harbinger relies on, how information is collected, how the collected information is used to make security decisions, which assumption Harbinger makes, and the privileges space Harbinger runs with. This information is useful to the adversary because Harbinger runs with administrative privileges; therefore, the adversary cannot directly stop Harbinger from running and would need an indirect method to overcome the IDS we have developed.

Such attempts can be in the form of feeding false information to our system by using software-generated UI activity or tampering with libraries Harbinger relies upon. Generating UI activity with software is possible due to how mouse and keyboard activity are handled in the Windows operating system. Windows applications are event-driven because they do not make explicit function calls to the operating system to acquire user input, but wait for the operating system to notify them of user input in the form of a Windows Message submitted to the application's message queue. Furthermore, Windows Messages designating user activity can be created not only by the operating system but by other applications as well. As a result, in our threat model, the adversary could easily mimic user input with software. We rely on a framework provided by the Microsoft Corporation to retrieve UI information of the applications under test. The framework loads DLLs into processes that want to access UI information, and these DLLs perform UI information acquisition and transmission. The DLLs associated with the framework may be replaced or modified by the adversary. We have taken precautions against such threats, through the usage of drivers and DLL checkers, and in the case, our approach is not enough, previous research done by trusted computing and virtualization communities have explored methods to place UI widgets inside the TCB to protect from malicious applications tampering with the GUI.

The goal of Harbinger is to move from a blacklist, default-allow policy, to a usable whitelist, a default-deny policy. When blacklists are used, all connections are allowed except for the ones that are in the blacklist. These blacklists contain the IP addresses or host names of destinations that have previously exhibited malicious behavior. This approach can be unreliable when the adversary uses a clean IP address or host name. This situation is not uncommon as the adversary can buy new IP addresses or hostnames, or commandeer an unwilling third party's machine for malicious usage. Due to these drawbacks, we investigate the feasibility of whitelist-based default-deny policy. Our system only allows a connection if the connection

was observed as baseline behavior (traditional whitelist) or the network connection was caused by GUI activity (e.g., clicking on a URL). We assume the defender is working in an enterprise setting with the capability to install Harbinger on all machines under care. Our approach can be used in conjunction with other tools such as anti-virus software, anomaly detection software, and blacklists. We assume all user actions are legitimate and performed with nonmalicious intent in mind. As a result, we do not protect against social engineering, phishing, and insider threats, as these attacks require user participation.

3.2 UI Workflows as a Precursor to Network Activity

User interfaces facilitate the interaction between the user and the machine. They are where user intent turns into machine action, whether that be accessing a file, running a program, or generating a network activity. End users have the choice of interacting with a machine by choosing from multiple user interfaces, including, but not limited to, the command-line user interface, voice user interface, and graphical user interface. From the list of possible user interfaces, the graphical user interface is the most popular, and all major operating systems provide one.

The machine action resulting from interactions with the GUI occurs temporally later than their corresponding UI activity. We believe monitoring the UI will provide the necessary contextual information for inferring the legitimacy of the resulting network activity, allowing us to move towards a whitelist-based intrusion detection system. For example, when the end-user clicks on the print button in Microsoft Word, the application makes network connections to the printer. Clicking on the print button, a GUI action, led to the application making a connection to the printer, a network activity. Monitoring the graphical user interface, therefore, can act as a precursor to network activity. The purpose of Harbinger is to collect and associate such data.

3.3 Design of Harbinger

In order to associate GUI and network activity, Harbinger must monitor GUI activity and intercept network traffic and find a way to associate them. Due to the multiple responsibilities' of Harbinger, we have divided it into three major components: the application monitor, the controller, and the policy engine as shown in Figure 1. They are responsible for monitoring user activity, approving or denying connections, and generating policy, respectively. The application monitor resides in the client machine to monitor and save user activity, and upon request, transmit the data to the controller. The controller resides on a remote server, and it is responsible for approving or denying a connection based on policies saved in a database with information provided by the application monitor. The policy generation is currently conducted offline with the usage of a Python script and training data.

Figure 1 also shows an example scenario of Microsoft Word printing a document. Without Harbinger, Microsoft Word will immediately process the user input and send a request over the network to the printer. With Harbinger, we intercept network requests (lines 3 and 4 in Figure 1) with the Network Monitor and

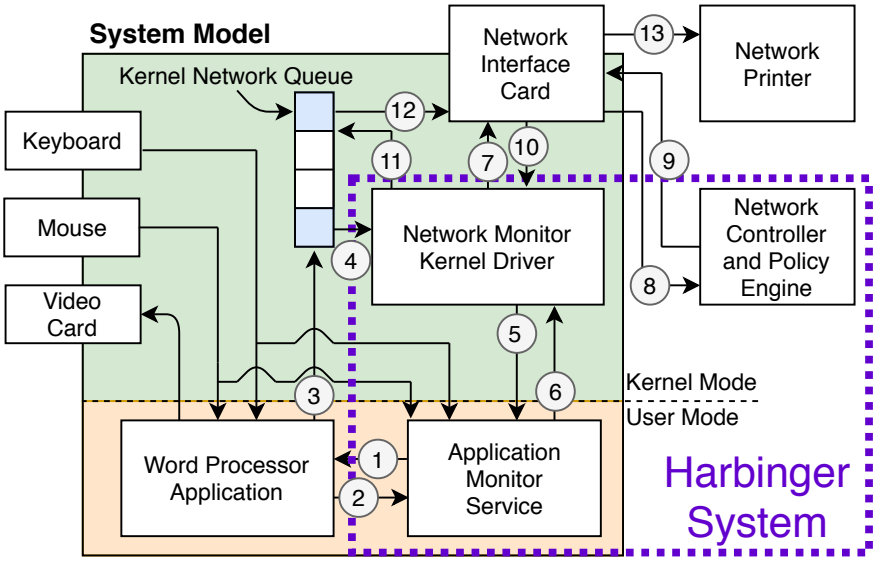

Figure 1: By default the application communicates directly with the destination. However, the outbound communication is intercepted by the centralized access controller and the UI context is retrieved from the UI Monitor. The centralized access controller allow the connection if a matching policy is found.

query the Application Monitor for the associated GUI data (lines 1 and 2 in Figure 1). The Network monitor then packages the network and GUI data and sends it over to the Policy Engine for a security decision (lines 5 and 6 in Figure 1). While waiting for a decision, the packets are queued by the Network Monitor. If the request is approved (lines 9 and 10 in Figure 1), all queued packets are re-injected for transmission (lines 11, 12, and 13 in Figure 1). We explain the implementation of each component in detail in Section 4.

3.4 UI Workflows and Network Activity

Contextual UI information is useful when they are unique, fine-grained, because it allows our system to accurately monitor and differentiate a user's actions when interacting with an application. However, if GUI contextual information is not sufficiently fine-grained, there is room for confusion and error. For example, if our system cannot differentiate between a "Print" button, which causes a network activity, and a document named "Print," which does not cause a network activity, the lack of differentiation is a possible security hole the adversary can take advantage of. The adversary may time their malicious activity shortly after the end user clicks on a document named "Print." In such a situation, if there is no way to differentiate between the "Print" button and the document named "Print," we cannot make an informed decision. As a result, the UI information we collect must be sufficiently fine-grained to differentiate between them. Furthermore, our collection mechanism must not only be fine-grained, but extensive in its capability to acquire UI information from a variety of applications. Besides the point that UI-aware system would need to be able to collect UI information from the various applications the end-user interacts with, the extensive collection of UI

```
Date: Sept. 10, 2018 at 9:54:17am
Source: [ANONYMIZED IP], port 49795, Host Name: [ANONYMIZED]
Destination: 93.184.216.34, port 443, Host Name: example.com
Protocol: TCP [SYN flag set]
Application: C:\Program Files (x86)\Microsoft Office\root\Office16\WINWORD.EXE
User: [ANONYMIZED_DOMAIN]\[ANONYMIZED USER]
Keystrokes (5 s, 15 s, 60 s, 3 m, 5 m): 0, 34, 34, 34, 34
Mouse Click (5 s, 15 s, 60 s, 3 m, 5 m): 5, 5, 6, 8, 8

User Interface (231ms ago):
Name: Document1 - Word, Class: OpusApp, Control: window
| Name: [none], Class: _WwF, Control: pane
| | Name: Document1, Class: _WwB, Control: pane
| | | Name: Document1, Class: _WwG, Control: document
| | | | Name: Page 1, Control: page
| | | | | Name: Page 1 content, Control: edit
| | | | | | Name: https://example.com, Control: hyperlink
```

Figure 2: Example trace from Harbinger while using the Microsoft Word application. Keyboard and moue activity are binned in cumulative time buckets of varying size. The GUI data is shown in a tree format from the root of the application to the actuated widget and with relative time since actuation. The GUI data also shows the class and control type of the GUI widgets.

information further increases the ability to differentiate between UI activity. For example, there is no easy way to differentiate between the "Print" button in Microsoft Word and the "Print" button in Microsoft Excel because the UI element in both cases is nearly identical. This can be overcome not only if we perform the GUI collection on the activated "Print" button, but also extend the collection to the parent GUI element and to its ancestors. We can then differentiate the print buttons as belonging to different applications, one to Microsoft Word and the other to Microsoft Excel.

To represent the relation between GUI and network activity, we define a trace: the union of UI workflow, user input, and the resulting network activity. For example, $\{\texttt{Application}_1\texttt{, (a, b, c, d), inputs}\} \rightarrow \{\ \texttt{connection}_1\texttt{:time}_1\texttt{, connection}_2\texttt{:time}_2\ \}$. In this case, Application could be Microsoft Word; a, b, c, d are GUI workflows, each composed of multiple widgets; and connection: time1 are the IP address and the time in which the connection was made respectively.

Our system produced a more expressive example when the user typed https://example.com and clicked on it in Microsoft Word. This example presents a UI activity and the associated network connection. Figure 2 shows the trace the activity of the user in terms of keystrokes, mouseclicks, GUI widget information, GUI

hierarchy, the application's executable path, the current logged in user, and traditional network information, including the IP addresses, host names, ports, and protocols. The trace was produced by Microsoft Word, when the end-user typed 34 characters and clicked 8 times in the last 5 minutes. The keyboard and mouse activity included typing https://example.com in the Word document body (automatically converted to a hyperlink) and clicking on the hyperlink, which resulted in network activity to https://example.com. The UI activity occurred 241 milliseconds before the associated network activity.

The trace formalism provides inherent support for a dynamic whitelist based approach. First, the trace formalism supports traditional whitelist policy, such as activities based on hardcoded destinations of the application (e.g., application to updated server) or to enterprise-configured destinations (e.g., email servers). In such cases, we would ignore the UI and user inputs fields because they do not affect the network activity. For such policies, we would provide the destinations as IP address or DNS host names. Second, the trace formalism supports variability in network behavior. For example, due to DNS caching, which is when an application saves DNS lookups temporarily for future use, the number of connections to a destination may vary. When an application first visits a remote destination, the application performs a DNS lookup and caches it, but on the second visit, the DNS lookup is skipped because it is in cache (temporary memory). As a result, the number of network connections differs. Computation load balancing also causes the temporal distance between a UI action and associated network activity to vary. For example, consider a machine only running the Chrome browser versus a machine concurrently running the Chrome browser and Microsoft Word. In the former has more computational resources are available for Chrome. As a result, UI interactions and the associated network activity will occur temporally closer in the first case than the second one. These variabilities can be accommodated with trace counters, which count both the number of consecutive traces to the same destinations and time bounds. Last, the trace can be self-referential, which is essential for dynamic whitelists. If the end-user initiates a connection to a destination by accessing a URL or IP address, the URL or IP address will appear in the UI field of the trace. We can dynamically whitelist a previously unseen destination if the host name or the IP address appears in the associated UI field.

Figure 3 illustrates the policy that the UI field must contain the destination host name or IP address as a substring to allow the connection. This policy would even allow any destination accessed by a hyperlink to be whitelisted even without knowing the destination in advance.

3.5 Monitoring the Graphical User Interface

Using GUI activity to predict low-level system activity (such as network activity) requires complete and comprehensive monitoring of the user's interaction with the GUI. This must occur in a timely manner, regardless of the application the user is interacting with or the operating system running on the machine. To achieve this goal, there are three possible approaches: instrumentation of each application on each operating system, the instrumentation of GUI libraries, or a specially designed library supported on most operating systems. Furthermore, the user's interaction with the UI must be obtained before the application processes

```
Destination IP: *,
Destination Host: $1,
Destination Port: 443,
Protocol: TCP [SYN flag],
Application Path: C:\Program Files (x86)\
Microsoft Office\root\Office16\WINWORD.EXE,
Mouse Clicks/Keystrokes: 1+ in last 5 seconds,
GUI: .*https://([a-zA-Z0-9\-]+)/\s*, Control: hyperlink.*,

Action: ALLOW
```

Figure 3: Example policy rule to allow the traffic matching the trace in Figure 2. We use regular expressions to handcraft a policy which encompasses hyperlink traffic.

the request, meaning that Harbinger would need to intercept requests before the user's action reaches the application. If we do not intercept the user's action, we would encounter a race condition in which the application makes the network request before Harbinger has the chance to acquire the associated GUI data.

Instrumentation refers to the action of adding additional code for the purpose of monitoring an application for performance measurements, diagnosing errors, and producing behavioral logs. In our case, we instrument the application under test to log the user interaction with the application's GUI. The first approach, instrumenting each application on each operating system to retrieve the GUI information of the application, would require a significant amount of work due to the number of applications used by enterprises. For example, we would need to instrument Microsoft Word to detect when the user interacts with the application, which specific UI widget the user interacts with, and the type of interaction. Just from a cursory glance, it is obvious that significant engineering effort is required to log the UI interaction of a single application, and we would need to repeat the process for every single application the enterprise uses. Furthermore, instrumentation requires us to have access to the source code or at least some knowledge of the application's inner working, because we cannot instrument the application without knowledge of which part of the application to modify or monitor. This would not be an issue if all applications were open-source (i.e., source code freely available), but this is not the case with Microsoft Office products' and other applications' source code being proprietary.

The second approach, instrumenting GUI libraries, is easier since applications share libraries among them, but this approach still requires a significant amount of work because there are quite a few UI libraries for each operating system and programming language (e.g., Chromium Embedded Framework, CEGUI, GTK+ for C, C++, Swing, JavaFx, SWT for Java, and Kivy for python). These libraries are a fraction of the UI libraries available to developers. Not only must we instrument each library, but we must also contend with the same issue as the first approach, mainly that the UI libraries can also be proprietary.

The third approach is the one we select. We use a specially-designed library supported on most operating systems that is capable of complete and comprehensive monitoring of the user's interaction with the GUI, regardless of the application under test, thereby overcoming the major issues of the first two approaches. We develop one instrumentation that works across applications and UI libraries, bypassing the need to have access to their source code. The framework which meets our demands is UI Automation, which is an accessibility framework provided by Microsoft. UI Automation is able to acquire the hierarchical relationship between GUI elements and the properties of each element such as its name, class, and content type of the GUI element. UI Automation is supported on all Windows operating systems since Windows XP and works with all applications and browsers not developed in Java. In the future, we will use the Java Accessibility Framework [25] to cover Java applications running on the Windows operating system.

When a user is interacting with an application, we intercept the GUI information before the application receives the user input, such as mouse clicks and key presses. This condition must be fulfilled because we use UI information as a prerequisite for network activity, so the UI information must be available before the network activity occurs. If we acquire the UI information after the application processes the user input, the resulting network activity occurs without the corresponding UI information. When analyzing the network activity, we would then be working with an empty UI field or with outdated UI information, and possibly make an incorrect judgement. We therefore use Windows hooking to intercept mouse clicks and keyboard presses and with the aid of UI Automation, retrieve the GUI information before the application receives the user input. In short, Windows hooking allows us to insert our own subroutine between the user input and application; we explain it in further in Section 4 of the thesis.

Some applications separate the GUI and network operations into different processes or services; thus, our system bundles the application's related processes in order to associate the UI and network activity. The Chrome browser for example, spawns multiple processes in the following categories: Renderer, GPU, Plugin, and Browser. A Renderer process is responsible for anything inside a tab where a website is displayed while the Browser process is responsible for the application as a whole including privileged activities such as network requests. As a result, we must associate UI information of the Renderer process with the network request of the Browser process. Otherwise, the prerequisite UI field will be empty when the browser makes a network request.

In the interest of performance, the GUI information is compressed with Huffman compression and saved in a linked list. UI Automation provides UI contextual information as a wide string, and each UI element can run as long as 100 characters or more with a single wide character taking sixteen bits in memory. As a result, UI information storage and transmission in string format is inefficient, so we compress the UI information with Huffman compression. Huffman compression is a lossless data compression technique which compresses data with symbol-to-symbol replacement, with frequent symbols replaced with short encoding, and rare symbols with a long encoding. In our case, the symbol is UI contextual information of a single UI element and the encoding is a two byte binary encoding. The compression performs well when the frequency of the

compressed elements is known. In our case, UI information is repeated frequently, because most users only interact with a subset of available UI widgets and, as a result, the compression technique best fits our need. The collected and compressed information is sent to a centralized access control server each time a network request is made.

Our approach must not only collect UI contextual information in real-time, but also check its validity. The adversary can compromise our system through deception or with an attack on the dependencies of the IDS itself. Some software can simulate keystrokes and mouse clicks, so it is necessary to differentiate between software-generated and hardware-generated user input. The attacker can also replace the UI Automation library with a fake one that returns false information. To combat these software-generated attacks, we differentiate between hardware and software-generated inputs. To achieve this, we monitor the mouse and keyboard activity at the kernel space and the administrative space separately. To do so, we install a mouse and keyboard driver that monitors user input at the kernel level and compare the activity to mouse and keyboard activity at the administrative level. Our system also ensures that applications have loaded the correct DLL for UI Automation by checking the path of the UI Automation DLLs loaded. An adversary may try to terminate the IDS, so we run major components of the IDS with administrator privileges.

3.6 Policy Generation

We develop three different types of policies to differentiate between benign and malicious network traffic: 1) baseline behavior policy, which accounts for background network activity independent of user behavior; 2) UI activity policy, which accounts for network activity to predefined destinations caused by end-user behavior; and 3) handcrafted dynamic behavior policy, which account for network activity to user-specified destinations, such as those caused by URLs.

The first two policy types, baseline and UI activity, are similar: they both result in network activity to predefined destinations. We can create policies for both cases by observing past behavior, because the destinations of the network activity do not change often and remain constant for a relatively long period. As a result, we can expect the same user behavior or lack thereof to result in the same network activity observed in the past. To create the policies for the first type, a simple script will suffice. The first type of policy accounts for background network activity independent of user behavior to predefined destinations that are hardcoded in the application's source code. For example, when a user launches Microsoft Word, it automatically checks Microsoft-owned servers for updates. To develop these types of policies, we observe the network activity of applications when there is no interaction with the end-user.

The second type of policy accounts for network activity to predefined destinations resulting from the end-user's interaction with the application. These interactions have increased along with the complexity of modern applications, due to richer UI interfaces. For example, in Microsoft Word, the Wikipedia widget, the Smart Lookup button, and the Thesaurus button connect to destinations predefined by Microsoft Word. As a result of predefined destinations, we can develop UI activity policies by observing network activity when

the user is interacting with the application.

The last type of policy requires special care due to user-specified destinations, so we handcraft them. The dynamic behavior policy is different from the baseline policies, because we cannot expect the same activity or similar activities to lead to network destinations observed in the past. Therefore, we cannot use the same approach used for the first two types of policies and would need to develop custom policies by hand. This does not mean that observing similar behavior is not beneficial. For example, assume, in Microsoft Word, that we observed the UI activity of the user clicking on www.google.com and the resulting network activity. We then observe the user clicking on www.facebook.com and the resulting network activity. These two activities are not the same, and we cannot use a simple script to develop the policy for such an activity, but they are similar nonetheless. The UI activity of clicking on a link is the same in both cases, and the resulting network activity is to hosts that are present in the UI activity. Thus, we create a handcrafted policy to generalize hyperlink activities that occur as a result of the user clicking on a link in Microsoft Word shown in Figure 3. Creating such a generalized policy would not have been possible without sufficient observation of similar activities. We explain how we create the three types of policies in further detail in the implementation section of the thesis.

3.7 Data Generation

To generate the training and test data for the experiment, we used Sikuli [6] and DLL injection [1]. The data generation can be done manually by end-users in the form of experiment participants or with software that emulates user behavior. The first approach closely resembles actual usage in an enterprise setting, but is challenging: it is laborious and requires significant human hours to collect large amounts of data. We, therefore, generate the data using software that emulates user behavior. To emulate a compromised application, we inject a mock-malicious DLL which is capable of making network connections that can be considered malicious.

We chose Sikuli, a GUI testing framework, to emulate end-user interactions with the application's UI. Sikuli uses visual recognition to detect GUI objects on the screen and simulates mouse clicks and keyboard presses to test an application's GUI. We developed workflows, a sequence of UI actions such as clicking on buttons and typing sentences. We developed some workflows which generate network traffic and some workflows which do not, and we further randomized the order and timing between workflows to more closely resemble a user's nonlinear behavior when interacting with the UI. Some examples of workflows are: the print document workflow, search a keyword workflow, and search for a Wikipedia article workflow. When we conduct experiments, we chose workflows at random. To simulate the malicious behavior of an adversary, we used DLL injection to inject a mock-malicious DLL into the application's process space and run it. Thus, we enabled the mock-malicious DLL to run with the application's credentials and UI context. We ran workflows a certain number of times to generate training and test data, and we only injected the malicious DLL when we generated the test data.

We did not use human subjects because it has significant drawbacks. First, using human subjects does not allow us to test the validity of our approach despite closely resembling a usage in an enterprise setting. Our approach assumes that all activity observed in the test data occurred in the training data. This assumption must hold because we use the training data to create policies which predict the behavior in the test data. As a result, if the test data contains UI activity that causes network traffic and is not present in the training data, we cannot evaluate our approach fairly. When human subjects generate training and test data, they can make small mistakes, such as clicking on the wrong widget. These small mistakes create new traces in the test data, making us unable to test the validity of our approach. Second, interacting with the UI and repeating specific actions is labor-intensive for the subjects and requires significant human hours to collect a meaningful amount of data. For example, to print a document in Microsoft Word, the subject interacts with three different widgets from Word's homepage. The subject would need to carry out the "print document" scenario dozens of times for us to collect enough data to create a firm policy that covers edge cases. We have many scenarios like the "print document" scenario for each application under test, which makes human subjects undesirable for our approach. Based on these considerations, we decided to use software to emulate user behavior.

3.8 Browser

We do not use the same approach as desktop applications with browsers because of a browser's ability to load web pages and the numerous network connections it makes. Each webpage a browser loads can be thought of as a single unique application because it is executing code provided by the server. As a result, if we profile web applications, we would have to profile each website individually as we do with applications. This profiling would need to be done regularly because, unlike desktop applications, web developers can easily change web pages. However, this is not possible because of the numerous webpages the user might visit, and, as a result, we would need to perform the training phase to create policies for each website. This is impractical in real life. Furthermore, handcrafting policies for websites is difficult because network behavior among sites is not consistent. As a result, we cannot feasibly craft a generic policy to handle all websites. For example, facebook.com mostly contains hyperlinks to other Facebook domains, but low-traffic personal sites may include hyperlinks to a multitude of other domains. Furthermore, most sites contain hundreds of links and can change daily. As a result, we develop a different approach for browsers.

Since it is not possible to predict browser behavior based on past activity or handcraft policies for it, we approach the problem by constructing the policies dynamically. The links within a webpage are requested after the browser loads the webpage. As a result, intercepting the webpage before the browser has a chance to process it will allow us to predict what the browser will request when the webpage is loaded. To achieve this, we can intercept the webpage at the user level with browser plugins or at the kernel level with the Windows Filtering Platform (WFP). Using browser plugins to analyze the Document Object Model (DOM) of a webpage and extracting the links requires us to create plugins for every browser on the market. While

this is possible, browser plugins run at the same privilege space as the browser itself. For example, if a user launches the browser with user privileges, the plugin will also have user privileges. It is therefore possible to compromise the plugin in our threat model. The loaded webpage itself can be compromised at the end-user's machine if an adversary injects links to the webpage. As a result, even though browser plugins are capable of intercepting the webpage, it is not secure under our threat model. On the other hand, WFP runs in kernel space and is capable of intercepting browser traffic. WFP allows us to access the packet processing and filtering pipeline of the IP/TCP stack. We use Application Layer Enforcement (ALE) portion of the WFP to intercept the TCP packets, which enables us to monitor network traffic at the per-connection or per-socket level and monitor TCP and UDP socket operations. We register callout functions in our network driver to that is notified when TCP packets are in flight.

Once we intercept the webpage, the links are extracted to create a policy. For example, the policy can be a general one that allows connections to the domain names of the hyperlinks found in the webpage. The engineering effort required for this approach is significantly more than creating browser plugins due to working in the kernel space, but it is secure under our threat model. As a result, we have settled on this approach to create a live policy mechanism for browsers.

4 Implementation

4.1 Application monitor

The application monitor must be comprehensive in its ability to acquire UI contextual information, regardless of operating system or application, and precise in its ability to monitor user activity such as keystrokes and mouse clicks. The implementation must be capable of intercepting user activity before the application. It must also know which application the user is interacting with.

4.1.1 UI context

Acquisition of UI context must be universal and accurate. We have considered three possible approaches: 1) instrumentation per application, 2) instrumentation of UI libraries, and 3) a dedicated API which fulfills the requirements. As described above in Section 3, the instrumentation of each application or individual UI library is labor-intensive, so we have opted to use UI Automation, the accessibility framework provided by Microsoft [18]. Microsoft developed UI Automation to support programmatic access to UI elements on the desktop for accessibility purposes, enabling technology such as screen readers. UI Automation is capable of acquiring information from UI elements/widgets, such as name, class type, control type, and the hierarchical information of UI widgets. All Windows operating systems since Windows XP support the framework, and the framework is compatible with all applications except those developed in Java. UI Automation represents the totality of the UI interface of the operating system as a tree data structure, with the desktop widget as the root element and all applications and other important control elements, such as the Windows widget

and the search bar, as its children. Using the previously mentioned capabilities of UI Automation, we can differentiate between widgets using their position in the hierarchy and their properties.

The basic building block of UI Automation is the IUIAutomationElement object, which represents a single UI widget in Windows operating systems that have a multitude of properties including, but not limited to, the name of the element, the class type, and the control type. The IUIAutomationElement offers a multitude of properties, but for our purposes, the name, class type, and control type properties are enough to differentiate between UI elements. Further IUIAutomationElement properties can be used to differentiate between elements, but they are more general. These properties do not vary significantly among UI elements and are not descriptive of the UI element. Further, the retrieval of multiple properties can slow down the machine. As a result, we limit ourselves to the previously specified three properties to fulfill the fine-grained requirement. The IUIAutomationElement object does not offer any way to traverse the UI tree or to acquire a specific UI element. As a result, we use other classes in the framework to traverse the UI tree. To acquire a starting point to traverse the UI tree, we use three methods in the framework: GetRootElement, GetElementFromPoint, and GetFocusedElement. These methods retrieve the root UI widget, the UI widget at the coordinates under the mouse pointer, or the currently focused UI widget respectively. We use IUIAutomationTreeWalker, a class which exposes methods to walk through the UI tree, to acquire the parent or the children of a UI widget provided the IUIAutomationElement object of the UI widget. Using these methods, our implementation is capable of acquiring the properties of the widgets the user has interacted with and traversing the UI tree from said element.

4.1.2 UI Activity

Users interact with the UI of applications primarily through the mouse and keyboard. Harbinger intercepts these events and extracts the necessary information, such as the coordinates of the mouse click before the application receives the user input. In the Windows operating systems, applications with GUIs are event-driven and must have message queues that process Windows messages meant for the application's GUI. Whenever a mouse or a keyboard event occurs in Windows, the operating system creates and sends a Windows message to the application currently in possession of the UI focus. As a result, our Application Monitor must intercept Windows messages pertaining to mouse and keyboard events.

Luckily, Windows offers a method to intercept these messages by using hooks. A hook allows us to register our subroutine to monitor Windows-message traffic in the Windows message-handling mechanism for certain types of messages. In Harbinger, we register two separate hooks for the mouse and the keyboard. The mouse hook is responsible for monitoring mouse click-down events, while the keyboard hook is responsible for key-down events. We register hooks with the SetWindowsHookEx function, which takes four parameters; the most important two are the first and second parameters, which take the identifier of the hook, which specifies the hook type, and a pointer to the function to call when the hook event occurs. For example, we register the mouse hook with the _MOUSE_LL identifier and a pointer to the function to call in the event of a mouse

click. Furthermore, the registered hook can not only differentiate between different mouse events such as left mouse down and right mouse up, but can also access the coordinates of the mouse. In the case of the keyboard hook, the hook provides the specific pressed key.

4.1.3 Inner Workings of the Application Monitor

Using UI Automation and hooks, Harbinger can monitor the user's interactions with the UI comprehensively and accurately. The following is a detailed description of the inner workings of the Application Monitor. When a user clicks on a UI element, our registered mouse hook checks whether the Windows message is a button-down message. If so, the mouse hook acquires the coordinates of the mouse pointer, and using the GetElementFromPoint functionality of UI Automation, acquires the UIAutomationElement object of the widget. Once the UIAutomationElement object is acquired, we retrieve the name, class type, control type, and process ID of the UI widget using the object's methods and save it in a custom data structure with the timestamp of the click event. However, there is a possibility that multiple widgets with the same properties exist both within and across applications. As a result, to differentiate such widgets, we iterate over the widget's ancestors until the root UI element and perform the same action on them. We save the retrieved information of the element and its ancestors in a custom data structure. Furthermore, we connect the custom data structure of the ancestors via a pointer, set the activated widget's data structure as the root, and push the root structure into a vector for storage and easy retrieval of UI information. The custom data structure has a wide string field for storing UI information, integer field for storing process ID, high-resolution time point field for storing timestamp, and a pointer field for storing a pointer to its parent.

In the case of keyboard events, a similar approach is inefficient. Keyboard events are numerous when the user is editing text, and populating structures consecutively with the same information is a poor utilization of computing resources. As a result, Harbinger takes a different approach utilizing UI Automation's event handler for UI focus change events. As the name suggests, this event handler is triggered when the focused UI changes, whether from one application to another application or from one widget to another widget within the same application. When a keyboard key is pressed down, the keyboard hook first checks whether a focus change event has occurred. If so, the keyboard hook performs the same actions as the mouse hook, except this time, it starts from the focused UI element. We acquire the IUAutomationElement object with GetFocusedElement method, which returns the currently focused UI widget. We made this change because the user is navigating the UI with the keyboard and not the mouse. However, if a focus change event did not happen, the user is likely editing text and not interacting meaningfully with the UI. In such a case, we update the timestamp of the last structure added to the vector of structures. We save UI information in a vector of structures with the most recent widget added to the beginning of the vector.

4.1.4 Validity of UI Activity and Context

An adversary may provide false UI contextual information or generate UI activity through software, but the Application Monitor (AM) takes precautions against such attacks. For an adversary to provide false UI information, they would have to compromise UI Automation. As a result, the adversary would either have to compromise the AM or the application under monitor (AUM).

UI Automation loads its libraries into both the AM and the AUM at the time of usage. As a result, in order to compromise the validity of the UI information the AM collects, the adversary can replace the UI Automation libraries with malicious libraries. UI Automation differentiates applications as client and provider and loads different libraries into each. The client application, in our case, the AM, is the application that requests the UI information from another application, and the application which returns the requested information is the provider, the AUM. As the roles of the applications differ, the libraries they load are different. The client and the provider both load UIAutomationTypes.dll and UiAutomationCore.dll, and the client and the provider are required to load UIAutomationClient.dll and UIAutomationProvider.dll, respectively.

An adversary may try to replace the UI Automation DLLs with malicious ones before the AM, or the AUM loads the libraries. To mitigate this risk, we check if the AM and the AUM loaded UI Automation libraries from the directories `C:\Windows\`, `C:\Program Files\`, and `C:\Program Files (x86)\`, which all require administrative privileges to alter, which the adversary lacks under our threat model. Furthermore, these directories are the default directories for UI Automation DLLs. To accomplish this task, the AM first acquires the AUM's process ID with a method from UI Automation, then, with the `CreateToolHelp32Snapshot` function, the AM lists all DLLs that the AUM loads and their corresponding directories. If the AUM loads UI Automation libraries from suspicious directories, it is apparent. We use the same procedure to validate UI Automation libraries in the AM. We validate UI Automation libraries in the AM and AUM only once during startup and first interaction, respectively. We save the verified process IDs in a vector so that we do not validate them more than once.

The adversary may also try to create false UI activity by generating mouse clicks and key presses using software. The adversary can achieve this by using tools such as Sikuli, a GUI testing tool, to mimic user behavior. We use Sikuli to generate the training and test data for our security evaluation, and we find that Windows hooks are unable to differentiate between hardware-generated and software-generated mouse and keyboard activity. As a result, we add mechanisms to differentiate between hardware and software generated user activity to the AM. We have created two drivers, one for the mouse and one for the keyboard. These drivers record which keyboard keys and mouse buttons were pressed, and save them in a linked list in the kernel space, with the most recent key pushed to the back of the list. When the hooks detect a mouse click or keyboard press, the AM checks if the corresponding mouse or keyboard activity was observed by the drivers, by requesting the most recent mouse click or keyboard press from the drivers. The AM compares the user input observed by the hook against user input observed by the driver. As a result, if a mismatch occurs, the

AM can detect the incongruity and alert the user.

4.1.5 Bundling of processes and services

Modern applications are no longer composed of only a single process, but spawn multiple processes for different purposes, as in the example of the Chrome browser we gave in Section Three. Thus, an application may decide to delegate its GUI and network functionalities to separate processes. In such a case, the AM cannot retrieve an application's GUI information because the process ID of the GUI and the process ID of the network components of an application are different. In order to overcome this, the AM finds all processes belonging to an application, and their process identifiers provided only one process ID belonging to the application. The controller provides the initial process ID to the AM. From our observation, we found that applications composed of multiple processes have all spawned their processes from the same executable and have parent-child relationships; as a result, we take advantage of this and bundle the application's processes by using their executable path.

In our implementation, given a process ID requested by the controller, the AM uses `OpenProcess` function to get the process handle, which is a unique ID given to processes in Windows systems, and with the handle and `GetModuleFileNameEx` function, acquires the executable path of the process. Once the executable path of the requested process is acquired, we use `CreateToolHelp32Snapshot` to take a snapshot of all running processes on the system. The snapshot is the read-only copy of the current state of all processes running on the machine, which includes their process ID. Then we iterate over the snapshot `CreateToolHelp32Snapshot` returned and retrieve all the process ID's of the active processes. Once all the process ID's are known, we use the same approach mentioned previously with `OpenProcess` and `GetModuleFileNameEx` functions to get the executable path of all the processes. By comparing the requested process ID's executable path against all running processes' executable paths on the system, we can get all the process ID's of the application. We save this information in a vector.

4.1.6 Browser Instrumentation

Web browsers require extensive attention due to their dynamic nature. It is not possible to use the same approach as desktop applications with browsers because of their ability to load web pages and the number of network connections they produce due to hyperlinks. As we have explained in Section 3, constructing policies based on past behavior or handcrafting is not possible, so we construct live policies by intercepting web-traffic before it reaches the browsers and extract the hyperlinks.

We use the Network Monitor to intercept browser traffic, both inbound and outbound, by monitoring TCP packets to and from ports 80 and 443, which are ports associated with the HTTP and HTTPS protocol. We register two callout functions at the transport layer to intercept TCP packets and send the packets to the Application Monitor. When the Application Monitor receives those packets, it sends the packets to the `Tshark` tool for reassembly and decryption. `Tshark` requires packets to have Ethernet headers to process

them correctly, but since we intercepted the packets at the transport layer, the Ethernet headers are missing. We, therefore, constructed Ethernet headers artificially and prepended the header to each packet. We used pipes to transport the packets between the Application Monitor and `Tshark`. Once the packets have been assembled and decrypted by `Tshark`, the Application Monitor searches the packets for hyperlinks and saves the hyperlinks in a map data structure with the process ID of the browser as key.

When the browser loads the webpage, which occurs after the Application Monitor extracts all the hyperlinks, the browser will make additional requests to retrieve resources pointed to by the hyperlinks depending on the user action (e.g., clicking on an article at nytimes.com). The Network Monitor intercepts these additional requests and consults the policy engine to make a decision. We compare the requests hostname against hyperlinks the Application Monitor acquired by intercepting HTTP and HTTPS traffic. Since both the Network Monitor and Application Monitor save and retrieve data based on process ID, this comparison is straightforward. If the Application Monitor has observed the requested webpage domain name, we allow the network request through.

4.1.7 Compression

When the controller requests UI information from the client, it is essential to send consistent and complete UI information for Harbinger to make correct decisions. However, the maximum amount of information currently transferable is 1500 bytes. Furthermore, UI Automation represents UI contextual information as widestrings which takes up 2 bytes, combined with the fact that depth of a trace varies greatly, and with three traces to send, it is not uncommon for the AM to send incomplete UI information to the controller. As a result, we implement functionality to compress UI information in the AM and the functionality to decompress the information in the controller.

We use Huffman Compression, which is a variable-length lossless compression algorithm, due to its ease of implementation and previous uses in text compression. Huffman compression compresses data with symbol to symbol replacement, with frequently occurring symbols, in our case UI element information, with a short encoding, and rare symbols with a long encoding. This algorithm works best when we know the probability distribution of the symbols to be compressed. For decompression, we perform the symbol to symbol replacement in reverse order.

In our implementation of Huffman Compression, we set our symbols to be the tuple combination of UI element's name, class type, and control type. Furthermore, we make additional changes to the default compression algorithm by changing the variable-length encoding to fixed-length encoding and added the ability to deal with symbols unencountered before. As mentioned, the probability distribution of the symbols to be compressed must be known. Thus, before we start compressing, we calculate the frequency of each symbol by generating training data for each application, and then we use a Python script to calculate the frequency of each symbol and generate the symbol to symbol compression file. We use this file as a lookup table for compression and in reverse lookup fashion for decompression. However, this do not guarantee

that we have an accurate representation of the probability distribution of the symbols, nor that we have encountered all the symbols/UI elements. As a result, our compression and decompression methods take this into account and does not try to compress or decompress previously unseen symbols. Encountering unseen symbols is a valid assumption not only because the training data can be incomplete, but also because the UI might change due to updates, in such a case, we generate the training data again.

4.2 Network Monitor

The network monitor serves two purposes, intercepting network connection at the host system and coordinating the application monitor with the policy engine. We implement the network monitor as a kernel-mode driver, which uses the Application Layer Enforcement (ALE) provided by Windows Filtering Platform (WFP) to filter packets on a per-connection or per-socket basis. ALE is also capable of monitoring all socket activities, such as the creation of TCP and UDP connections.

The Network Monitor registers callout functions to detect new network flows, and uses the inverted call model [24] to query the application monitor for the GUI data of said flows. While driver routines cannot directly perform a callback to a user-level application, we can use the inverted call model, which is similar to registering an event handler. The user-level application registers device control codes that are pending and accessible by drivers. When a driver wishes to notify the application of some event, the pending control code is completed. We register callout functions to monitor connect, accept, and close operations of sockets, and also to intercept TCP packets for the browser instrumentation. We include the process ID of the application creating the new network flow, so the Application Monitor can return the GUI information associated with it.

The Network Monitor uses the inverted call module to communicate with the Application Monitor. The inverted call must include the process ID of the application making the network request, and we acquire the process ID from the registered callout functions. After the GUI information is retrieved, the network monitor packages the packet with the GUI information and sends it to the policy engine for an access control decision; all subsequent packets are queued in the network monitor until the policy engine returns a decision. If the policy engine denies the flow, the network monitor discards all subsequent queued packets, and it also drops any future packets in that flow associated with the application. If the policy engine approves the flow, the network monitor re-injects all the queued packets and marks the flow as approved in the WFP.

In an enterprise setting, all network monitors and machines will communicate with a centralized controller that manages the policy engine. We have set the size of the packets between the network monitor and the controller to be 15000 byte MTU to avoid packet segmentation or fragmentation. Therefore, the controller can make quick decisions because there is no need to reassemble the packets. Harbinger uses the OpenFlow 1.0 protocol to communicate with the controller and policy-engine for compatibility purposes with a software-defined network [23]. The network monitor can communicate with multiple controllers for access control decisions to account for network failure between the network monitor and the controller. Harbinger can

communicate with an arbitrary number of controllers, choosing one as the primary and the rest as backup controllers. In case communication with the primary controller is lost, the network monitor switches to a backup controller for access control decisions.

4.3 Policy Engine

Our policy engine receives information collected by the Application and Network monitors at each host machine and saves them in a local database. When the policy engine receives a new network request, it consults the event traces received from the application in reverse chronological order to arrive at a decision. We examine traces in reverse order because user behavior can be nonlinear, such as switching between applications or multitasking. Clients only report the three most recent event traces (GUI traces) to the policy engine because older traces are likely to be irrelevant to the application's most recent network behavior.

The policy engine creates policies offline based on the collected information periodically. Therefore, when the policy engine receives a network request, the policies governing the access control decisions are pre-defined. The policies are saved in a backend MySQL database and implemented as a rule cache that is refreshed periodically. A policy rule includes network and transport fields such as IP address, protocol, and hostname, as well as process path and process ID, and UI information. Policies also have a priority field, which indicates the importance of the policy, and the controller makes flow decisions according to the highest priority matching policy. We also added a wildcards for coarse-grain matching. The controller writes to a buffer each time we make a decision, and we periodically save the information to the backend database.

We create two types of policies to encompass software-defined and user-specified network activity. An application's network behavior stays relatively constant if they are software-defined, so the same UI elements always trigger the same network activity. We can create policies for this type of network behavior by observing past UI activities and their corresponding network activity. We observe the number of consecutive network connections the same GUI trace makes and the timestamps (temporal distance) of those connections to create the policies. Due to DNS caching and the availability of computing resources at the end-user's machine, the number of network connections and the temporal distance between them may vary. Therefore, we create policies with upper and lower bounds on the number of connections a GUI trace can make and the temporal distance between the activation of the element and the network activity. We do not take the same approach when creating policies for user-specified network destinations because we cannot predict them based on past behavior because the user can provide previously unseen destination. We tackle this problem by observing the relationships between the GUI field and other fields such as the destination IP and hostname to create handcrafted policies. Our policy engine implements a handcrafted policy for handling hyperlinks as an example. The handcrafted policy searches if the destination hostname is present in the GUI field, as this would happen when the user clicks on a hyperlink. In such a case, the handcrafted policy would allow the network activity. This single handcrafted policy works across the applications we are testing and is a proof of concept.

We create the policies using a Python script and with the data saved in the backend database from observing each end-user. The data output as a text file so that our Python script can parse and create policies. We use Pandas, NumPy, and regular expression libraries for indexing and comparing UI fields[36, 22, 28]. Our Python script searches for consecutive traces with matching UI fields and creates a single policy for that event. The policy has additional fields for lower and upper bounds for keyboard and mouse activity and timestamps. We observe the bounds from the consecutive traces. The policy rules look similar to traditional firewall rules with additional fields, including UI context information and executable path, and a field indicating the approve or deny order. If our policy engine does not find a matching policy for a network request, we deny it by default.

5 Experiment environment

We performed the security and performance evaluations on virtual machines running on separate physical VM hosting servers. The policy engine and the controller ran on Ubuntu 16.04 VM, which was allocated 1 core and 2 Gbytes of RAM on a machine with 12-cores and 64 Gbytes of RAM. We run the applications under test on Windows 7, which was allocated with 2 cores and 4 Gbytes of RAM. Furthermore, during testing, one controller and 5 virtual machines running Windows 7 were used.

5.1 Application Under Test

We decide to test applications that are commonly found in enterprise settings, as we built Harbinger for detecting intrusions in an enterprise network. However, we cannot test every prominent application used in an office setting, so we choose to pick applications that would represent a whole class of applications in enterprise settings. We chose Microsoft Word, Powerpoint, Excel, and Onenote to represent office suites commonly used for processing documents, creating slideshows, calculating spreadsheets, and sharing notes, respectively. With this, we can represent a myriad of applications without testing each.

5.2 Automatic generation of UI activity

To perform a security evaluation, we need to produce and collect training and test data that would represent normal usage of an application under enterprise settings. Furthermore, we must make sure that user activity which occurs during the collection of the test data also occurs during the collection of the training data because we cannot judge user activity we have not seen before. To generate a sufficient quantity of such data would be error-prone and tedious if it were to be performed by people; as a result, a tool that mimics user behavior is appropriate.

Sikuli is a tool which was developed for the purpose of testing the GUI portion of applications by mimicking user behavior by automating any interaction the user may have with the visible portion of the screen [32]. It uses image recognition to identify GUI components and interacts with them by controlling

the mouse and keyboard through software means. This tool fits our needs perfectly not only because of its ability to mimic user behavior but allows us to create user behaviors for each application without the need for access to each application's source code.

For each application under test, we create dozens of GUI traces. A trace could be thought of as a natural progression of GUI activity such as mouse clicks and key presses which achieve a single purpose such as searching for an image or printing a document. For example, to print a document in Microsoft Word from its homepage, first, the user would need to click on the File tab positioned at the top of the Word GUI, then on the Print tab located at the left side, and finally the Print button. This action achieves one goal/trace but is a combination of multiple user activity. Furthermore, traces created for each application fall under two categories, ones that produce network activity and ones which do not. We have decided to include both types because users do not exclusively follow one type of traces over the other. However, there is difficulty in convincingly mimicking user behavior because of the rigidity of software-generated traces in timing and order. To overcome this difficulty, we introduce randomness in the choice of trace and the variability between traces and the actions within traces in the form of random number generator and random timer. We have decided that a single run of Sikuli test script would be composed of roughly 10 random traces per application, excluding the traces required for starting and ending the execution of said application. For training and test data collection, we executed the test script consecutively over the course of a week and two days, respectively.

6 Security Evaluation

To test the effectiveness of the IDS, we conduct three different experiments with changes to the behavior of the Sikuli test script and the mock-malicious DLL to simulate changes in user behavior and the attacker's goals respectively. Furthermore, to prove the value of using both UI activity and traditional network information as opposed to only traditional network information, the same greedy matching algorithm is used to make security decisions in both cases. As a result, direct comparison of detection rates will be valid and shed light on the importance of UI contextual information.

When evaluating Harbinger, we have two main scenarios to consider: 1) a deterministic network destination and 2) a user-supplied network destination. Deterministic destinations do not change based on the end-user's behavior or the environment. They are pre-configured within the application by the end-user or the organization in advance. Changes in deterministic destination usually occurs when the application is updated, or the IT staff configures them manually. These changes are infrequent, and, therefore, pre-configured destinations remain constant for some time. User-supplied network destinations are provided during run time by the user, usually in the form of URLs. Thus, they can be considered to be in a constant state of flux until the application makes the network request. These two scenarios are the complete opposites of each other. The security evaluation section will discuss how Harbinger utilizes different approaches to building whitelists, that function well in both scenarios. We discuss the policy creation for each approach

in-depth and compare and contrast the security performance of each policy class.

6.1 Policy Classes

We create three different classes of whitelists, which we differentiate according to the features taken into consideration when we created the policies. The reason for differentiating policies according to their features is to highlight the importance of them when creating policies, and the insight the GUI field offers.

1. **Class 1: Traditional IP Whitelists.** We construct the first class of whitelist the same way traditional firewalls construct whitelists. These whitelists are created based on the standard five-tuple (IP_{src}, IP_{dest}, transport protocol, port_{src}, port_{dest}). In this case, the source IP address is fixed with random source ports, thus effectively focusing on destination IP addresses and port numbers. We consider the performance of this whitelist to be the baseline because of its resemblance to traditional firewalls.

2. **Class 2: DNS Whitelists with User and Application Path.** We construct the second class of whitelists based on the standard five-tuples with the addition of domain name resolution and application path. This additional contextual information allows Harbinger to make decisions based on which application initiates the network connection and whether IP addresses point to the same domain. In this case, we focus on destination IP, Port, DNS, and application path. This class of whitelist highlights the contribution to security offered by Harbinger's extra visibility on the host system. However, we do not include the contextual information offered by the user in the form of GUI.

3. **Class 3: Dynamic Whitelists with User Inputs and GUI Data.** We construct the third class of whitelist by leveraging the full contextual information Harbinger has access to on the host system. The additional contextual information provides detailed GUI data, mouse clicks, keypresses, in addition to those available to the second class. This GUI data will sufficiently prove their usefulness as they give access to hyperlinks and buttons, which may cause network traffic.

We use a confusion matrix for comparison to determine the effectiveness of each class of whitelists. The matrix lists how much benign and malicious traffic each class of policy correctly classifies and misclassifies.

6.2 Policy generation

Before running any experiments, we monitor baseline activities of the applications devoid of user interactions. We believe recording such activity allows Harbinger to know about connections that run in the background regardless of user interaction. We do not want to create policies which incorrectly infer background network activity to be caused by the end-user. Therefore, we must differentiate between background and user-initiated network activities. We collected background network activity by starting each application and monitoring its network traffic over some time. We use the baseline data for constructing BaseLine Whitelists for all

three classes and each application. We do not take into consideration the GUI field when creating whitelists for background activity, and any activity to destinations in the BaseLine Whitelists are allowed. In Class 1 Baseline Whitelist, we only whitelisted the destination IP addresses. In Class 2 and Class 3 Baseline Whitelists, we whitelisted both the destination IP addresses and destination hostnames.

In addition to Baseline Whitelists, we create whitelists which covers network activity caused by the end-user. We use a training phase in which we programmaticaly simulate user interactions with each application under test to generate the training data. We then use a greedy algorithm to create application-specific whitelists that allow all network activity in the training data for each application. We focused on creating tight policies when possible specifying the exact destinations, but for properties which vary such as source port numbers, we use wildcards. We do not perform packet analysis because packets are usually encrypted and can be modified by the adversary.

In Class 1 whitelists, we allow network traffic to any destination IP address observed in the training data and the BaseLine Whitelist. In Class 2 whitelists, we allow network traffic to any destination IP address or hostname that occurs in either the training data or the BaseLine Whitelist. We deny the network traffic otherwise.

In Class 3 Whitelists we allow network traffic 1) if the destination IP address or DNS is observed in the BaseLine Whitelists or DNS Pinning Whitelist (described below), 2) if the destination hostname or IP address appears as a GUI substring in the form of a URL in the GUI data, or 3) if either the destination DNS or IP address observed in the training data appears with its corresponding GUI string. Browsers cache the IP address associated with hostnames for performance purposes and to defend against DNS binding attacks. This technique is called DNS pinning [14]. Our policy script saves the associated IP address of DNS hostnames, which appeared in the training data with the corresponding GUI field to create DNS pinning Whitelist. For any future occurrence of the corresponding GUI field, we match it with its DNS hostname, IP address, or DNS pinned IP address. For such occurrences, we allow the network traffic. We deny network traffic unable to meet any of the above criteria.

6.3 Experimental Setup

We examined the performance of the three different classes of whitelists on traffic from four desktop applications representing their specific classes of application: Microsoft Word, Microsoft Excel, Microsoft PowerPoint, Microsoft Word. We interacted with the applications using the Sikuli framework [37]. For each application, we provided application-specific workflows that include a variety of traces that generate network traffic and ones that do not, with differing mouse and keyboard activity. We hoped to emulate the end-user behavior by randomizing the order of workflows, the pause between workflows, and the number of mouse clicks and keypresses. The workflows may be fully deterministic or provide user-supplied destinations from a list depending on the experiment. The setup is sufficient to emulate uncompromised applications to test Harbinger for usability with false-positive rates.

Table 1: Confusion matrices for the three whitelist classes across our three experiments.

Experiment 1: Deterministic legitimate destinations with adversary connections to destinations **not** in the training data

	Baseline	Training	Testing	True Positives			True Negatives			False Positives			False Negatives		
Application	Samples	Samples	Samples	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3
Excel	4,173	3,425	3,446	956	956	956	2,464	2,489	2,487	26	1	3	0	0	0
OneNote	1,764	1,690	2,439	794	794	794	1,641	1,645	1,645	4	0	0	0	0	0
PowerPoint	2,366	792	2,392	721	721	721	1,573	1,670	1,670	98	1	1	0	0	0
Word	3,826	2,809	2,655	560	560	560	2,022	2,095	2,091	74	0	4	0	0	0

Experiment 2: Deterministic legitimate destinations with adversary connections to destinations **contained within** the training data

	Baseline	Training	Testing	True Positives			True Negatives			False Positives			False Negatives		
Application	Samples	Samples	Samples	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3
Excel	4,173	3,425	3,446	0	0	940	2,464	2,489	2,487	26	1	3	956	956	16
OneNote	1,764	1,690	2,439	0	0	794	1,641	1,645	1,645	4	0	0	794	794	0
PowerPoint	2,366	792	2,392	0	0	702	1,573	1,670	1,670	98	1	1	721	721	19
Word	3,826	2,809	2,655	0	0	540	2,022	2,095	2,091	74	0	4	560	560	20

Experiment 3: User-supplied legitimate destinations with adversary connections to destinations **not** in the training data

	Baseline	Training	Testing	True Positives			True Negatives			False Positives			False Negatives		
Application	Samples	Samples	Samples	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3
Excel	5,419	3,733	1,958	644	644	644	714	736	1,311	600	578	3	0	0	0
OneNote	1,913	1,929	1,025	541	541	541	462	482	482	22	2	2	0	0	0
PowerPoint	4,427	1,249	1,652	559	559	559	477	493	1,091	616	600	2	0	0	0
Word	6,002	4,273	1,486	370	370	370	548	555	1,116	568	561	0	0	0	0

In order to mimic malicious behavior, we would need to compromise the applications under test. The method of compromise would need to be valid for all applications we intend to test and have the ability to make network connections arbitrarily. As a result, we have chosen DLL injection as our method of compromise. DLL injection, a technique used for running arbitrary code in the address space of a target application by injecting the code as a dynamically linked library. This technique is commonly used by malicious individuals to compromise applications [2]. We used a DLL injecter developed by a third party while designing our own malicious DLL code to be injected into the applications. The code we injected will attempt to make a network connection to an arbitrary IP address every 15 seconds if a random number generator generates a specific number. Depending on the experiment, the injected DLL will either attempt to initiate network traffic to benign or malicious IP addresses. We are emulating propagation attacks when the DLL attempts to connect to a benign IP address, and we are emulating attempts to reach a command and control center when the DLL attempts to connect to a malicious IP address. In the confusion matrix of our security evaluation, we consider any network traffic initiated by the injected DLL as malicious (true positive) regardless of the destination and all other traffic as benign (true negative).

6.4 Experiment 1: Deterministic Legitimate Destinations with Command and Control Server

Our first experiment is the simplest one with user workflows having deterministic destinations like the case in which the "Wikipedia" button in Microsoft Word always resolves to a connection to the same server. In this experiment, the adversary only attempts to initiate network connections to priori unknown hosts outside the organization. This scenoria emulates attempts such as exfiltrating enterprise data to an outside server, downloading additional malicious software, or just attempting to connect to a command and controller server. The adversary's destinations are not known priori, so it is not feasible for the defenders to create a blacklist policies. However, our approach of whitelisting prior known IP address are likely to succeed in detecting the malicious traffic because the destinations of the malicious traffic are not present in the baseline or training data. In other words, the destination is not part of the organization's IT infrastructure, so detection should be easy.

The top section of Table 1 shows the results of experiment 1. Each application in experiment 1 had at the minimum 1,764 network connections in the baseline data, 792 network connections in the training data, and 2,392 network connections in test data. As expected, the false negatives are zero for each class because the malicious destination IP addresses do not exist in the baseline and training data, thus did not appear in the whitelists. However, the false-positive rates differ. The Class 1 whitelists have the worst performance due to IP load balancing, a scenario in which the hostname is the same, but the associated IP addresses vary. In some of the cases, the destination IP address or the destination hostname is not present in the baseline whitelists, and it affected the performance of all three classes. In Class 3, we see that false-positive rates for Excel and Word are more in Class 2. This was caused by incorrectly associating UI activity with background network traffic and could be remedied with more extensive baseline data collection. Nonetheless, the Class 3 whitelists performed adequately across applications with false positives rates between 0% and 0.2% percent. All three classes of whitelists have 100% detection rate, but Class 1 whitelists may have usability issues because of its higher false-positive rates than Class 2 and Class 3 whitelists.

6.5 Experiment 2: Deterministic Legitimate Destinations with Propagation Attack

Our second experiment is similar to the first experiment except the fact that the adversary changes strategy. While in Experiment 1, the adversary tries to connect to an outside infrastructure, in Experiment 2, the adversary tries to propagate within the enterprise's infrastructure. In other words, the adversary targets benign IP addresses to make network connections. In order to emulate this scenario, we did not collect additional data but simply rewrote the destination IP addresses made by the injected DLL to random ones observed in the training data set. We did not choose destination addresses from the baseline data set because these addresses tend to be manufacturer-operated. In summary, the adversary targets enterprise's

IT infrastructure. The policy matching approaches used in Experiment 1 and Experiment 2 are the same.

The middle section of Table 1 shows the results of Experiment 2. The results differ significantly from Experiment 1, especially in the case for Class 1 and Class 2, as they fail to detect any of the malicious traffic, due to the injected DLL targeting whitelisted IP addresses. As a result, Class 1 and Class 2 have true-positive rates of 0 and 100% false-negative rates. However, unlike Class 1 and Class 2, Class 3 performs markedly well with 0% to 3.6% false-negative rates. This is due to the fact that, while the adversary may provide benign IP address or DNS as destinations, they fail to provide the corresponding GUI string associated with the destinations. However, the false-negative rates are not 0% because some of the adversary's attempts were able to make connections at opportune times where the associated GUI activity with the destination occurred recently. The false-positive rates of Class 3 shows no changes. A more intelligent adversary may try to time their attacks when the associated GUI activity of the destination occurred recently. The adversary could accomplish this feat by monitoring the GUI, just like Harbinger. To combat the above scenario, Harbinger keeps a stateful connection-counting system for policies; therefore, the extra malicious network activity can be detected.

6.6 Experiment 3: User-Supplied Legitimate Destinations with Malicious Command and Control Server

Our third experiment reveals a further advantage Harbinger holds over more traditional approaches. In this experiment, we explore the capabilities of Harbinger when the end-user dynamically sets the destination during runtime, via a URL. To collect the data for this experiment, we modified the Sikuli script to type a URL in the type field of the application and, subsequently, accessing the URL in the typed field. We choose the URL randomly from a database containing the most popular half-million websites, according to Quentcast [31]. This experiment is designed to evaluate the capability of dynamic whitelists, so no changes were made to the behavior of the adversary from Experiment 1, meaning the adversary chose to connect to destinations, not in the baseline or training sets. Each application typed and clicked on between 240 and 420 URLs in the training set, and between 149 and 390 URLs in the test set.

The results of the experiment are shown on the bottoms section of Table 1. We again see 0% false-negative rates, the same as Experiment 1, because the adversary chose destinations absent in the whitelists. However, we see markedly different results when it comes to true negatives and false-positive rates. The Class 1 and Class 2 approaches to constructing whitelists correctly classify between 44% and 55% of network connection. The correctly classified network activity was destined for Microsoft authentication service, which was present in the Baseline Whitelists. However, almost all network traffic caused by URL clicks, were misclassified due to its absence in the baseline or training sets. Class 3 outperforms Class 1 and Class 2 significantly with accuracy between 99.6% and 100% across applications because the GUI string contains the contextual information necessary for extracting the destination hostnames.

6.7 Security Evaluation Results

The three experiments conducted show that application paths and DNS data are useful for correctly classifying pre-configured or pre-programmed benign and previously unseen malicious network activity. However, further information is needed to construct whitelists for correctly classifying destinations provided by the end-user and detecting propagation of malicious activity inside an organization's infrastructure. We see this from the false-positive rates exceeding 50% and false-negative rates of 100% for Class 1 and Class 2. The required contextual information can be provided by monitoring user GUI activity, which can provide information regarding end-user provided destinations, via URLs, and restricting network activity to benign destinations by requiring destinations to be associated with its corresponding GUI activity. We see the usefulness of said information from the results of class 3 whitelists, with false-positive rates and false negative rates between 0% and 0.4% and 0% and 3.6% respectively across applications and experiments.

6.8 Direct Attacks on Harbinger

Harbinger components are secure from direct attacks under our threat model because the adversary only operates with user-level privileges. As a result, the adversary is not able to directly tamper with or stop admin-level and kernel-level processes. Specifically, the Network Monitor is a kernel-level process, and the Application Monitor is an admin-level process, so they cannot be compromised by the adversary. The Policy Engine is located on a remote server, so malware residing on the host system is not able to compromise it as it would need to initiate network traffic to do so. In which case, Harbinger will detect the adversary.

7 Performance Evaluation

We examine the performance of Harbinger by evaluating 1) the end-to-end delay introduced by the controller, which includes the time required to consult the policy engine, and 2) the stress the application monitor puts on the end-user's system while monitoring user behavior and collecting GUI information. Our experiments showed that end-to-end delay introduced by the need to consult the controller is negligible with median delay less than 3 milliseconds, and the application monitor does not introduce noticeable overhead to the end-user with delays less than 65 milliseconds for the most complicate application we tested. We describe the experiments and the outcome more in detail below.

7.1 Network Monitor and Policy Engine

We use ALE layer in WFP to intercept network packets at the host's kernel. We use ALE to intercept packets in a flow and make decision per flow, not per packet. As a result, once we approve a flow, we do not need to classify all subsequent packets belonging to that flow. Our network monitor elevates each flow to the policy engine for a decision and enforces the decision locally in the host's ALE classifier. As a result,

the overhead associated with Harbinger concerns only the first packet of the flow and its query to the policy engine.

The overhead associated with network monitor requires us to monitor end-to-end latency introduced by Harbinger, which is the time between when the first packet of a flow reaches the network monitor to the time spend waiting a response from the policy engine. As a result, we can consider the end-to-end latency as the sum of the delay introduced by the network monitor and the policy engine. The local processing time of the first packet of a flow includes queuing the packet, adding an OpenFlow header, adding contextual information, updating the status of the flow once receiving the policy engine's decision, and reinjecting the packet back into the packet queue. These operations are not computationally intensive, proven by our end-to-end delay results shown below. The delay associated with querying the policy engine, which is located on a separate machine, depends on network conditions and time spent matching policies at the policy engine. The delay will be independent of the origin of the packet. We conduct the experiment using an in-LAN policy engine because this will most closely resemble enterprise conditions.

To measure the performance results introduced by the network monitor and the policy engine, we used the `KeQueryPerformanceCounter` function, which is a kernel-level routine used to acquire high-resolution timing. The `KeQueryPerformanceCounter` routine returns the current value of the performance counter since the system was booted. The routine takes one argument and fills the argument with the frequency of the performance counter. Once we retrieve both values, we divide the performance counter by the frequency counter. Doing so returns the elapsed time since the system was booted, with an accuracy of 100 nanoseconds or roughly 0.1-microsecond resolution. Since we only measure the performance of the network monitor, we exclude all interactions the network monitor may have with the application monitor to a simple return statement. We use the experimental environment as described in Section 5. When measuring performance, we interacted with Microsoft Word, Microsoft Excel, Microsoft PowerPoint, Microsoft Visual Studio, Google Chrome, Internet Explorer, Mozilla, Firefox, Slack, Skype, and PyCharm.

Figure 4 shows the experiment results of the end-to-end delay introduced by the controller in the form of a cumulative distribution function (CDF). Furthermore, the two lines shown in the CDF represent cases in which a GUI was present and not. The end-to-end delay for both cases when there is an absence, and the presence of GUI was under 3 milliseconds for the majority of the flows, with all flows with an empty GUI traces completed under 6 milliseconds, and 95% of those with GUI traces completed under 6 milliseconds. These overheads are not enough to be perceivable by the end-user [20].

We ran a total of 653,189 trials in which the network monitor intercepted the first flow of the packet, queried the policy engine for a verdict, and reinjected the packet into the queue. We excluded the 0.17% of the data, which is 1,125 of the trials, because they exceeded the mean by a factor of six, thus considered outliers. It is unlikely for the network monitor the cause these situations because the outliers constitute less than 0.5% of the trials. Thus, we conclude that poor network conditions to be the likely cause. From the experimental results, we observed the end-to-end delay to be roughly 835 miscroseconds. The biggest

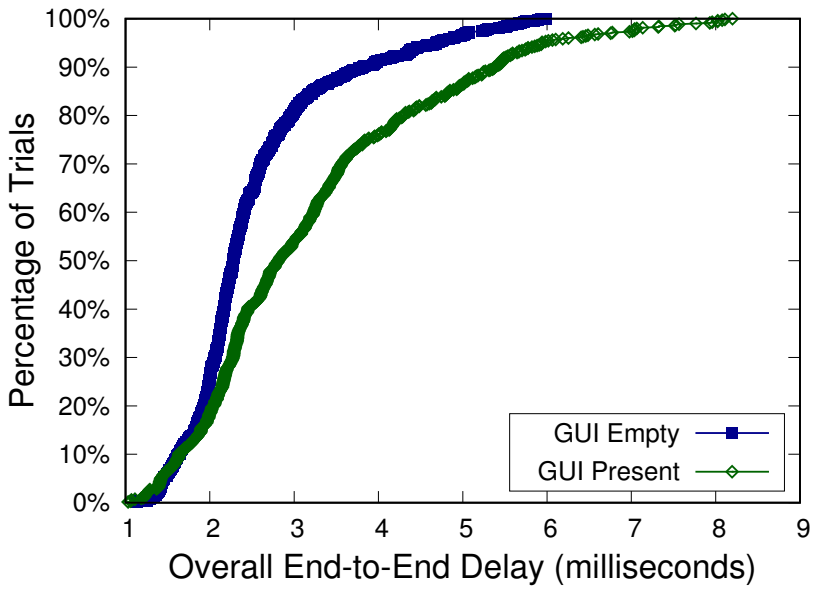

Figure 4: CDF of end-to-end delay introduced by Harbinger on new flows. There were 743 (43 outliers removed) traces with GUI information and 765 (34 outliers removed) traces with no GUI information.

contributing factors to the delay were the encryption of the response, decryption of the query, parsing GUI contextual information, and finding a matching policy rule. They consistute 290 microseconds, 159 microseconds, 117 microseconds, and 76 microseconds, respectively. The policy engine currently constitutes 31 rules, and with the addition of more rules, the consultation delay may increase.

7.2 Application Monitor

Harbingers's application monitor introduces two types of overheads: 1) the overhead introduced when monitoring UI activity 2) and the overhead introduced when the network monitor requests UI information. The overhead when monitoring UI activity is caused by hooks and event handlers, which are triggered when the end-user interacts with the system's GUI. While the event handler runs concurrently with the application, thus not affecting user experience, the mouse and keyboard hooks start and finish execution before the application, so a high latency could hinder user experience.

To measure the performance of each hook and event handler, we used the C++ high-resolution clock API, which can measure time in nanoseconds. When monitoring UI activity, the hooks and the event handler traverses the GUI tree upwards from the activated element the root element. As a result, the overhead of UI activity monitoring may be highly dependent on the depth and complexity of the GUI tree and the library used for constructing the applications GUI. As a result, we decided to test 3 applications 1) Notepad, a text editor with a simple GUI, 2) Notepad++, a text editor with a moderately complex GUI, and 3) Microsoft Word, a word processor with a complex GUI. Furthermore, comparing these applications seem fair because

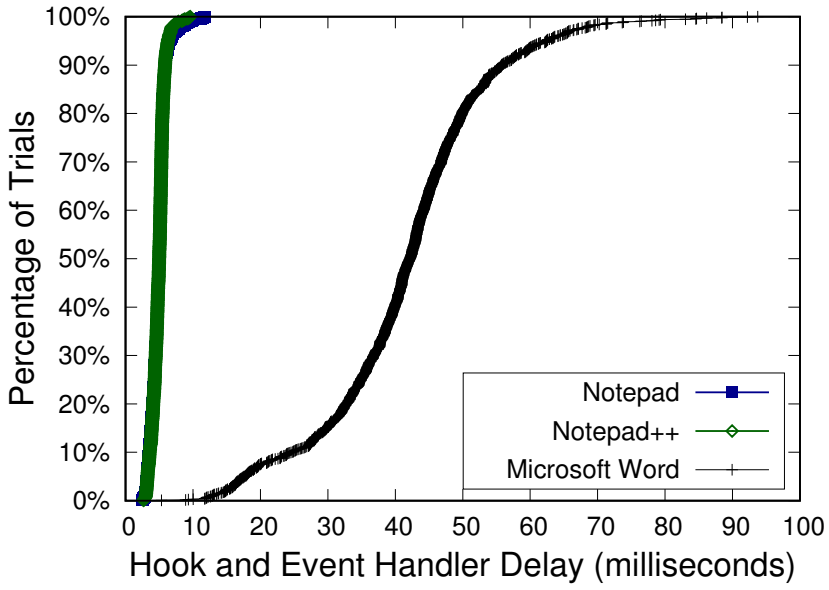

Figure 5: CDF of the time required for the AM to retrieve GUI information from applications. The Notepad and Notepad++ lines are mostly overlapping in the graph. The Notepad application had 1,731 runs, with 30 outliers removed. Notepad++ had 1,191 runs, with 63 outliers removed. Microsoft Word had 1,310 runs, with 1 outlier removed.

their primary purpose is editing text. In each application, we monitor and save a minimum of 1,191 GUI traces.

In Figure 5, we show the results of the experiment of each application. As can be seen from the Figure, both notepad and notepad++'s hooks and event handlers completed the acquisition of GUI elements under 12 milliseconds. However, for Microsoft Word, the acquisition took under 50 milliseconds for 80% of the GUI traces and under 65 milliseconds for over 95% of the GUI traces. From the results, we observe that the overhead is not affected by the complexity of the GUI but depends on what library the developers used when constructing the application's GUI. We observe this behavior from the fact that both notepad and notepad++ used the same GUI library, but notepad++'s has a more complex GUI than notepad. Common sense would lead us to believe that notepad++ should have higher latency in GUI acquisition, but in actuality, the overhead for both applications was similar. On the other hand, Microsoft Word's GUI, developed with a different GUI library, has a larger latency in GUI acquisition than either notepad and notepad++. Nonetheless, for even the most complex application, Microsoft Word, GUI elements can be quickly retrieved.

The other overhead introduced by the application monitor occurs when the network monitor requests GUI information from the application monitor. The application monitor stores GUI information in a vector class provided by the C++ stand library, so retrieval of GUI information by the network monitor is application agnostic. We used the C++ high-resolution API clock to measure the overhead. We ran 1,383 trials, and

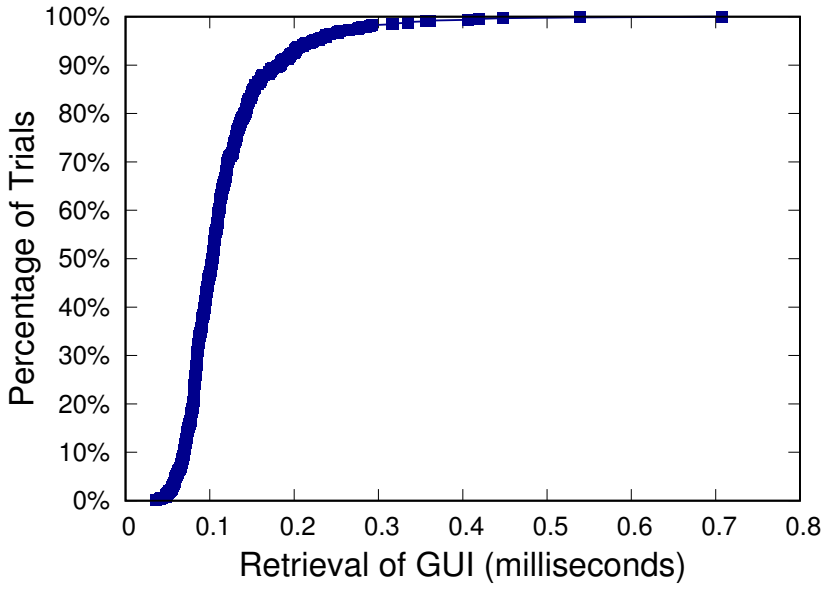

Figure 6: CDF of the time required to retrieve GUI element cached GUI elements across 1,383 trials. We removed 4 outliers.

we show the results in Figure 6. We see that 95% of requests completed under 0.23 milliseconds.

7.3 Performance Evaluation Results

From the experiment, we see that Harbinger introduces small amounts of delay when it comes to both the network monitor and application monitor that is imperceptible to the end-user. The end-to-end delay of acquiring saved GUI information, consulting the network monitor, and enacting the decision of the network monitor is under 6 milliseconds in 95% of the cases. The application monitor's overhead when monitoring and acquiring GUI information while the end-user interacts with the GUI of event complex applications takes under 65 milliseconds in 95% of the cases. With such overheads, it will not hinder network traffic nor the end-user's experience when interacting with the GUI.

8 Discussion and Limitations

In this section, we discuss the limitations of Harbinger and further possible improvements. Specifically, we discuss generalizing our approach to other operating systems and applications; furthermore, we discuss limitation of inferring network activity in-depth.

8.1 Generalization and Limitations

We developed Harbinger for Microsoft Windows due to its ubiquitous use in enterprise settings. With the capabilities of UI Automation framework, we were able to monitor UI activity without extensive engineering effort. For other operating systems, we have not found a similar framework; as a result, to generalize our approach across operating systems, instrumentation of GUI libraries may be required. We do not believe this is a major concern as the market share of Windows operating systems is 80% followed by macOS with 20%, Linux around with 2%, and the rest shared among less common operating systems. The only limitation we experience with UI Automation is its inability to acquire UI elements from applications developed in Java due to Java applications using the Java Swing library to develop its GUI. To instrument Java applications, we would probably need to use accessibility framework developed for Java applications.

For performance measurements of the Application Monitor, we have tested three text editing applications. When using UI Automation, we have not experienced noticeable delays with the following applications: Word, Excel, PowerPoint, Notepad, Pycharm, Chrome, Edge, Firefox. However, we believe that further testing is neccesary to support our claim with data. We do not believe testing every application in Windows is necessary, but testing a chosen few developed in GUI libraries that represent a vast portion of applications will suffice. Our performance of the Network Monitor should closely resemble the deployment in enterprise settings, and the only significant change might be the number of polices the Policy Engine might need to consult.

8.2 Inference

Harbinger uses temporal proximity of UI actions and their corresponding network activity to infer causal links. However, direct observation may be necessary for our approach to work with applications that use helper processes to delegate network activity. For example, some applications may use Microsoft Windows's built-int DNS functionality to resolve DNS lookups, which the Service Host process (svchost.exe) provides. As a result, there is no way to infer the causal link between an application and its DNS lookups. To tackle this problem, we would need to observe the inter-process communication between client and service processes directly. With the appropriate instrumentation, it is feasible but requires significant engineering efforts, as clients may use a multitude of inter-process communication methods including but not limited to ports, pipes, and shared memory. Furthermore, such an instrumentation would introduce further complexity and performance overheads.

9 Conclusion

In this thesis, we have explored the usability of dynamically created whitelists for classifying network traffic. We have found that with the help of GUI contextual information, we were able to support user-generated network traffic that was previously unseen and create more fine-grained policies. Furthermore, our approach

was feasible with moderate engineering efforts and capable of cross-application support on Windows operating systems. From our experiments, we found that our approach introduced minimal overhead with false positive and false negative rates better than traditional firewalls using the same greedy algorithm.

References

[1] Antoniewicz, B. Windows DLL injection basics. http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html, January 2013.

[2] Artzi, S., Dolby, J., Jensen, S. H., Møller, A., and Tip, F. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), ACM, pp. 571–580.

[3] BeyondTrust. External Attacks and Privileged Accounts: 5 Steps to Control the Threat Potential. https://www.beyondtrust.com/resources/whitepapers/external-attacks-privileged-accounts-5-steps-control-threat-potential, Dec 2019.

[4] Bhukya, W. N., Kommuru, S. K., and Negi, A. Masquerade detection based upon GUI user profiling in Linux systems. In *Annual Asian Computing Science Conference* (2007).

[5] Brian Krebs. Krebs on Security. https://krebsonsecurity.com/2019/05/first-american-financial-corp-leaked-hundreds-of-millions-of-title-insurance-records/, Dec 2019.

[6] Chang, T.-H., Yeh, T., and Miller, R. C. Gui testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2010), ACM, pp. 1535–1544.

[7] Cui, W., Katz, R. H., and Tan, W.-t. BINDER: An extrusion-based break-in detector for personal computers. In *USENIX Annual Technical Conference* (2005).

[8] Cybersecurity and Infrastructure Security Agency. Least Privilege. https://www.us-cert.gov/bsi/articles/knowledge/principles/least-privilege, Dec 2019.

[9] Fink, G. A., Duggirala, V., Correa, R., and North, C. Bridging the host-network divide: Survey, taxonomy, and solution. In *Proceedings of the 20th Conference on Large Installation System Administration* (Berkeley, CA, USA, 2006), LISA '06, USENIX Association, pp. 20–20.

[10] Garfinkel, T., Rosenblum, M., et al. A virtual machine introspection based architecture for intrusion detection. In *Ndss* (2003), vol. 3, pp. 191–206.

[11] IBM Security, Ponemon Institute. Cost of Data Breach Study: Global Overview. https://www.ibm.com/downloads/cas/ZYKLN2E3, Dec 2017.

[12] Jang, Y., Chung, S. P., Payne, B. D., and Lee, W. Gyrus: A framework for user-intent monitoring of text-based networked applications. In *Network and Distributed System Security Symposium (NDSS)* (2014).

[13] Kwon, J., Lee, J., and Lee, H. Hidden bot detection by tracing non-human generated traffic at the zombie host. In *International Conference on Information Security Practice and Experience* (2011).

[14] Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., and Kirda, E. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 399–412.

[15] Lee, W., Stolfo, S. J., et al. Data mining approaches for intrusion detection. In *USENIX Security Symposium* (1998), San Antonio, TX, pp. 79–93.

[16] Memon, A., Banerjee, I., Nguyen, B. N., and Robbins, B. The first decade of gui ripping: Extensions, applications, and broader impacts. In *Reverse Engineering (WCRE), 2013 20th Working Conference on* (2013), IEEE, pp. 11–20.

[17] Mesbah, A., Bozdag, E., and Van Deursen, A. Crawling ajax by inferring user interface state changes. In *Web Engineering, 2008. ICWE'08. Eighth International Conference on* (2008), IEEE, pp. 122–134.

[18] Microsoft Corporation. UI Automation. https://docs.microsoft.com/en-us/windows/desktop/winauto/entry-uiauto-win32, May 2018.

[19] Morgado, I. C., Paiva, A. C., and Faria, J. P. Dynamic reverse engineering of graphical user interfaces. *International Journal On Advances in So ware* (2012).

[20] Nielsen Norman Group. Response Time Limits: Article by Jakob Nielsen. https://www.nngroup.com/articles/response-times-3-important-limits/, Dec 2019.

[21] Norton. 2019 Data Breaches: 4 Billion Records Breached So Far. https://us.norton.com/internetsecurity-emerging-threats-2019-data-breaches.html, Dec 2019.

[22] NumPy. NumPy. https://numpy.org/, Dec 2019.

[23] Open Networking Foundation. OpenFlow Switch Specification. https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf, Dec 2019.

[24] Open Systems Resources, Inc. Kernel: Calling user mode - using the inverted call model. http://www.osronline.com/article.cfm?id=94, August 2002.

[25] Oracle Corporation. Java Accessibility Framework. https://docs.oracle.com/javase/10/access/JSACC.pdf, May 2018.

[26] Paxson, V. Bro: a system for detecting network intruders in real-time. *Computer networks 31*, 23-24 (1999), 2435–2463.

[27] ProPublica. The Extortion Economy: How Insurance Companies Are Fueling a Rise in Ransomware Attacks. https://www.propublica.org/article/the-extortion-economy-how-insurance-companies-are-fueling-a-rise-in-ransomware-attacks, Dec 2019.

[28] Python.org. 7.2. re - Regular expression operations. https://docs.python.org/2/library/re.html, Dec 2019.

[29] Roesch, M., et al. Snort: Lightweight intrusion detection for networks. In *Lisa* (1999), vol. 99, pp. 229–238.

[30] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 513–528.

[31] Shirley, J., and Evans, D. The user is not the enemy: Fighting malware by tracking user intentions. In *New Security Paradigms Workshop* (2008).

[32] Sikuli Developers. Sikuli script. http://www.sikuli.org/, September 2018.

[33] Silva, J. C., Silva, C., Gonçalo, R. D., Saraiva, J., and Campos, J. C. The guisurfer tool: towards a language independent approach to reverse engineering gui code. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems* (2010), ACM, pp. 181–186.

[34] Stiegler, M., Karp, A. H., Yee, K.-P., Close, T., and Miller, M. S. Polaris: virus-safe computing for windows xp. *Communications of the ACM* (2006).

[35] Wagner, D., and Soto, P. Mimicry attacks on host-based intrusion detection systems. In *ACM Conference on Computer and Communications Security* (2002), ACM.

[36] Wes McKinney. Python Data Analysis Library. https://pandas.pydata.org/, Dec 2019.

[37] Yeung, D.-Y., and Ding, Y. Host-based intrusion detection using dynamic and static behavioral models. *Pattern recognition 36*, 1 (2003), 229–243.