The Pennsylvania State University
The Graduate School

BUILDING AN EVENT DRIVEN ATTACK GRAPH FRAMEWORK

A Thesis in
Computer Science and Engineering
by
Rahul Titus George

© 2021 Rahul Titus George

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2021

The thesis of Rahul Titus George was reviewed and approved by the following:

Trent Jaeger
Professor of Computer Science and Engineering
Thesis Advisor

Tom LaPorta
Professor of Computer Science and Engineering

Chita R. Das
Distinguished Professor
Head of the Graduate Programe and CSE Department Head

Abstract

Existing intrusion detection systems fail to detect unknown attacks, zero-day vulnerabilities and sophisticated multi layer attacks. This is because current intrusion detection systems lack visibility into all system components, specifically programs. They focus on known attacks/vulnerabilities, limiting their ability to detect unknown attacks. As a result they fail to track the evolution of attacks across multiple layers (network, host and program). We envision using modular component attack graphs for intrusion detection. Attack graphs have been used earlier in networks for risk analysis, reliability analysis. They've been employed in a reactive manner. We propose using program attack graphs for better visibility into programs and proactive intrusion detection (removing the dependency on known vulnerabilities or exploits). We identify program analyses needed to compute and connect attack surfaces, a source interface in a compute which may receive adversarial input, to attack states, flaws or security property violations, which grant the adversary privileges that create threats, and actions, operations that exploit those threats. We model two security properties in our program attack graphs: memory safety and information flow. We also identify and develop techniques to propagate these attack actions, as exploit operations may lead to subsequent safety property violations. We implement a framework which is able to automatically compute program attack graphs using these techniques and track the possible evolution of potential attacks across components through dynamic events. We also evaluate the efficacy of existing intrusion detection systems through a case study on the Shellshock vulnerabilities. We construct program attack graphs for multiple programs and were able to capture two known vulnerabilities. We estimate how our program attack graphs may evolve and illustrate the evolution for a specific program through a scenario. We simulate input surface expansion events for a few programs and record how they affect the attack graph.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like thank all my fellow researchers in the SIIS Lab, specially Frank Capobianco and Kaiming Huang, for their contributions to this research. I would like to specially thank my advisor Dr.Jaeger for his invaluable guidance. I would also like to thanks Dr.LaPorta for his suggestions that significantly helped the quality of my thesis. This research was sponsored by the Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and National Science Foundation grant CNS-1801534. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not withstanding any copyright notation here on.

Chapter 1
Introduction

Prolific increase in zero-day attacks against computer systems indicates the failure of current defenses employed. In addition sophistication of the attacks also continue to increase. Intrusion detection systems are a commonly deployed defense to detect such attacks. Intrusion detection systems can be classified into two types namely misuse and anomaly based. Anomaly based intrusion detection systems attempt to define a baseline normal behavior and flag any behaviour that deviates from the defined normal. Misuse or signature based intrusion detection systems leverage known attacks. Additionally, intrusion detection systems may be classified as host based, network based or hybrid depending on where they operate (at which layer). NIDS deployments, such as Bro [1], Snort [2] and Suricata [3], monitor packets at one or more network monitoring locations to detect network-borne attacks. Current HIDS solutions, such as OSSEC [4] and Samhain [5], examine system operation logs for behaviors often associated with attacks, such as root logins and sensitive resource modifications. Hybrid intrusion detection systems integrate network and system monitoring to correlate log entries from both layers to improve attack detection confidence.

Network intrusion detection systems [1, 2] lack visibility into the host and programs. They rely on known vulnerabilities through the use of scanners [6, 7]. Anomaly based host intrusion detection systems [8, 9] suffer from false positives and are even susceptible to evasive attacks [10, 11] based on the mechanism they use. Misuse based host intrusion detection system [4, 5] analyze system logs using rules manually created based on known attacks. While correlating network and host operations can improve attack detection, hybrid IDS' [12, 13] still focus on known attack behaviors and lack visibility into programs. All these intrusion detection systems lack visibility into all system components, specifically programs. As a result they fail to track the evolution of attacks across multiple layers (network, host and program). Their focus on known attacks/vulnerabilities limits their

ability to detect unknown attacks.

Attack graphs have been widely employed in computer networks mostly in a reactive manner though such as for vulnerability analysis. The typical process for generating attack graphs is as follows. First, vulnerabilities of individual hosts are determined using scanning tools, such as COPS [7] and Nessus Scanner [6]. Using this local vulnerability information along with other information about the network, such as connectivity between hosts, attack graphs are then produced. Subsequent work [14, 15] explored using attack graphs for intrusion detection. However, these attack graph approaches have several caveats. First, these attack graphs are primarily focused on the network layer and therefore lack visibility into all system components, specifically programs. As these approaches rely on known vulnerabilities they are limited in their ability to detect unknown attacks. Second, they fail to handle evolution of the attack graph (dynamic changes) due to their limited visibility. These include changes to the component or something detected at run time which was missed in the attack graph. Third, scalability remains an open issue in the construction of attack graphs using these approaches.

We propose using modular component attack graphs for intrusion detection. This provides a more systemic view. The insight here is that attack graphs naturally correspond to the intrusion detection steps of acquiring threatening permissions and leveraging them in operations to perform exploits. There are challenges in using modular component attack graphs for intrusion detection. Intrusion detection aims to be performed at run time, which limits the budget one can devote to attack graph analysis and upkeep. This could include both computational resources and storage resources available. Scalability of these modular component attack graphs is an open issue. Additionally, the attack graphs may need to be updated. There may be potential attack paths across components and across different layers through input and output attack surfaces. We would rely on run-time monitors to make these inter component connections. In addition the underlying component may also change. The attack graphs need to be updated due to these dynamic changes. We need a mechanism to update the component attack graphs appropriately as per the dynamic change in a principled fashion. Additionally, we might require incremental analyses to update the attack graphs to keep the overhead reasonable.

We implement a framework to automatically compute program attack graphs. The framework can track the possible evolution of the attack graph across layers through dynamic events. We identify and develop program analyses as needed for computing program attack graphs. We use LLVM [16] passes for our program analyses. We support events through an event plane and an analysis plane. We use a single model graph

database, Neo4j [17] to store/persist the attack graph and the underlying component graph. We identify the attributes needed for each attack graph node and for each program component graph node (program dependence graph node). We create indexes on important attributes to speedup performance. We modify the analyses necessary to interact with the database. We identify the program analyses needed to compute and connect attack surfaces, states and actions. Attack surfaces are source interfaces in a compute which may receive adversarial input. For example, an open system call or a socket. Attack states are flaws or security property violations, which grant the adversary privileges. We model two security properties in our program attack graphs: memory safety and information flow. For example, an information flow violation would be the adversary's ability to corrupt some important piece of data in the program. These states (security property violations) create potential threats. Attack actions are operations the adversary may use to exploit the privileges gained to further escalate his privileges. We also identify and develop techniques to propagate these attack actions, as exploit operations may lead to subsequent security property violations. To motivate the problem we evaluate the efficacy of existing intrusion detection systems through a case study on the shellshock vulnerabilities. We automatically construct program attack graphs for five programs. We estimate how these program attack graphs may evolve and illustrate the evolution for a specific program through a scenario. We simulate input surface expansion events for a few programs and record how they affect the attack graph.

On average we identified 8 potential attack surfaces, 44 information flow violation states and 300 memory violation states in each program. In addition, we found 36 attack actions (including 6 unsafe pointer actions) on average. We find that though security property violations, specifically memory safety, are common in programs exploit operations are comparatively limited. We can see that the number of memory safety violation states (unsafe pointers) are larger than the information flow violation states. (This may be because the memory violation states correspond to IR variables in SSA form rather than at source code level.) However, we find that the exploit operations (unsafe pointer actions) that can leverage these violations are scarce. Our computed program attack graphs demonstrate that component (program) attack graphs may be generated automatically without depending on known vulnerabilities in a principled manner. We argue that our approach is principled in that it finds security property violations (which grant privileges) and operations which exploit these privileges to gain new privileges. Our program attack graphs capture potential attack paths without depending on known vulnerabilities making them suitable for intrusion detection. Our

analyses tend to over-approximate and this may affect scalability. We acknowledge that scalability remains an open issue.

We found on average that there may be 77 maximum input surface expansion events and 12 possible output surface expansion events in a program. This illustrates how often program attack graphs may expand and possibly lead to inter component attack paths. We envision using run-time monitors to detect these changes and thereby track the evolution of such attacks. A program attack graph may expand due to changes such as detecting untrusted input resulting in a new input attack surface, or even a change in the underlying component. It is important to track these changes as they may lead to new attack paths. We simulate the input surface expansion event to record how a new input attack surface may affect the attack graph. We found that on average a new input attack surface may lead to 50 new potential paths, 9 new states and 8 new actions. New states and actions refer to new reachable states and actions, which are reachable only from the new surface. Attack graph expansion is based on the dynamic change and the semantics of the program. It may or may not find new states and actions. This demonstrates how a dynamic change such as a new input attack surface may lead to new attack paths and even new exploit operations.

This thesis makes the following contributions.

1. We propose using modular component attack graphs for intrusion detection. We identify and develop techniques needed to compute program attack graphs. These program attack graphs provide better visibility into programs and are not dependent on known vulnerabilities.

2. We develop a framework for the automatic computation of program attack graphs. Additionally, the framework can track the evolution of the potential attacks across components through modular attack graphs using dynamic events.

3. We automatically construct program attack graphs for five programs, two of which had known vulnerabilities (CVEs). We were able to capture known vulnerabilities as attack paths in our program attack graphs. We estimate how our program attack graphs may evolve through the input surface expansion and output surface expansion events. We simulate input surface expansion events for a few programs and record how they affect the attack graph.

Chapter 2
Background

2.1 Intrusion Detection

NIST [18] describes an intrusion as an attempt to compromise CIA (confidentiality, integrity or availability), or to bypass the security mechanisms of a host or network. Intrusion detection is the process of detecting such actions in a system. Intrusion detection systems can broadly be classified into two types namely misuse and anomaly based. Additionally, intrusion detection systems may be classified as host based, network based or hybrid depending on where they operate (at which layer). Anomaly based intrusion detection systems attempt to define a baseline normal behavior and flag any behaviour that deviates from the same. Misuse or signature based intrusion detection systems leverage known attacks. This could be based on the corrupt packet structure from an earlier attack, or a known sequence of system calls used in well known attacks. Activities that match these patterns are classified as attacks. In reality, neither approach is perfect. Misuse detection can never identify all possible attacks. Likewise, it is a non trivial task to define "good" behaviour for anomaly detection. An overly conservative definition would lead to several false positives and this is one of the primary caveats of this approach. On the other hand a weak definition would result in failure to detect legitimate attacks.

2.1.1 Anomaly based detection

These anomaly based detection systems could further be classified into three main categories [19] based on methodology namely statistical based, data-mining based, and machine learning based. In statistical methods for anomaly detection, the system observes the activity of subjects and generates profiles to represent their behavior. The profile

typically includes various features such as CPU usage, packets received and programs executed. Two profiles are computed for each subject [1]: the current profile and the stored profile. As various system or network events occur such as audit log records, incoming packets are processed, the intrusion detection system updates the current profile and periodically calculates an anomaly score (indicating the deviation from normal behavior) by comparing the current profile with the stored profile. If the anomaly score is higher than a certain threshold, the intrusion detection system generates an alert. Haystack [9] is one of the earliest examples of a statistical anomaly-based intrusion detection system. It used audit trails and employed both user and group-based anomaly detection strategies. It maintained a database of user groups and individual profiles for the aforementioned purpose. If a user had not previously been detected, a new user profile with minimal capabilities was created using restrictions based on the user's group membership. Subsequent work, IDES [20], developed a real time intrusion detection expert system. It monitored the activities of individual users, groups, remote hosts and entire systems, to detect suspected security violations, by both insiders and outsiders, as they occur. As the analysis methodologies developed for IDES matured, scientists at SRI developed an improved version of IDES called the Next-Generation Intrusion Detection Expert System (NIDES) [21]. Interestingly both systems also supported including rules to detect for known attacks in combination with the user profiles for statistical based anomaly detection. Statistical Packet Anomaly Detection Engine (SPADE) [22] is a statistical anomaly detection system that is available as a plug-in for SNORT [2]. It detects anomalous packets by maintaining probability tables that contain information regarding the number of occurrences of different kinds of packets seen. As stated earlier it is a non trivial task to define "normal" behavior. In an attempt to better define normal behavior, Maxion and Feather [23] characterized the normal behavior in a network by using different templates that were derived by taking the standard deviations of ethernet load and packet count at various periods in time. However, their goal was to identify network faults which may or may not be adversarial.

Machine learning methods for anomaly detection, unlike statistical approaches which tend to focus on understanding the process that generated the data, focus on building a system that improves its performance based on previous results. In other words systems that are based on the machine learning paradigm have the ability to change their execution strategy on the basis of newly acquired information, they "learn". In a seminal paper, Forrest et al. [24] proposed building a profile for programs that run as root based

[1] A subject is an active entity that requests access to a resource or the data within a resource.

on the sequence of system calls used. It leverages the sliding window method. Programs that deviated from these normal sequence profile would then be considered victims of an attack. Subsequent work [8, 25, 26] has built on this general seminal approach. However, Wagner et al. [11] subsequently showed how these anomaly based host intrusion detection systems [8, 24–26] are susceptible to mimicry attacks. There is also a body of work that has explored using other machine learning methods such as Bayesian networks [27], Markvov chains [28, 29] and Principal Component Analysis(PCA) [30] in anomaly detection. A general drawback of machine learning based anomaly detection is the fact that the machine learning method itself might be vulnerable due to adversarial machine learning [10]. These techniques tend to be computationally expensive [19] as well. Data mining [31] has been defined as being "concerned with uncovering patterns, associations, changes, anomalies, and statistically significant structures and events in data". Therefore, leveraging these techniques can help improve the process of intrusion detection by adding a level of focus to anomaly detection. There is a significant body of work that employs data mining techniques such as neural networks [32, 33], support vector machines (SVM) [34] and clustering [35, 36]. These techniques often have a very high false positive rate [19].

2.1.2 Misuse detection

The second approach, based on methodology, used in intrusion detection is misuse detection. In this approach systems tend to look for signatures or leverage knowledge corresponding to known attacks or vulnerabilities. This is why it is often referred to as signature based or knowledge based intrusion detection. These systems look for patterns that match existing or known attacks. They look for sequences of steps or operations that match an attack signature. They fail to identify unknown or zero day vulnerabilities as a result. It is non trivial to write good quality signatures [1]. A signature should ideally be able to detect all possible variations of a pertinent attack. Host intrusion detection systems such as OSSEC [4] and Samhain [5] rely on system logs. OSSEC for instance analyzes the log to identify potential misuses and identifies potential rootkits on the system. It employs a catch-all rule wherein it looks for well known keywords in the system log such as segmentation fault. An expert would have to analyze these or rules modelling known attacks could be applied when analyzing these logs. Additionally, OSSEC supports user defined rules for log analysis to model attack signatures. Similarly, Samhain monitors logs and also checks file integrity. It also performs rootkit detection, port monitoring. File integrity checking is known to suffer from false positives.

Network intrusion detection systems tend to employ more signature based approaches compared to host intrusion detection system. Initial signature-based NIDS suffered from false positives as they were matching signatures(fixed length strings) without considering the context [37]. There is a significant amount of work done to help improve the signatures used. One such technique is signature enhancement, where network context information is added to the signatures, thus each signature is now stand alone and carries complete context information when the signature is valid and applicable. Sommer and Paxson [38] had developed an approach to enhance signatures with context information (contextual signatures). The contextual low level information is provided in the form of regular expressions. A regular expressions based approach is more flexible (expressive) for matching compared to the fixed strings approach. Additional information can be attached to the signature for verification since algebraic operations can be performed on the regular expressions. Similarly, an object oriented approach [39] to include contextual information with signatures was also proposed. Snort [2], a well known NIDS, also uses context aware signatures.

Conventional signature-based NIDS analyzed individual network packets and tried matching them against a database of signatures. These systems would fail to detect attacks which spanned multiple packets. Stateful signatures [37, 40] based approaches were proposed to address this. These signatures analyzed only relevant portion of the traffic rather than the entire traffic. For example, to deal with a known authentication attack the system would use a stateful signature to check traffic only during the login session. Known multi-step attacks and scenarios could be specified through a stateful signature. However, adversaries could still evade these signature based intrusion detection systems. These techniques were still exploit focused. There are several ways to exploit a vulnerability and new exploits would evade detection by these systems. In order to deal with this vulnerability signature based detection [41–44] was proposed.

2.2 Prior Attack Graph Approaches

Attack graphs have been widely employed in computer networks mostly for vulnerability analysis rather than intrusion detection. The typical process for vulnerability analysis of a network proceeds as follows. First, vulnerabilities of individual hosts are determined using scanning tools, such as COPS [7] and Nessus Scanner [6]. Using this local vulnerability information along with other information about the network, such as connectivity between hosts, attack graphs are then produced. Each path in an attack graph is a series of

exploits, which are called atomic attacks, that leads to an undesirable state, e.g., a state where an intruder has obtained administrative access to a critical host. Further analyses is then performed, such as risk analysis [45], reliability analysis [46], or shortest path analysis [47], on the attack graph to assess the overall vulnerability of the network. Constructing attack graphs is a crucial part of doing vulnerability analysis of a network of hosts. Construction by hand, however, is tedious, error-prone, and impractical for attack graphs larger than a hundred nodes.

The earliest work related to attack graphs is a paper by Dacier and Deswarte [48]. In this work the authors proposed using a privilege graph to extend the TAM (Typed Access Matrix) model to detect violations of the safety problem [49] in more realistic protection systems. They do this by identifying which privilege transfers could lead to unsafe protection states. Subsequent work [50] extended this approach by associating arcs or privilege transfers in the privilege graph, with vulnerability classes. They built a tool, ASA, to do the same. The aim of this work was to help system administrators in monitoring the security of the systems. The administrator has to decide which vulnerabilities (arcs) in the privilege graph can be ignored and which ones must be eliminated. To help the administrator make a more informed decision the authors propose, define and estimate two variables to evaluate the security risks induced by the vulnerabilities . The two variables are time and effort. They estimate the mean time and effort by transforming the privilege graph into a Markov model.

Phillips and Swiler [51] developed an attack graph model for networks in 1998. They constructed attack graphs using three inputs: a database of common attacks, "attack templates", configuration information and an attacker profile. This approach goes beyond scanning tools as they consider the network topology in conjunction with a set of attacks. The attacker profile captures assumptions about the attacker's initial capabilities. The configuration file contains information relevant to the host operating system, router configuration and network topology. Interestingly, they advocated generating network flows from configurations. The "attack templates" represent generic steps in known attacks, including preconditions necessary for the attack. They incorporated key elements of the attack graph, such as preconditions, actions (edges in the template) and post-conditions. Additionally, they proposed using edge weights to represent the attacker's success probability, or cost to attack and average time to succeed.

Subsequent research then explored methods to compute attack scenarios from attack graphs. Sheyner et al. developed a model checking approach to compute attack graphs [52]. They model the network as a finite state machine, where state transitions correspond

to atomic attacks launched by the intruder. If a safety property fails, the authors provide a mechanism, to produce an attack graph, which includes all of the counterexamples that can be analyzed, thereby allowing the analyst to understand all possible attack scenarios. Additionally, they described two ways of analyzing these attack graphs. First, an algorithm to compute the minimal critical set of atomic attacks which would guarantee the intruder's failure (failure to reach a specific desired state). Second, a probabilistic reliability analysis that determines the likelihood of the intruder's success. However, it is well known that model checkers suffer from scalability problems, and there is good reason to doubt whether a model checker can handle directly a realistic set of exploits for even a modest-sized network. Ammann et al. [53] proposed a more compact and scalable representation. This relies on a critical assumption of monotonicity, which, in essence, states that the precondition of a given exploit is never invalidated by the successful application of another exploit. In other words, the attacker never needs to backtrack. This assumption reduces the complexity of the analysis problem from exponential to polynomial, thereby bringing even very large networks within reach of analysis. Ou et al. [54] further optimized the representation to make the attack graph independent of hosts and made it more scaleable. They propose logical attack graphs, which directly illustrate logical dependencies among attack goals and configuration information. The attack graph generation tool builds upon MulVAL [55], a network security analyzer based on logical programming. The logical attack graph presented in this work was found to be a a special case of Schneier's attack tree [56].

2.3 Motivation

We present a case study to motivate the need for better intrusion detection and to illustrate the caveats of current intrusion detection systems. We studied and evaluated the efficacy of two existing intrusion detection systems with respect to the Shellshock vulnerability and all its variants. We conducted experiments using these intrusion detection systems by triggering Shellshock attacks on Cybervan, an emulation test bed. We setup OSSEC [4], a host based intrusion detection system which relies on misuse rules based on the system log, and Seconion, a network intrusion detection system, which employs Snort [2] rules to detect malicious packets. Shellshock refers to a group of exploits or bugs in UNIX bash that were caused mainly due to how function definitions passed via environment variables are parsed in bash. This vulnerability is a result of incorrect parsing of these function definitions in environment variables. Bash continued

CVE	Description	Seconion	OSSEC
2014-6271	Remote code execution	✗	✓(W)
2014-6277	Heredoc (Uninitialized use)	✗	✗
2014-6278	Command substitution RCE (Unusual parser state)	✗	✗
2014-7169	File Creation (Parser error)	✗	✗
2014-7186	Redir stack (Buffer overflow)	✗	✗
2014-7187	Off by one (Buffer overflow)	✗	✗

Table 2.1. Shell shock variants

to parse even after the function definition ended. Hence, anything after the function definition was parsed and executed as well. This feature itself has been employed in a variety of ways, including for web servers and embedded systems. The Shellshock vulnerability was discovered in 2014, and characterized primarily in the context of web servers. Shellshock was exploitable because adversaries may make network requests that both update environment variables and trigger the execution of Bash shells. Web servers used such functionality commonly to run scripts (CGI scripts) to process web requests. Embedded systems that run Bash also often propagate network request input to Bash shells. With the proliferation of Internet of Things devices, opportunities for exploitation of embedded devices is greater since not all have been patched. There are six variants (six vulnerabilities) of the Shellshock vulnerability. The variants are due to other parsing errors and memory errors in Bash. All of these additional variants, except for the command substitution (CVE-2014-6278), could only be remotely exploited without the first vanilla Shellshock vulnerability (CVE-2014-6271). We evaluated the efficacy of two existing IDS systems against all six variants as shown in the table 2.1. The last two columns represent whether these intrusion detection systems, Seconion and OSSEC respectively, would have detected these vulnerabilities ahead of time. The W in the bracket indicates that the intrusion detection system OSSEC would detect it, but an adversary could have used a worked around to get past the IDS.

Shellshock is a zero day vulnerability. The NIDS, Seconion, would not have detected a shellshock attack (zero day exploit) ahead of time as there were no appropriate rules to detect the malicious HTTP request payload used to exploit Shellshock. It would fail to detect the two variants which enabled the exploitation of the other variants and therefore fail to detect the other variants as well. All network intrusion detection systems, including Seconion, employed a Snort rule based on the shell shock (signature based) to protect against it after the Shellshock vulnerability was disclosed. Similarly, OSSEC would not be able to detect Shellshock as it focuses on certain kinds of misuses such as

rootkit detection and performs analysis of the system logs it collects. It supports user defined misuse/signature based rules for the log analysis. OSSEC employed a catch all rule which was able to detect vanilla Shellshock (CVE-2014-6271) payload as it triggered a segmentation fault. However, it would still require some manual effort and expertise as OSSEC captures a generic error (segmentation fault). The command substitution variant (CVE-2014-6278) would not be detected and therefore the other variants could be exploited. Additionally, in our experimentation we found that by modifying the payload slightly an adversary could get past OSSEC and exploit the vanilla Shellshock vulnerability. This illustrates our argument about existing intrusion detection systems focusing on using known vulnerabilities and their lack of visibility into programs.

Chapter 3
Related Work

3.1 Limitations of Prior Attack Graph Approaches

Aforementioned attack graph approaches relied on known vulnerabilities, suffered from limited visibility into all system components, specifically programs, and scalability was still a concern. They were mainly focused on the network layer. Capobianco et al. [57] proposed and defined a model for intrusion detection using attack graphs. The critical insight being that attack graphs naturally correspond to the intrusion detection steps of acquiring threatening permissions and leveraging them in operations to perform exploits. Additionally, using attack graphs for each layer addresses the limited visibility problem thereby helping protect systems against attacks that traverse the system's network, host and program layers. They identify the problems, opportunities and issues associated with computing these attack graphs. However, they do not identify the necessary analyses and how they can be used to construct such an attack graph (for any layer). We build on the attack graph based intrusion detection vision presented in this work by identifying and developing the necessary analyses needed to construct program attack graphs.

3.2 Limitations of Intrusion Detection

One acknowledged limitation of intrusion detection systems is that they focus on known attacks and attack behaviors, which limits their ability to detect new attacks and attacks similar to benign behaviors. Another limitation is that NIDS and HIDS each lack visibility into the programs where vulnerabilities are often exploited. They essentially treat programs as black boxes. As a result, NIDS and HIDS must detect attacks based on communications in the network and system calls, respectively, lacking critical information

about threats and their exploitation. Researchers have explored a variety of ways to improve IDS effectiveness, but they do not completely overcome the limitations above. First, to improve visibility, systems sometimes combine network and host IDS into a hybrid IDS, such as SolarWinds [12] and Sagan [13]. They integrate network and system monitoring to correlate log entries from both layers to improve attack detection confidence. Sagan performs log analysis similar to OSSEC [4] and Samhain [5]. It also supports Snort [2] like rules to monitor and detect intrusions at the network layer. SolarWinds' Security Event Manager (SEM) collects logs from network intrusion detection system (NIDS) that determines the amount and types of attacks on the network and integrates those details with logs from other infrastructure. While correlating network and host operations can improve attack detection, hybrid IDS' still focus on known attack behaviors and lack visibility into programs. This was clearly illustrated by the recent Solarwinds hack [58]. Second, to reduce dependence on known attacks, systems have been proposed to perform vulnerability-focused detection as well as exploit-focused detection [41], [59], [60]. Exploit-focused detection has several flaws. The most important one being its inability to detect attacks or different exploits with variations. For example, one popular way for attackers to evade exploit-focused systems [41] is by using one or more encoding techniques that cause the attack to appear differently on the wire but decode the same way at the client. Additionally, this method requires discovering, cataloging and writing signatures for new exploits. Vulnerability-focused detection systems [41–44] focus on detecting the vulnerabilities itself rather than the exploits. This provides better detection than exploit-focused detection. However, both these approaches still depend on known vulnerabilities and manual creation of signatures using the same, limiting their ability to detect new attacks. Hybrid intrusion detection systems [12,13] and vulnerability focused detection [41–44] are closest to our work in spirit.

3.3 Program Analysis for Security

Program analysis techniques, both dynamic and static, have been used to to better secure programs. Taint analysis is an information flow analysis where objects are tainted based on the source(s) specified and tracked using a data-flow analysis. Taint analysis has been used to check for information leakage [61]. Similarly, fuzzing is a well known, quick and cost-effective technique to find bugs/potential security flaws in applications. More advanced fuzzing techniques are either taint-based, symbolic, or concolic (a portmanteau of concrete and symbolic). However, it is limited to bugs that cause the program to

crash. Sanitizers such as ASAN [62], UBSAN [63] use program analysis techniques to identify potential bugs/vulnerabilities, and instrument programs to detect and protect against them. This is similar to an intrusion prevention system. These tools handle vulnerabilities such as buffer overflows, use-after-free and integer overflow bugs. They are overly conservative and monitor all operations that could be a potential bug. Additionally, adversaries may still get past the monitor employed and further propagate the attack.

Approaches using program analysis techniques for security have been used to find and/or prevent exploitation of bugs as we see above. These approaches focus on detecting specific vulnerability classes and/or protecting against them to make a program more secure. They do not identify attack paths that may lead to a vulnerability and the attack path's subsequent propagation from the vulnerability. This is crucial for intrusion detection as it is not always possible to patch/harden or detect these bugs with complete accuracy at the operation. Additionally, it may be beneficial to know how a bug may be exploited. Recent work using program analysis has proposed a more principled approach of leveraging security properties, given an exploit, to automatically repair the program [64] or to perform root cause analysis [65]. This is closely related to the notion of attack states being security property violations we define and use in our program attack graphs. However, these tools identify security property violations in a reactive manner after it has been exploited and detected.

Chapter 4
Program Attack Graphs

This chapter is as organized as follows. We first present the formal definition of attack graphs, explain what they represent with examples and discuss the attack graph structure in section 4.1. Subsequently, in section 4.2 we describe both the techniques we employ and the framework we developed to construct program attack graphs. Additionally, we discuss propagation in Section 4.3.

4.1 Attack Graph

We adopt the attack graph definition of the Ou et al. [54] paper. This definition was adopted and proposed to be used for a more principled systemic intrusion detection by Capobianco et al. [57]. We further build on their attack graph vision for intrusion detection. We envision employing modular attack graphs for each component to have a more systemic approach towards intrusion detection.

Definition 1 *An* attack graph*, $G = (V, E)$, is a graph where: (1) the set of vertices, $V = S \cup A$, where S is the set of* attack states *(called* facts *in Ou et al. [54]) and A is the set of* attack actions *(called* derivations *in Ou et al. [54]) and (2) the set of edges, $E = R \subseteq P$, where $(u, v) \in R$ is a* precondition edge *when $u \in S$ and $v \in A$ and $(u, v) \in P$ is a* postcondition edge *when $u \in A$ and $v \in S$.*

We refer to individual programs, hosts and networks as components. Attacks may traverse in the same layer, for example between two programs, or even across layers. An example would be an attack traversing through the network, host and a program such as shellshock [66]. Attack surfaces in a program could be system calls which receive data through the network, or from another process. Additionally, programs may open

adversarial files. Similarly, hosts may receive data from other untrusted hosts in the same network.

Definition 2 *An attack surface is a source interface in a component which may receive adversarial input.*

Attack states correspond to security property violations such as information flow or memory safety. These violations are similar to anomalies and they grant the adversary new privileges. These new found privileges create potential threats. For example, an information flow violation could mean that the adversary has introduced some low integrity data that a high integrity process may use. This violation may allow the adversary to trick the user into using this data. Uses could including executing parts of that low integrity data.

Definition 3 *An attack state is a security property violation which grants the adversary privileges.*

The threats created by the attack states, security property violations, could be exploited by the adversary through various operations to gain additional privileges. These operations are analogous to misuses. We refer to these exploit operations as attack actions. Examples of attack actions in a program could be a buffer overflow (memory error), remote code execution (exec system call) or even a send or write (information leak).

Definition 4 *An attack action is an operation the adversary may perform to exploit the privileges gained to further escalate his privileges.*

We have defined the attack graph and each kind of node formally. We now define the edges. The precondition edges $R \quad E$ between states and actions represent the privileges granted by the states which are then used in the exploit operation. This corresponds to a logical conjunction (AND) as the exploit operation (misuse) requires all the necessary privileges, that is, security property violations (states). For example, for an adversary to trigger a buffer overflow the adversary requires an unsafe operation (an unsafe pointer) and some data he could control. The unsafe pointer corresponds to a memory safety violation and the unsafe data corresponds to an information flow violation. The operation, that is, the buffer overflow (exploit/misuse) requires both these states. The post condition edges $P \quad E$ between actions and states represent the new

privileges (attack states) the adversary could gain with the exploit operation. This is a logical dis-junction (OR) as the exploit/misuse may violate any security property to gain a new privilege. Let's consider the same buffer overflow example. The exploit operation would be the memory operation which actually overflows the buffer. The adversary may overflow the buffer to overwrite the return address or a code pointer (control flow integrity violation), or overwrite some critical data as done in a data oriented attack (information flow violation). This naturally aligns with a logical dis-junction as the buffer overflow (exploit operation) may be used in different ways to gain different privileges (new attack states).

Attack Graph Structure

We propose a layered structure for attack graphs. This means the attack graph would be built over the underlying component graph. Therefore, our program attack graphs are built over the program component graph, a program dependence graph. Program dependence graphs capture the control flow and data flows in the program. We leverage a LLVM [16] pass to compute the program dependence graph from an earlier work [67]. The rationale behind a layered structure is that it separates the attack graph from the underlying component, capturing only necessary information in the attack graph. It also provides an easy way for us to access any additional information we need. This is because each attack graph node is connected to the underlying component graph node. We hypothesize that as an attack may evolve we may need to hoist necessary information upwards from the underlying component graph to the attack graph. We do not include these edges in the attack graph definition as these edges are not considered when performing analysis using the attack graph rather they are used only when needed (in case of attack graph events).

4.2 Program Attack Graphs

The attack graph framework supports the construction and use of attack graphs (all layers). It handles the evolution of attacks through dynamic events across all layers. In this work we have identified and employed techniques for program attack graph and developed the framework. We now describe the attack graph framework and its implementation details. The attack graph framework consists of an event plane and an analysis plane. The event plane is implemented in python and it employs various analyses (LLVM [16] passes) written in C++, which constitute the analysis plane. The analysis we employ for programs operates on source code. We think this is a reasonable

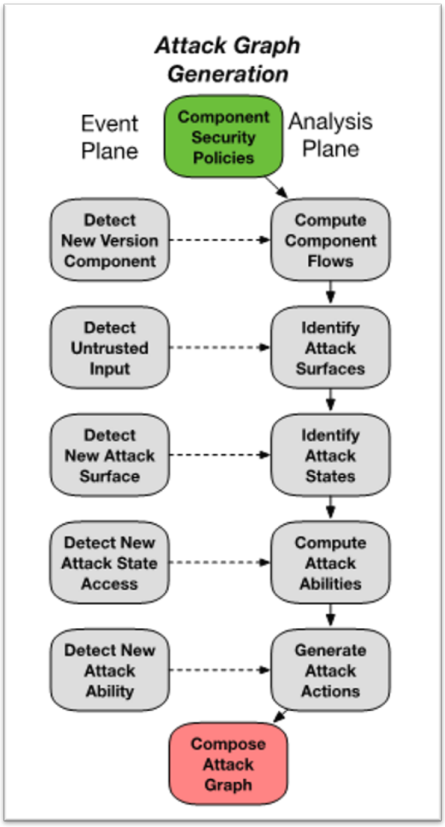

Figure 4.1. Attack Graph Generation

assumption. The framework uses Neo4j [17], a single model graph database, to store the computed attack graph and the component graph. We employ indexes (caches) in the database to improve performance. Our analyses interact with the database. The general workflow of the construction of the attack graph is as illustrated in the Figure 4.1. We discuss the events in a subsequent Chapter 5. The framework consists of five modules[1]. The engine module, is the primary module which spawns of a new process for each of the other modules. It initializes the other modules, synchronizes them and performs error handling . The engine module is the only exposed part of the framework which can interact with external entities such as run time monitors. Each of the modules employs caching as they are capable of maintaining state whereas the underlying analysis cannot. We will now describe each module, that is, the techniques/analyses each module uses to compute the relevant parts of the attack graphs.

[1]We refer to them as modules as they are implemented as python modules.

4.2.1 Attack Surfaces

The attack surfaces are computed in the attack surface module. As defined earlier the attack surfaces are any source interfaces that may receive adversarial input. In a program this could be system calls that may receive adversarial input such as IPC(inter process communication) related system calls, or file related system calls, or network related system calls. We leverage the component graph, program dependence graph, to identify such system calls. We maintain a list of sensitive system calls, which may be an attack surface, and create a regular expression using the same. We leverage Neo4j's powerful querying to identify all relevant system calls and create attack surface nodes using the same. This is a coarse approach to computing the program attack surfaces. However, even with this approach we may miss program attack surfaces due to library calls. These library calls may internally invoke sensitive system calls we account for. Unfortunately, as we do not have the code for these library calls we miss out on them. We could statically link all libraries before analyzing the program but this may affect scalability. Additionally, we may also miss different variants of well known system calls such as the fopen64 system call. We envision using run time monitors to account for missing attack surfaces.

This module performs three main tasks. One, it computes the various component graphs. This is a one time computation and may only be triggered again if the component graph changes. Second, it computes the attack surfaces. For the program attack surfaces we use the technique described above. Third, it handles surface related and component related events such as a component graph change or expansion of an attack surface.

4.2.2 Attack States

The attack states correspond to security property violations. For the program attack states we consider two security properties: information flow and memory safety. We compute two kinds of attack states corresponding to an information flow violation. We differentiate between the two based on the misuse/exploit operation that it enables. They do no differ fundamentally as they both correspond to an information flow violation. To compute the information flow violations we leverage the program attack surfaces computed static taint analysis, and LLVM [16] def use chains. The attack surfaces by definition serve as the starting points for the attack graph. Taint analysis is an information flow analysis where objects are tainted based on the source(s) specified and tracked using a data-flow analysis. We use taint analysis treating the program attack surfaces as the sources. We compute the values that it may taint which may be used

Algorithm 1 nesCheck's type inference algorithm

```
1: for each declaration of pointer variable p do do
2:    classify(p, SAFE)
3: end for
4: for each instruction I using pointer p do do
5:    r ←− result_of(I)
6:    if I performs pointer arithmetic then
7:       classify(p, SEQ)
8:       classify(r, SAFE)
9:    end if
10:   if I casts p to incompatible type then
11:      classify(p, DYN)
12:      classify(r, DYN)
13:   end if
14: end for
```

either in a system call or in a memory operation. We use the def use chains to identify whether the tainted value may be used in either an important system call or a memory operation. We refer to former as system call argument based attack states. We refer to the latter as tainted operand states. Both of these states violate information flow as they are indicative of untrusted adversarial data flowing through the surfaces to values (data objects). For the system call argument based states we use a configurable list of critical system calls. For the tainted operand states, we currently consider only store/write operations to compute potential buffer overflows. Additionally, we found that we missed certain memory operations because we failed to account for library calls. For example, sprintf writes to a buffer and is therefore a memory operation. To account for this we modify our analysis to treat the appropriate library calls as memory operations and we hard code checks based on the signature of these library calls. Currently, we use this approach for sprintf and snprintf.

For memory safety we leverage the "nesCheck" tool from an earlier work [68]. This tool identifies unsafe pointers. It classifies pointers as safe, sequential or dynamic based on its uses. A pointer that may be manipulated by pointer arithmetic is classified as sequential. A pointer involved in unsafe cast operations such as casting between different levels of indirection or between different root types is classified as dynamic. We create unsafe pointer states using the sequential and dynamic pointers identified by this tool. We limit the analysis to only pointers declared on the stack and ignore global pointers. Additionally, we modify this tool to use points to analysis (SVF) to record whether the pointer only points to the stack or if it may point to the heap as well. We capture this information in the attack graph node. These unsafe pointer states correspond to

Algorithm 2 Attack States Computation

```
1: {TOS- Tainted Operand States}
2: {SCA- System Call Argument states}
3: initialize set TOS = null
4: initialize set SCA = null
5: {Step 1 - Taint the surfaces(S)}
6: for each surface s in S do
7:    for each value v in taint_output(s) do
8:       {Step 2 - Check where v is used (def-use)}
9:       if v used in memory operation then
10:         TOS = TOS   v
11:         connect surface s to state v
12:      else if v used in a sensitive system call then
13:         SCA = SCA   v
14:         connect surface s to state v
15:      end if
16:   end for
17: end for
```

memory safety violations. The algorithm for classifying pointers [68], which we leverage to identify unsafe pointers(sequential and dynamic), can be found in Algorithm 1. The algorithm for the attack state computation specifically for tainted operand states and system call argument based states can be seen in Algorithm 2. The attack state module performs two main tasks. One, it computes the various attack states (from the surfaces). To compute the program attack states from the surfaces we use the techniques described above. Two, it propagates computed attack actions. We discuss propagation and the techniques used in Section 4.3.

4.2.3 Attack Actions

The attack actions are exploit operations (misuses) the adversary may perform using the privileges gained , due to earlier security property violations, to further escalate his privileges (reach new attack states). We consider two security properties as we stated earlier : information flow and memory safety. Let's consider an information flow violation, this would enable an adversary to use low integrity data in an operation. A natural choice for the adversary would be to use critical system calls as one cannot do useful things without system calls. This is why we compute system call based attack actions as they represent meaningful exploit operations. To compute these, system call based actions, we leverage the system call argument based states computed earlier and LLVM's [16] def-use chains to identify which system calls are reachable. We create system call attack

Algorithm 3 Attack Action Computation - Unsafe pointers and System calls

```
 1: initialize set upa_actions = null
 2: initialize set sys_actions = null
 3: {Compute system call actions}
 4: for each state s in SCA do
 5:    for each use u of s do
 6:       if u is a sensitive system call then
 7:          sys_actions = sys_actions   u
 8:          connect state s to action u
 9:       end if
10:    end for
11: end for
12: {Compute unsafe pointer actions}
13: for each state s in TOS do
14:    for each use u of s do
15:       if u is a memory operation then
16:          if pointer ptr in u is an unsafe pointer then
17:             upa_actions = upa_actions   u
18:             connect state s to action u
19:          end if
20:       end if
21:    end for
22: end for
```

actions using these. The arguments to a system call correspond to capabilities. When we computed the system call states we identified all system call arguments (information flow violations) the adversary can control/corrupt. We initially connect all these system call states to the corresponding system call actions. This is inaccurate as the adversary need not have to control all these arguments to perform a meaningful exploit operation (to gain new privileges). Also, they may have to control specific argument or arguments to escalate their privileges. This depends on the semantics of the system call and its signature. We create and use a declarative specification which maps the minimum arguments (capabilities) needed for the attacker to escalate privileges. This specification provides an enumeration of the meaningful capabilities the adversary would need to escalate privileges. We use this declarative specification to enumerate all such states and action combinations to each critical system call. We refer to the attack actions which correspond to the same system call in the program but differ in terms of capabilities as logical attack actions.

Let's consider a memory safety violation, any operation that could exploit it would have to be a memory operation. Based on the unsafe pointer states we compute, the

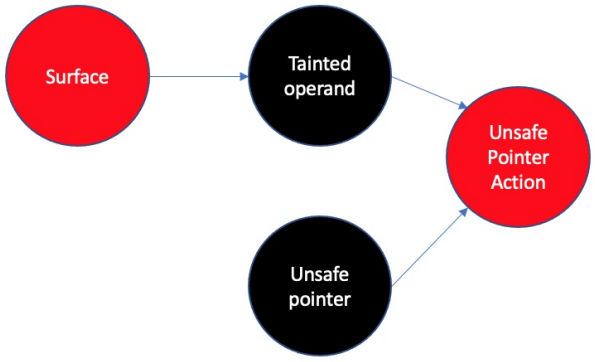

Figure 4.2. Unsafe pointer attack action : Tainted operand (information flow violation) and unsafe pointer (memory safety violation)

sequential pointers could be used in a bounds misuse and the dynamic pointers in a type misuse (which may lead to a subsequent bounds misuse). As stated earlier we limit ourselves to bounds misuse, specifically buffer overflows. We want to compute exploit operations where we can leverage both a memory safety violation and information flow violation. The rationale here is that we want to compute buffer overflows where the adversary can control the data being written (could overwrite the address of the pointer or the data being pointed to). We think that implicit buffer overflows, where the adversary does not control the data (no information flow violation), is not that interesting as it could mostly just be used in a DoS (denial of service) attack. Additionally, this may introduce a large number of false positives as it would be hard to argue about each operation that uses an unsafe pointer (memory safety violation). It is for these reasons we compute unsafe pointer actions, which use privileges gained through both a memory safety violation (specifically bounds) and an information flow violation (tainted operand states). As can be seen in the Figure 4.2 this is a logical conjunction. We leverage LLVM's def use analysis to ensure that the pointer used in the operation (destination operand) is an unsafe pointer (unsafe pointer state) and the data is tainted (information flow state). As stated earlier for library calls such as sprint and snprintf we use their function signatures to do the same check. When we compute unsafe pointer states we record whether it strictly points to the stack or it may point to the heap. We record this information when computing unsafe pointer actions as well. The algorithm we employ for computing attack actions can be seen in Algorithm 3 (We omit low level details here). The attack action module performs two main tasks. One, it computes the various attack actions. To compute the program attack actions, system call and unsafe pointer actions,

we use the techniques described above. Two, it handles action related events such as output surface expansion. We define output surface in Chapter 5.

Attack Graph Module

This module's task is to provide various graph analysis techniques. We provide a simple querying technique currently. To query the graph to check for specific attack actions or specific paths. Our program attack graphs just like prior attack graph approaches can also be used for vulnerability analysis and risk analysis. We envision using this module to provide other analysis and to even convert our graph into other formats such as a Mulval [55] based representation to be used by other tools.

4.3 Attack Action Propagation

We've seen how one can compute a program's attack surfaces, attack states and actions in the previous Section 4.2.3. These attack actions, exploit operations, may propagate further, to gain or escalate privileges (corresponding to safety property violations) using the privileges the adversary already gained due to security property violations (states).

Definition 5 *Attack action propagation refers to the process by which an adversary may gain additional privileges through an exploit operation (action), using his existing privileges.*

We've explained why we compute system call based actions in the earlier Subsection 4.2.3. We propagate the system call based attack actions by only considering information flow violations it could lead to. We do not fully account for the semantics of the system call. We use taint analysis to see which data objects/values (information flow violations) this system call based action could affect. We include data objects/values which may be used either in a sensitive system call or a memory operation. This is similar to the algorithm we use to compute states from surface as seen in Algorithm 2. We use a manual declarative specification to decide whether to taint the return value or an argument of the system call. This partially captures the semantics of the system call. We propagate unsafe pointer actions considering only information flow violations as well. Information flow violations resulting from unsafe pointer actions correspond to data oriented attacks. We define and model two capabilities of these unsafe pointer actions based on the adversary's capabilities. The same stack frame capability allows the adversary to use the unsafe pointer action (buffer overflow) to corrupt any data object in the same stack frame. The global capability allows the adversary to use the unsafe

pointer action (buffer overflow) to corrupt any data object in the same stack frame and subsequent [2] frames. We are only interested in data objects that are used either by a subsequent[2] system call or a subsequent memory operation. As explained earlier in Subsection 4.2.3 we focus on only system call based actions and unsafe pointer actions (buffer overflows). However, for simplicity's sake we have limited the propagation to only data objects used in subsequent system calls.

We propagate unsafe pointer actions using an LLVM pass, "Pointer Effects". The algorithm employed in the analysis for the global capability is illustrated in Algorithm 4. It consists of two main steps. First, it finds all the critical system calls reachable from each function as per the control flow. It does this by analyzing each function until the analysis reaches a fixed point. Two, it computes all data objects/values (states), which are subsequently used in a system call, an unsafe pointer action may affect. We do this by using the data collected in the first step. We check what system calls are reachable from the function (function containing the unsafe pointer action) and are after the unsafe pointer action according to the control flow. We then check if any of arguments to that system call can be corrupted/affected by the unsafe pointer action (data dependence check). To check if the argument can be corrupted we check if it is declared on the stack. This is accurate for unsafe pointer actions corresponding to pointers that point only to the stack as they can overwrite at least the address even if the argument data is on the heap. However, this is an over-approximation for unsafe pointer actions corresponding to pointers that may point to the stack or the heap. This is because the argument data may also be on the stack. This can be extended to check whether the argument is on the heap, or in the data section (global data). We use a similar algorithm for the "same stack frame" capability except we do not have to record all reachable system calls. Also, we do not have to check subsequent calls in the function (with the unsafe pointer action) which are not system calls. Additionally, we can extend this algorithm to account for propagation of unsafe pointer actions to data objects/values used in memory operations as well. We can do this by recording memory operations reachable from each function.

This analysis has a few caveats. One, it is limited to data oriented attacks (information flow violations) . We do not consider code pointers (control flow integrity violations). This would correspond to control flow hijacking attacks such as return oriented programming. Two, we may introduce a significant number of false positives using this algorithm in case of unsafe pointer actions with pointers which may point to heap . Three, we currently limit ourselves to data objects used in subsequent system calls only.

[2]By subsequent we mean a subsequent instruction in the program according to the control flow

Algorithm 4 Pointer Effects - Global capability

```
1: initialize set S = null
2: converged = false
3: {Step 1 - Compute all reachable system calls for each function}
4: for all f ∈ F do
5:    sys_calls[f] = Sensitive sys calls in f
6: end for
7: while not converged do
8:    for all f ∈ F do
9:       for each call instruction c in f do
10:         sys_calls[f] = sys_calls[f] ∪ sys_calls[callee]
11:         if sys_calls[f] changed then
12:            converged = false
13:         else
14:            converged = true
15:         end if
16:      end for
17:   end for
18: end while
19: {Step 2 - Compute sys call args that may be a□ected}
20: for all upa ∈ UnsafePtrActions do
21:    f = function containing upa
22:    {Subsequent to the upa with respect to control flow - Captures control flow dependence}
23:    for each subsequent call c ∈ f do
24:       if c is a sensitive sys call then
25:          {Data flow dependence check}
26:          if ∃ args to c can be corrupted then
27:             S = S ∪ args
28:             connect upa to each argument in args
29:          end if
30:       else
31:          for all sys_call ∈ sys_calls[callee] do
32:             {Data flow dependence check}
33:             if ∃ args to sys_call can be corrupted then
34:                S = S ∪ args
35:                connect upa to each argument in args
36:             end if
37:          end for
38:       end if
39:    end for
40: end for
```

Chapter 5
Evolution of Attack Graphs

We have built our framework based on the attack graph vision in Capobianco et al. [57]. We've applied the same principled approach, identified and developed various analysis to compute program attack graphs. We envision using modular component attack graphs to have better visibility into the system and to handle the evolution of attacks. Our attack graph vision can be seen in figure 5.1. We can leverage modular attack graphs for components in different layers to deal with multi layer and multi component attacks. These attack graphs at different layers connect naturally through the inter layer attack surfaces. Multi layer or sophisticated attacks can traverse through multiple layers in multiple components. It is therefore crucial for an intrusion detection system using these attack graphs to capture such data flows which traverse multiple layers. We can compute attack graphs for a whole system, considering the network, host and program, following the general principles described in our program attack graph construction and using the appropriate analyses. However, our attack graphs may miss out on certain attacks (or attack paths) in their computation. We can supplement our attack graphs using run time monitors. The run time monitors, placed based on our initial attack graph, may record something important which might require the attack graph to be updated. For example, a run time monitor might find that a program is sending adversarial input to another program and the interface in the receiver program was not identified as an attack surface. Additionally, our attack graphs may be outdated as components change. Changes in any component could enable new attacks both inter and intra component. These could be changes in a program, network configuration changes or even changes to a system's access control policy. It is therefore crucial we have a mechanism in place to evolve our attack graphs as needed. This would ensure we update our attack graphs when such critical changes occur. We provide an event based mechanism to evolve attack graphs as needed.

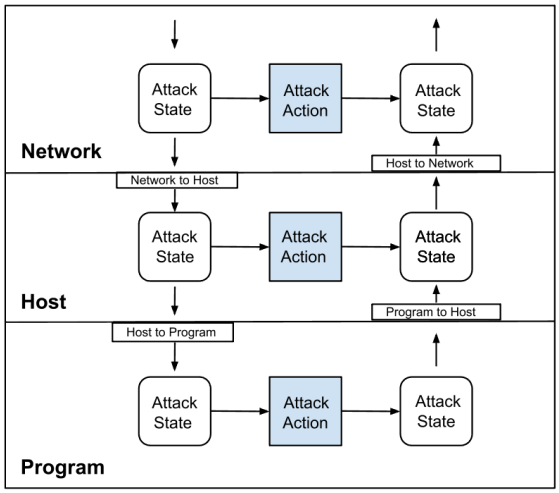

Figure 5.1. Attack Graph Vision

5.1 Attack Graph Events

As stated above we may have to update our attack graphs based on run time monitoring. These changes may be to capture information the attack graph missed or capture changes in the component itself. We've implemented our framework taking this into consideration. We identify different kinds of events and associate the same with the necessary analysis in our framework. This can be seen in Figure 5.2. Our initial attack graph computation follows the workflow in the figure. We start by computing the component graph and then identify attack surfaces. We proceed to compute attack states and possible attack actions. We further propagate the attack actions computed which may lead to new states and so on. We continue this until we either reach a fixed point (no new states or actions) or some hard limit. We can see the association between the various events and analysis that it triggers. For example, we see that if a new attack surface is detected we would need to identify the attack states, compute new actions possible and son. Our framework associates each event with the related module and triggers the appropriate analysis. We will now explain in detail how we implement events and event handling in our framework. Currently we have defined three events in our framework : input attack surface expansion, output surface expansion and component change.

Attack surface expansion is an event which indicates we have observed a new attack surface. Similarly, an output surface expansion event indicates that an attack action may expand to affect another component. We refer to these as output surfaces. An output surface is a reachable attack action which may affect another component The component change event indicates that a change in the underlying component has occurred. For

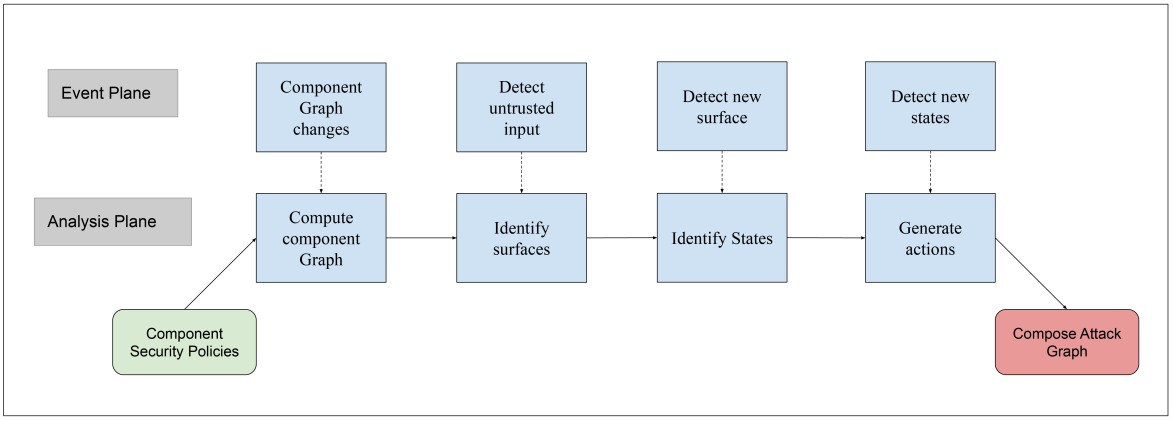

Figure 5.2. Attack Graph Events

example, a program's source code may have changed. We envision using incremental analysis to deal for certain events such as component graph change.

Events Implementation

We implement two kinds of events: internal and external. We define internal events as intra framework events (inter modular, here the modules are part of the framework) as they are internal with respect to the framework. External events are related to an external entity (external with respect to the framework). External events may lead to or trigger internal events. Internal events are implemented using UNIX domain sockets. This can be replaced through any other mechanism by updating the appropriate event related methods in the modules. The main (engine) module starts a server socket. Each individual module such as the attack surface module "registers" its event handler by connecting to the said socket. The main module acts as the event dispatcher. It triggers events by sending data to the appropriate module using the socket. Additionally, other modules can trigger an event using the main module. We define and use an event info abstraction which captures all the necessary information. This includes event type, component type or layer, and component identifier. An event may additionally want to pass data such as which surface to expand in case of the surface expansion event. Python provides the flexibility of modifying the event info data structure easily. We leverage python's pickle package along with the UNIX domain socket for marshalling and unmarshalling the event data.

The main module starts a new thread on which it opens a UNIX domain socket. It accepts incoming connections and once all the modules have connected ("registered") it forks a new thread (internal event dispatcher). The new thread dispatches internal events to the appropriate modules when an internal or external event is triggered. The main

module keeps track of which "handle" (socket) belongs to which module. The internal event dispatcher uses the select system call to check if any of the modules have triggered an internal event. Each module such as the attack surface module has an event handler running on the main thread. It first registers its event handler, and the registered event handler then runs with the appropriate handle. It waits on the handle for any event using select. Based on the event type it forks of a new thread invoking the appropriate method.

We have to deal with external entities such as run time monitors and therefore external events. We implement an external event handler in the main module using a TCP socket. The main module runs the external event handler on the main thread. It accepts incoming connections and receives externals events using the same event info-based data format. The external event handler's mechanism could be changed to better match the external entities that would trigger such events. An external event would have to be translated into an internal event for the appropriate module to trigger the appropriate analysis. For example, the main module on receiving the attack surface expansion event would have to translate this event to the attack surface expansion event for the attack surface module.

5.2 Scenario

We describe a scenario to illustrate how an attack graph may evolve. Let's consider "log-user-session", a program used to log a user's ssh session. The sshd program has to be configured to launch "log-user-session". There was a vulnerability found in "log-user-session" - CVE 2018-1000857 in 2018. This vulnerability would enable remote code execution. The vulnerability stems from the improper sanitization of an environment variable (SSH_CLIENT) which is later to create the log file. This allows the adversary to create and write to a file. Unfortunately, to make things worse log-user-session runs as root. We illustrate a scenario in the figure 5.3. The scenario is the adversary can write any desired content into the file as it logs the sshd session. The adversary can achieve remote code execution by writing to a cron tab.

Let us assume we have computed attack graphs for sshd and log-user-session. Additionally, we have missed the "getenv" attack surface through which the remote code execution is possible. A run time monitor could detect the new "getenv" attack surface and trigger an attack surface expansion. This would update log-user-session's attack graph to include the attack path corresponding to the vulnerability (CVE 2018-1000857).

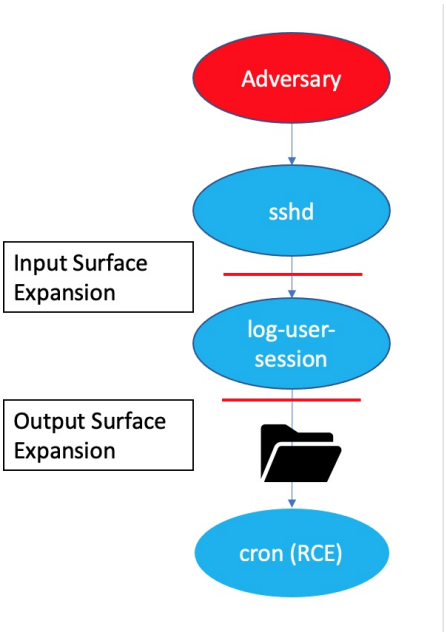

Figure 5.3. Scenario: log-user-session

An external entity may analyze this attack path and trigger an output surface expansion event as write is an output surface. Therefore, with respect to log-user-session we can see both its input attack surface expand and the output surface expand (to crond).

Chapter 6
Results

6.1 Attack Graph Generation

We computed program attack graphs for five programs using our attack graph framework. Interestingly, both uftpd and log-user-session programs had two zero-day vulnerabilities, CVE-2020-20276 and CVE-2018-1000857 respectively, which our program attack graphs captured. The uftpd program had a memory error whereas log-user-session allowed an adversary to create and write to any file. This could be exploited to achieve remote code execution as described in the scenario. The information related to these program attack graphs can be found in Table 6.1. We refer to log-user-session as log in the Table due to a lack of space. These are the final statistics after the attack graph computation reached a fixed point, that is no new states or actions. However, we find that very few of these, if any, are exploitable, that is through unsafe pointer actions. This would be because very few uses of an unsafe pointer (potential exploit operations) use tainted data (adversarial data). Only three programs : uftpd, bzip2 and crond have unsafe pointer actions. We find system call based attack actions are more common. We will now explain in detail how our program attack graphs captured the log-user-session vulnerability (CVE-2018-1000857) and the uftpd vulnerability (CVE-2020-20276).

Log-user-session vulnerability

This vulnerability corresponds to an information flow violation which allows the adversary to create any file through an unsanitized environment variable. The log-user-session program runs as root as it is a suid program which is configured to run with sshd. This program is used to log the user's ssh session. Log-user-session is referred to as log in the Table 6.1.

The first step in the attack graph computation is the component graph generation, for a program this would be the program dependence graph. We then proceed to compute

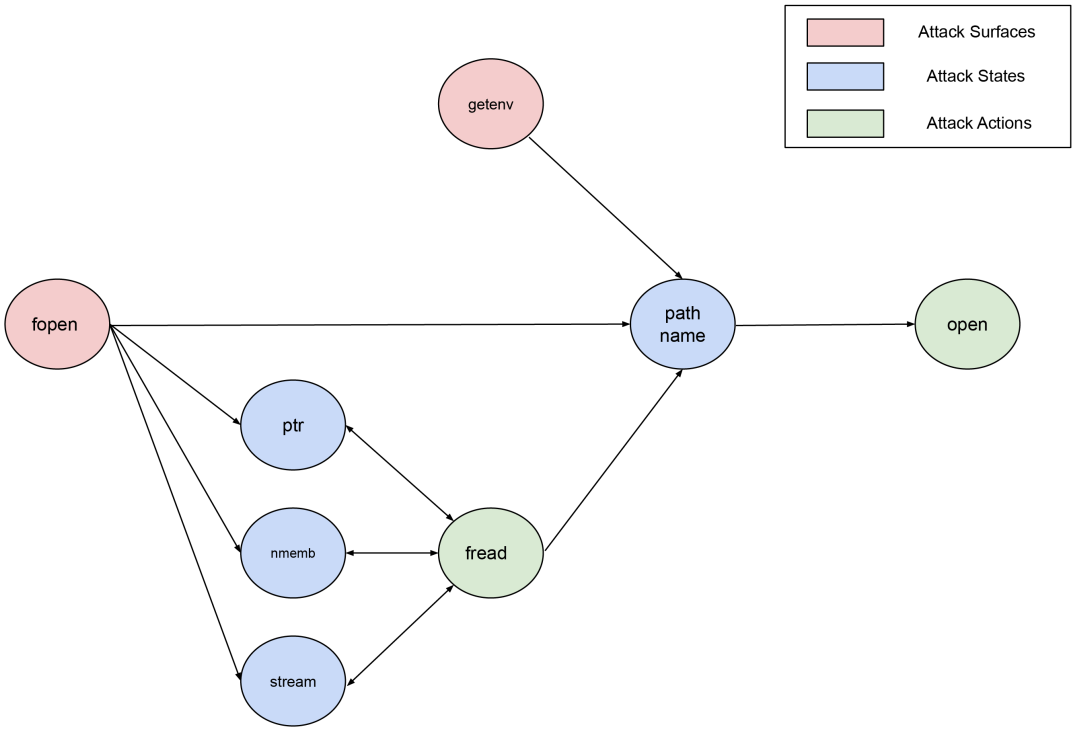

Figure 6.1. log-user-session vulnerability: computed sub-graph (CVE-2018-1000857)

the attack surfaces. For log-user-session our computation, which uses a regular expression constructed from a hard coded list of sensitive system calls, found four attack surfaces: open, fopen, command line arguments (argc,argv) and getenv. The next step in our attack graph computation is to compute the attack states (information flow violations and memory safety violations). Our attack states computation for information flow violations, found 39 system call argument states but no tainted operand states. For this computation we use taint analysis and def-use analysis. Both these states represent information flow violations we just differentiate between the two based on their uses. We also found 125 unsafe pointer states (memory safety violations). We then compute the possible attack actions from these attack states. In this step our framework computes the possible system call actions as well as memory operation actions. We do not find any unsafe pointer actions. However, we do find 25 system call attack actions. The next step is the propagation of the computed attack actions. We find these computed actions do not propagate to any tainted operand states. We find they do propagate to the existing 39 system call states (no new system call argument states). Upon complete computation of the attack graph (fixed point) we find there are 39 system call argument states, 125 unsafe pointer states, 25 system call actions and no tainted operand states (therefore no

unsafe pointer actions) from the 3 attack surfaces.

We now explain how our computed program attack graph captures the vulnerability. We identified getenv as an attack surface. When we computed the attack states from the surfaces we found one of the system call argument states computed from the getenv attack surface to be the path name argument, used in a subsequent open system call. Therefore, the open system call is identified as an attack action in the subsequent attack action computation. Thus the computed attack path from getenv (surface) to open (action) corresponds to the vulnerability in the log-user-session program. Additionally, when we computed the propagation of the attack actions we found the file descriptor argument (state) from the open attack action , which is used in a subsequent write. This implies there might be a an attack path computed where the adversary can create as well as write to the file. This makes sense as the normal functionality of the program (log-user-session) is supposed to log the user session by design. The visualization of the computed sub-graph (part of the attack graph) which captures the vulnerability can be seen in Figure 6.1. (We can see loops in the sub-graphs between fread and the states corresponding to its arguments). The sub-graph shows us a path from fopen to open through fread and getenv to open. The latter is the path which corresponds to the vulnerability.

Uftpd vulnerability

This vulnerability corresponds to a memory error involving an information flow violation which allows the adversary to overflow a buffer on the stack. The uftpd program starts a file server on the system and receives various commands through a socket. This vulnerability stems from the unsafe use of the sprintf function without any bounds check and an information flow violation. An adversary when configuring the address through the PORT command could overflow the local buffer used to store the address.

The first step in the attack graph computation is the component graph generation, the program dependence graph. We then proceed to compute the attack surfaces. For uftpd our computation, found 9 attack surfaces : socket, command line arguments (argc, argv), and fopen. There are 5 instances of the fopen system call and 3 instances of the socket system call. The next step in our attack graph computation is to compute the attack states (information flow violations and memory safety violations). Our attack states computation for information flow violations from the surfaces, found 43 system call argument states but no tainted operand states. We also found 466 unsafe pointer states (memory safety violations). We then compute the possible attack actions (system call actions as well as memory operation actions.) from these attack states. We find

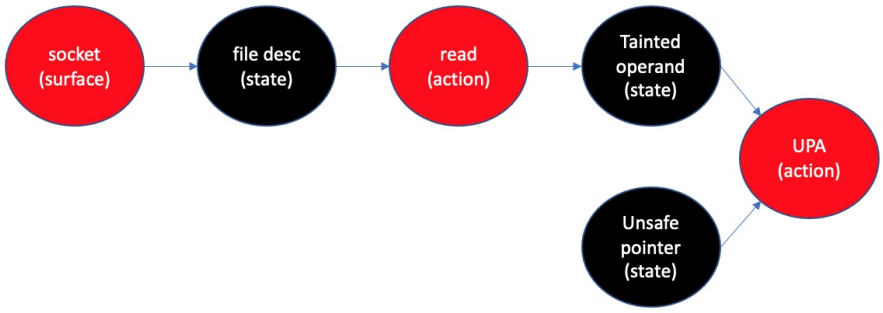

Figure 6.2. Uftpd: attack path computed (CVE-2020-20276)

no unsafe pointer actions, but we do find 28 system call actions. We then propagate the actions computed in the previous step. We find they propagate to 12 new tainted operand states and 27 existing system call argument states (no new system call argument states). This subsequently leads us to find 9 unsafe pointer actions. Upon complete computation of the attack graph (fixed point) we find 43 system call states, 12 tainted operand states, 28 system call actions and 9 unsafe pointer actions from the 9 attack surfaces.

We now explain how the computed attack graph captures the sprintf vulnerability (CVE-2020-20276). Our computation identified the socket call as an attack surface. In the attack states computed from the surfaces, we found that the socket system call (attack surface) lead to a system call argument state (information flow violation) corresponding to a file descriptor data object, which is subsequently used in a read operation. The read system call is then identified as an attack action. Therefore, this computed path corresponds to the vulnerability. We can see the attack path (corresponding to the vulnerability) in our computed program attack graph in Figure 6.2.

Program	Surfaces	SC states	TO states	UP states	SC actions	Stack UPA	Non stack UPA
cat	2	14	0	190	9	0	0
crond	10	60	16	163	64	8	8
bzip2	14	32	1	538	26	0	1
log	4	39	0	125	25	0	0
uftpd	9	43	12	466	28	9	0

Table 6.1. Computed Program Attack Graphs information. SC:- System call, TO:- Tainted operand, UP:-Unsafe pointer, UPA:-Unsafe pointer actions

6.2 Event-based attack graph generation

We estimate the possible number of ways our attack graph may evolve with respect to two kinds of changes. The two changes/events are input surface expansion and output surface expansion. As stated earlier in Section 5.1 we define input surface expansion as any occurrence where we are notified by a run time monitor about a new attack surface. Similarly, output surface expansion is any occurrence where we find that an attack action (reachable from a surface) may affect another component and therefore act as an output surface. We estimate the maximum possible number of input surface expansion events by counting the number of library calls (including system calls) in the program and subtracting the number of computed attack surfaces from it. We estimate the possible output surface expansion events by manually analyzing the system call attack actions for each program and identifying which ones could be output attack surfaces. We defined output attack surfaces as reachable attack actions which may affect another component For example, write or send system calls could affect the local host, another program (IPC) or the network. The possible attack graph evolution results are shown in Table 6.2.

We simulate the input surface expansion event for two programs and record how it affects the attack graph as seen in Table 6.3. We envision using run time monitors which would trigger these events to help us evolve our attack graphs. We manually analyzed the programs to identify potential new surfaces. The number of new potential paths was approximated based on the number of new edges added from the new surface to different attack states. We recorded whether the new attack surface (input surface) led to any new states and actions. In addition we identified the new output surfaces, which are new output system calls in the updated attack graph, where monitoring would be necessary (to detect output surface expansion) .

We can see that for the crond program the new input attack surface (pipe) did not lead to any new states or actions. This means that there were no states or actions found which were reachable only from the new input attack surface. These states and actions

Program Name	Input Surface expansion Events	Output surface expansion events
cat	67	2
crond	101	26
bzip2	39	18
log-user-session	63	7
uftpd	117	9

Table 6.2. Estimation of Program Attack Graphs Evolution

Program Name	Input Attack Surface	New Potential Paths	New SC States (TO States)	New SC Actions(UPA)	New Output Surfaces
crond	pipe	60	0	0	0
bzip2	getenv	47	15(4)	12(4)	10

Table 6.3. Simulated Input Surface Expansion Results

were already reachable and part of the computed program attack graph. Therefore, the input surface expansion event just added new potential paths from the new input surface to these states and actions. On the other hand, we can see that for bzip2 the new input attack surface (getenv) led to new states and actions. This means the new surface led to states and actions which were unreachable from the existing surfaces. The new actions included 10 output system call actions which we refer to as output surfaces.

Chapter 7
Conclusion

Intrusion detection systems are a commonly deployed defense to detect attacks. The prolific increase in the number of vulnerabilities make them critical. However, these systems fail to detect unknown attacks or zero-day vulnerabilities and sophisticated multi layer attacks. This is because current intrusion detection systems lack visibility into all system components, specifically programs. They focus on known attacks/vulnerabilities, limiting their ability to detect unknown attacks. As a result they fail to track the evolution of attacks across multiple layers (network, host and program). We propose using modular component attack graphs for proactive intrusion detection. We develop an attack graph framework and identify techniques to construct program attack graphs. These program attack graphs provide better visibility into programs and break the dependency on known vulnerabilities/ exploits by computing potential attack paths in a program in a principled manner. We identify analyses needed to compute and connect attack surfaces, a source interface in a compute which may receive adversarial input, to attack states, flaws or security property violations which grant the adversary privileges that create threats, and actions actions, operations that exploit those threats. We model two security properties in our program attack graphs: memory safety and information flow. We also identify and develop techniques to propagate these attack actions, as exploit operations may lead to subsequent safety property violations. Our framework supports the possible evolution of attacks graphs through dynamic events. We evaluate the efficacy of existing intrusion detection systems through a case study on the shellshock vulnerabilities. We construct program attack graphs for multiple programs. Additionally, we were able to capture known vulnerabilities in two of the programs (uftod, log-user-session) in our computed program attack graphs. We estimate the number of potential input surface expansion and output surface expansion events with respect to the evolution of our attack graph. We simulate the input surface expansion event for two programs and record how it affects

the attack graph. We also illustrate how an attack graph may evolve with a detailed scenario.

Our program attack graphs by design are proactive and well suited for intrusion detection. They compute the potential attack paths by considering potential security property violations and exploit actions possible starting from the attack surfaces. Prior work using program analysis techniques for security usually focus on either finding a vulnerability [69–72] or detecting and patching/preventing [62,63] them. The goal of these approaches is to help secure programs and usually geared towards specific vulnerability classes. They do not compute whether the vulnerability is exploitable (reachable), and even if they do they assume it can be patched or monitored with complete accuracy. They do not not consider and compute the subsequent privileges an adversary could gain by exploiting this vulnerability. This information is crucial as an intrusion detection system can decide how and where to monitor based on the potential attack paths captured. More recent work has leveraged using security properties , similar to our computation of attack states, along with program analysis techniques, to automatically repair programs [64] and for root cause analysis [65]. These approaches do consider security properties and are more principled. However, their goal is to help after a vulnerability or an attack is detected. We leverage these program analysis techniques and use them in a manner better suited for intrusion detection through our program attack graphs.

Existing hybrid intrusion detection systems such as Solarwinds [12], and Sagan [13], do consider both the host and the network layer. They integrate network and system monitoring to correlate log entries from both layers to improve attack detection confidence. However, they still focus on known attack behaviors and lack visibility into programs. Network intrusion detection systems, both misuse [1, 2] and anomaly based [20–22], have the same caveat where they lack visibility into programs. Misuse based host intrusion detection systems [4, 5] analyze system logs using rules based on known attacks and perform rookit detection (based on known rootkits). Additionally, anomaly based host intrusion detection systems [9, 20, 21, 24] usually suffer from false positives. Wagner et al. [11] had shown how several of the anomaly based host intrusion detection systems based on Forrest et al. [8]'s seminal paper (sliding window mechanism) were susceptible to mimicry attacks. Data mining based anomaly intrusion detection systems are resource expensive. Machine learning based anomaly intrusion detection might introduce vulnerabilities due to their inherent learning mechanisms [10]. Machine learning techniques tend to be computationally expensive [19] and it is difficult to identify the right features.

Most attack graph approaches [47, 48, 50] used attack graphs for reliability analysis

[46], shortest path analysis [47] or risk analysis [45]. Subsequent work [52] then explored methods to compute attack scenarios from attack graphs. These attack graphs were not scalable enough to be used for large networks with hundreds of hosts. Scalability [53, 54] was addressed later, but these attack graphs were primarily for networks and lacked visibility into programs. Additionally, they focused on known vulnerabilities on hosts through the use of scanners such as NESSUS [6]. Our program attack graphs provide visibility into programs and are not dependent on known attacks. We also track the evolution of our attack graphs through events in our framework. We acknowledge the fact that program attack graphs' scalability is an open issue. We envision constructing and using the program attack graphs on demand for intrusion detection. We leave this for future work.

Bibliography

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

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

[3] (2010), "Surricata," .
URL https://suricata-ids.org/

[4] Cid, D. B. (2008), "OSSEC," .
URL http://www.ossec.net/

[5] Wichmann, R. (2006), "SAMHAIN," .
URL https://la-samhna.de/samhain/

[6] Anderson, H. (2003) "Introduction to nessus," *Retrieved from Symantec*.

[7] Farmer, D. and E. H. Spafford (1990) "The COPS security checker system," .

[8] Forrest, S., A. S. Perelson, L. Allen, and R. Cherukuri (1994) "Self-nonself discrimination in a computer," in *Proceedings of 1994 IEEE computer society symposium on research in security and privacy*, Ieee, pp. 202–212.

[9] Smaha, S. E. et al. (1988) "Haystack: An intrusion detection system," in *Fourth Aerospace Computer Security Applications Conference*, vol. 44, Orlando, FL, USA.

[10] Huang, L., A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar (2011) "Adversarial machine learning," in *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pp. 43–58.

[11] Wagner, D. and P. Soto (2002) "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 255–264.

[12] (2019), "SolarWinds," .
URL https://www.solarwinds.com/

[13] Quadrantsec (2015), "Sagan," .
URL https://quadrantsec.com/

[14] Wang, L., A. Liu, and S. Jajodia (2006) "Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts," *Computer communications*, **29**(15), pp. 2917–2933.

[15] Noel, S. and S. Jajodia (2008) "Optimal ids sensor placement and alert prioritization using attack graphs," *Journal of Network and Systems Management*, **16**(3), pp. 259–275.

[16] Lattner, C. and V. Adve (2004) "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, IEEE, pp. 75–86.

[17] "Neo4j," https://neo4j.com/.

[18] Bace, R. and P. Mell (2001) "Intrusion detection systems, National Institute of Standards and Technology (NIST)," *Technical Report 800-31*.

[19] Patcha, A. and J.-M. Park (2007) "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer networks*, **51**(12), pp. 3448–3470.

[20] Lunt, T., A. Tamaru, F. Gilham, R. Jagannathm, C. Jalali, P. Neumann, H. Javitz, A. Valdes, T. Garvey, and A. R.-t. I. D. Expert (1992) "System (IDES)," *Computer Science Laboratory, SRI International, Menlo Park, CA, USA, Final Technical Report*.

[21] Anderson, D., T. Frivold, and A. Valdes (1995) "Next-generation intrusion detection expert system (NIDES): A summary," .

[22] Biles, S. (2003) "Detecting the unknown with snort and the statistical packet anomaly detection engine (spade)," *Computer Security Online Ltd., Tech. Rep*.

[23] Maxion, R. A. and F. E. Feather (1990) "A case study of ethernet anomalies in a distributed computing environment," *IEEE transactions on Reliability*, **39**(4), pp. 433–443.

[24] Forrest, S., S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff (1996) "A sense of self for unix processes," in *Proceedings 1996 IEEE Symposium on Security and Privacy*, IEEE, pp. 120–128.

[25] Somayaji, A. and S. Forrest (2000) "Automated Response Using System-Call Delay." in *Usenix Security Symposium*, pp. 185–197.

[26] Warrender, C., S. Forrest, and B. Pearlmutter (1999) "Detecting intrusions using system calls: Alternative data models," in *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*, IEEE, pp. 133–145.

[27] Valdes, A. and K. Skinner (2000) "Adaptive, model-based monitoring for cyber attack detection," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, pp. 80–93.

[28] Ye, N., Y. Zhang, and C. M. Borror (2004) "Robustness of the Markov-chain model for cyber-attack detection," *IEEE Transactions on Reliability*, **53**(1), pp. 116–123.

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

[30] Shyu, M.-L., S.-C. Chen, K. Sarinnapakorn, and L. Chang (2003) *A novel anomaly detection scheme based on principal component classifier*, *Tech. rep.*, MIAMI UNIV CORAL GABLES FL DEPT OF ELECTRICAL AND COMPUTER ENGINEERING.

[31] Grossman, R. (1997) "Data mining: challenges and opportunities for data mining during the next decade," *Disponível: Magnify site. URL: http://www. magnify. com, Consultado em dez*.

[32] Lee, W., S. J. Stolfo, P. K. Chan, E. Eskin, W. Fan, M. Miller, S. Hershkop, and J. Zhang (2001) "Real time data mining-based intrusion detection," in *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, vol. 1, IEEE, pp. 89–100.

[33] Tan, K. M. and R. A. Maxion (2003) "Determining the operational limits of an anomaly-based intrusion detector," *IEEE Journal on selected areas in communications*, **21**(1), pp. 96–110.

[34] Sung, A. H. and S. Mukkamala (2003) "Identifying important features for intrusion detection using support vector machines and neural networks," in *2003 Symposium on Applications and the Internet, 2003. Proceedings.*, IEEE, pp. 209–216.

[35] Sequeira, K. and M. Zaki (2002) "Admit: anomaly-based data mining for intrusions," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 386–395.

[36] Lane, T. and C. E. Brodley (1999) "Temporal sequence learning and data reduction for anomaly detection," *ACM Transactions on Information and System Security (TISSEC)*, **2**(3), pp. 295–331.

[37] Hubballi, N. and V. Suryanarayanan (2014) "False alarm minimization techniques in signature-based intrusion detection systems: A survey," *Computer Communications*, **49**, pp. 1–17.

[38] Sommer, R. and V. Paxson (2003) "Enhancing byte-level network intrusion detection signatures with context," in *Proceedings of the 10th ACM conference on Computer and communications security*, pp. 262–271.

[39] Massicotte, F., M. Couture, L. Briand, and Y. Labiche (2007) "Model-driven, network-context sensitive intrusion detection," in *International Conference on Model Driven Engineering Languages and Systems*, Springer, pp. 61–75.

[40] Krishnamurthy, S. and A. Sen (2001) "Stateful intrusion detection system (sids)," *ICIW*, **1**, pp. 1–10.

[41] Cisco (2015), "Vulnerability-Focused Threat Detection: Protect Against the Unknown," Cisco White Paper.
URL `https://www.cisco.com/c/en/us/products/collateral/security/ips-4200-series-sensors/white_paper_c11-470178.html`

[42] Wang, H. J., C. Guo, D. R. Simon, and A. Zugenmaier (2004) "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits," in *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 193–204.

[43] Li, Z., G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv (2010) "Netshield: massive semantics-based vulnerability signature matching for high-speed networks," *ACM SIGCOMM Computer Communication Review*, **40**(4), pp. 279–290.

[44] Brumley, D., J. Newsome, D. Song, H. Wang, and S. Jha (2006) "Towards automatic generation of vulnerability-based signatures," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, IEEE, pp. 15–pp.

[45] Schneier, B. (1999) "Modeling security threats," *Dr. Dobb's journal*, **24**(12).

[46] Somesh, J. and J. M. Wing (2001) "Survivability analysis of networked systems," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, IEEE, pp. 307–317.

[47] Swiler, L. P., C. Phillips, D. Ellis, and S. Chakerian (2001) "Computer-attack graph generation tool," in *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, vol. 2, IEEE, pp. 307–321.

[48] Dacier, M. and Y. Deswarte (1994) "Privilege graph: an extension to the typed access matrix model," in *European Symposium on Research in Computer Security*, Springer, pp. 319–334.

[49] Harrison, M. A., W. L. Ruzzo, and J. D. Ullman (1976) "Protection in operating systems," *Communications of the ACM*, **19**(8), pp. 461–471.

[50] Dacier, M., Y. Deswarte, and M. Kaâniche (1996) "Models and tools for quantitative assessment of operational security," in *IFIP International Conference on ICT Systems Security and Privacy Protection*, Springer, pp. 177–186.

[51] Phillips, C. and L. P. Swiler (1998) "A graph-based system for network-vulnerability analysis," in *Proceedings of the 1998 workshop on New security paradigms*, pp. 71–79.

[52] Sheyner, O., J. Haines, S. Jha, R. Lippmann, and J. M. Wing (2002) "Automated generation and analysis of attack graphs," in *Proceedings 2002 IEEE Symposium on Security and Privacy*, IEEE, pp. 273–284.

[53] Ammann, P., D. Wijesekera, and S. Kaushik (2002) "Scalable, graph-based network vulnerability analysis," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 217–224.

[54] Ou, X., W. F. Boyer, and M. A. McQueen (2006) "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 336–345.

[55] Ou, X., S. Govindavajhala, and A. W. Appel (2005) "MulVAL: A Logic-based Network Security Analyzer." in *USENIX security symposium*, vol. 8, Baltimore, MD, pp. 113–128.

[56] Schneier, B. (2000) "Secrets & Lies: Digital Security in a Networked World, John Wiley & Sons," *Inc. New York, NY, USA*.

[57] Capobianco, F., R. George, K. Huang, T. Jaeger, S. Krishnamurthy, Z. Qian, M. Payer, and P. Yu (2019) "Employing attack graphs for intrusion detection," in *Proceedings of the New Security Paradigms Workshop*, pp. 16–30.

[58] (2020), "Solarwinds hack," .
URL https://en.wikipedia.org/wiki/2020_United_States_federal_government_data_breach#Microsoft_exploits

[59] Networks, P. A. (2019), "What Is An Intrusion Prevention System?" .
URL https://www.paloaltonetworks.com/cyberpedia/what-is-an-intrusion-prevention-system-ips

[60] Farroukh, A., M. Sadoghi, and H.-A. Jacobsen (2011) "Towards vulnerability-based intrusion detection with event processing," in *Proceedings of the 5th ACM international conference on Distributed event-based system*, pp. 171–182.

[61] Yang, Z. and M. Yang (2012) "Leakminer: Detect information leakage on android with static taint analysis," in *2012 Third World Congress on Software Engineering*, IEEE, pp. 101–104.

[62] Serebryany, K., D. Bruening, A. Potapenko, and D. Vyukov (2012) "AddressSanitizer: A fast address sanity checker," in *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pp. 309–318.

[63] "Clang undefined behavior sanitizer," http://clang.llvm.org/docs/UsersManual.html, accessed: 2010-09-30.

[64] Huang, Z., D. Lie, G. Tan, and T. Jaeger (2019) "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 539–554.

[65] Yagemann, C., M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee (2021) "*{*ARCUS*}*: Symbolic Root Cause Analysis of Exploits in Production Systems," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[66] Chazelas, S., M. Zalewski, and T. Ormandy (2014), "ShellShock vulnerability," CVE-2014-6271, CVE-2014-6277, CVE-2014-6278, and CVE-2014-7169.

[67] Liu, S., D. Zeng, Y. Huang, F. Capobianco, S. McCamant, T. Jaeger, and G. Tan (2019) "Program-mandering: Quantitative privilege separation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1023–1040.

[68] Midi, D., M. Payer, and E. Bertino (2017) "Memory safety for embedded devices with nesCheck," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 127–139.

[69] Wang, T., T. Wei, G. Gu, and W. Zou (2010) "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, IEEE, pp. 497–512.

[70] Haller, I., A. Slowinska, M. Neugschwandtner, and H. Bos (2013) "Dowser: a guided fuzzer to find buffer overflow vulnerabilities," in *Proceedings of the 22nd USENIX Security Symposium*, pp. 49–64.

[71] Pham, V.-T., M. Böhme, and A. Roychoudhury (2020) "AFLNet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, pp. 460–465.

[72] Stephens, N., J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna (2016) "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." in *NDSS*, vol. 16, pp. 1–16.