Metrics-based investigation of distributed intrusion detection and attack surface reduction

CS-TR-5014

Jeff Stuckman James Purtilo
Computer Science Department
University of Maryland

August 2012

Abstract

Two distinct but related projects – titled *Improved product assurance through automatic trace generation and analysis* and *Improved cyber security via decentralized intrusion detection and dynamic reconfiguration* respectively – have been under way in this laboratory, both with support from Office of Naval Research, which the authors gratefully acknowledge. The purpose of this report is to frame the even broader goal we envision, which is ultimately to understand how to not just measure properties of a running system which characterize its susceptibility to vulnerabilities in the eyes of potential intruders, but also to dynamically adjust the running system so as to either reduce or remove those vulnerabilities. What is of greatest concern in a running system is not the vulnerabilities we already know about – after all, they would likely have been repaired at the point of discovery – but rather the vulnerability that only an intruder understands. Our hypothesis is that static analysis together with measurements at run time may telegraph suggestions for dynamic reconfiguration which might repel an intruder, without loss of service by the system, long enough for operators to identify and understand whatever might have been the specific defect that had been probed. The present report updates our statement of the long term research goals and presents our status on the two projects under way.

1 Introduction

Software systems have long suffered from the fact that once an adversary penetrates one component or function of an application, the adversary can then gain control over the entire application (and sometimes the hardware or operating system that the application is running on as well). This makes it much harder to secure a system against malicious actors, as the number of failure points that would allow any given function of the system to be compromised will increase as the complexity of the system increases. If system administrators were able to erect internal firewalls, restricting the actions of a malicious user who gains control over one component while leaving legitimate users of the system unaffected, the risk associated with more complex systems could be mitigated.

The problem of mitigating the security risks associated with complex systems raises several measurement questions:

1. On a source code level, what is the aspect of large or complex systems (versus less complex systems) that increases the likelihood or power of exploits?
2. How can this aspect be measured and quantified?
3. Which methods of mitigating a system's complexity result in an improvement in this metric?

4. Does an improvement in this metric result in a real-world reduction in the number of vulnerabilities or the severity of such vulnerabilities?

These measurement questions motivate our integrated approach toward mitigating the complexity of a system and measuring the effects of our mitigation. First, we develop compile-time or deployment-time frameworks that monitor the inter-component activity of a software system for signs of security breaches. Next, we develop mitigations that reconfigure or secure a software system when a breach in one component has been detected, in order to reduce the severity of the exploit. As a parallel effort, we also develop and evaluate metrics for the attack surface of a software system, examining how attack surface is related to complexity. We then evaluate the metrics by observing their success in predicting past vulnerabilities and exploits. Finally, we evaluate our mitigatory approaches against previous security efforts by examining the impact of the mitigations on attack surface metrics, both theoretically and in real-world test cases.

Other DoD-sponsored efforts, such as SELinux [4] with Mandatory Access Control, have studied how inter-process privilege in software systems may be limited to mitigate the effects of an exploit. We take an alternative approach to limiting the privilege of a program, focusing on inter-program flows (rather than intra-program flows). Another ONR-funded effort [5] examines how to distribute software systems across small, redundant components. Aside from providing redundancy, such an approach could compliment ours by providing our program analysis tools with additional opportunities to insert instrumentation.

In the next sections, we present an outline of this project, along with the technical approach and current state of each subtask. Progress indicators are given using the following statuses:

- Planned: The activity has been planned but not yet executed
- Design complete: Detailed technical design work has been performed for the activity, but the activity has not yet been realized
- Placeholder implemented: Some design work has been performed for the activity and the activity is a prerequisite toward a later step, but the activity is not yet complete. Therefore, a placeholder of limited functionality has been implemented so later activities can proceed.
- In progress: Realization of the activity has started
- Prototype completed: The activity has been completed and has yielded a prototype, suitable for research but not production use

2 Infrastructure to evaluate attack surface metrics

The concept of attack surface has long been informally used to discuss the number of opportunities that a potential attacker has to compromise a system. More recently, quantitative attack surface metrics have been developed which analyze a system and quantify its attack surface with a numerical score. [3] [1] Attack surface metrics are of particular interest to security engineers and software deployers, due to the notion that they reflect the probability that a piece of software contains yet-undiscovered security flaws. If such metrics were vetted and universally applicable, they could be used to help select the most secure software from several alternatives, or predict if a feature being added to a piece of software will be an undue security risk.

Several researchers have developed attack surface metrics, using informal arguments and ad-hoc experiments to argue for their validity. However, there has not yet been a serious effort to develop an evaluation method that would allow these metrics to be formally vetted. Our goal is to develop such an evaluation method and then evaluate the known attack surface metrics against each other (and against new metrics that we develop.) The result will be a vetted attack surface metric that could be used both to measure the security of a piece of software and to argue for the relative power of exploit countermeasures (such as firewalls) based on their theoretical impacts on attack surface.

Below, we summarize our current and planned work on attack surface measurement. Technical details of this work can be found in our related publication [8].

2.1 A corpus of vulnerabilities with associated trace data

Under our method for evaluating attack surface metrics, it is necessary to first acquire a *corpus* of past, known security vulnerabilities, whereas a corpus consists of a set of software that contains known security vulnerabilities such that the vulnerabilities and their exploits are represented in a machine-readable manner. Such a corpus provides a set of real-world scenarios where a security exploit was possible, allowing us to determine if any given attack surface metric would have predicted where or how the attacker entered the system.

2.1.1 A repository and workbench for conducting repeatable security experiments

Status: Prototype completed

In order to perform a variety of experiments involving a corpus of security vulnerabilities, it is necessary to build a "live" database of vulnerabilities that allows each vulnerability to be tested in an operational setting. We have implemented such a database that uses virtual machine technology in order to instantiate a "workbench" for any vulnerability in the corpus, allowing the vulnerability to be studied and exploited by the experimenter, who can observe the mechanics or the effects of the exploit.

The vulnerability repository (which stores the corpus) is implemented as a wiki hosted by the Semantic Mediawiki platform. We chose Semantic MediaWiki for the repository because it allows a structured database to be designed and built incrementally, eliminating unnecessary bottlenecks that would impede the collection of the corpus. The wiki contains one page per vulnerability in the corpus. Each page contains structured fields containing vulnerability identifiers (pointing to existing databases such as the CVE or OSVDB), the real-world application version which contained the vulnerability, the degree to which we can reproduce the exploit, and pointers to the virtual machine snapshots containing the vulnerable version of the application (which have already been prepared for the exploit). The free text in the wiki page contains instructions for exploiting the vulnerability, any payload that must be pasted into the web application to execute the exploit, and any scripts that the malicious user would run to execute the exploit.

The vulnerability workbench (which facilitates experiments involving vulnerabilities in the corpus) is implemented as a set of virtual machines and snapshots running under Virtualbox, supported by a set of scripts to help manage these virtual machines. For every version of every application represented in the corpus, we created a Linux virtual machine which contains a working, running copy of the application. Each virtual machine is based on Debian Linux version 4 or version 6, depending on the age of the application, allowing very old applications (which might be incompatible with recent operating systems) to be run. For every vulnerability in the corpus, we manipulated the virtual machine to put it in a state where the vulnerability could be exploited without performing any additional preparation work. Such preparation work includes actions such as creating user accounts, creating database entries such as blog posts, enabling features of the application, and adding user privileges. Such work is often necessary because some systems must contain assets for the malicious user to target. Once each virtual machine entered the exploitable state, we took a snapshot of the virtual machine, which allows this state to immediately be revisited by experimenters at any point in the future. The scripts associated with the vulnerability workbench automate the process of starting and stopping multiple virtual machines on the same host, allowing for simultaneous and automated experiments to be performed. The scripts also create and clean up virtual machines that allow for the aforementioned snapshots to be executed. Virtual machines are stored as differentials to base Linux images, allowing 42 virtual machines (and all of their associated snapshots) to be stored in only 16 GB of disk space.

2.1.2 A corpus of vulnerabilities for security experiments

Status: Prototype completed

We then used this repository and workbench to build a corpus of security vulnerabilities to facilitate our research. This corpus of vulnerabilities consists of 50 vulnerabilities in PHP web applications. Each vulnerability has been added to the corpus as described above, storing virtual machines with the vulnerable version of the application in a state where the vulnerability can quickly be exploited.

We chose web applications for the corpus because we expect that the transactional nature of web applications running in scripted languages will ease control and data flow analysis of the exploits, reducing the need to track persistent state stored in memory. We chose PHP web applications in particular because we found that the vast majority of reported and publicized exploits in web applications were on PHP web applications. Types of vulnerabilities represented in the corpus include:

- Arbitrary file upload
- Cross-site request forgery
- Information disclosure
- Parameter tampering
- Privilege escalation
- SQL injection
- Cross-site scripting

The concept of a security vulnerability corpus is described in more detail in our related work [7].

2.1.3 A database of exploit traces for the vulnerability corpus

Status: In progress

In order to perform the required analysis when evaluating attack surface metrics, we also need to collect a complete set of exploit traces for each vulnerability.

A *trace* consists of a sequence of statement references (file, line number, and possibly statement number), which may or may not include the first or last statement in a transaction (web request; hence, the trace need not be complete). A *set* of traces for this purpose encodes a set of (possibly incomplete) traces out of the domain of all possible traces which encompass no more than one transaction (web request). A set of traces need not be enumerated explicitly; such a set may be defined in relation to membership in a language (for example, a regular expression producing a set of partial traces).

A vulnerability's set of traces must be defined such that, for any transaction, if no portion of the transaction's trace is a member of the vulnerability's set of traces, then the transaction is not an exploit transaction. An exploit transaction is the transaction directly initiated by a malicious user that exploits the vulnerability in question. Essentially, a vulnerability's set of traces specifies events that *must* occur while the vulnerability is being exploited. For example, a vulnerability in a blog's commenting code could not be exploited if the malicious user never submits a comment, so portions of execution traces related to comment submission should be included in the vulnerability's trace set. The set of traces may overapproximate the malicious activity (allowing transactions to be tagged as potentially malicious even though they are not), but it may not underapproximate the malicious activity.

To this end, we have developed a tool that integrates with PHP's debugger interface in order to produce a complete execution trace for any transaction executed on the web application. We then developed scripts and Debian packages that allow the tool to be automatically installed on any VM in the vulnerability workbench, allowing for the tool to be incorporated in automated security experiments. We next plan to develop a search tool that assists the researchers in generating a trace set by viewing examples of exploit and non-exploit traces and then suggesting trace fragments that may be related to the exploit activity.

2.1.4 A database of feature metadata for the vulnerability corpus

Status: Planned

In order to evaluate attack surface metrics in relation to a corpus of vulnerabilities, we also need to formalize how features may be enabled or disabled on each application containing a vulnerability in the corpus.

A *feature* of a web application can be disabled with the effect of preventing a portion of the application's code from being executed. As described above, one purpose for attack surface metrics is to gauge the risk of adding new features to a program. An attack surface metric that fits this purpose would tend to give worse scores to features in real-world applications that were later found to contain vulnerabilities. The concept of *features* is integral to our attack surface metric evaluation technique.

We plan to provide a syntax to annotate applications in order to mark global configuration variables, SQL database values which act as configuration settings, and environmental conditions (such as the contents of configuration files) which are intended to toggle features on and off. We then plan to perform static analysis on the program to determine which segments of the program become unreachable when features are turned off. We can then compute how much of the program's attack surface each feature contributes, along with how many vulnerabilities each feature contributes (considering that some features may be overlapping and some vulnerabilities are not associated with features; see our related work [8] for the complete algorithm.)

2.1.5 A fully automated exploit simulation testbed

Status: In progress

Our current vulnerability corpus and vulnerability workbench automatically set up a virtual machine so it is ready to be exploited, allowing the experimenter to read and follow the exploit instructions from the vulnerability's wiki page. To complete the automated testbed, we are in the process of writing scripts to automate the exploit process using the Selenium browser automation framework. These scripts formalize the exploit instructions found in the wiki, allowing the entire exploit process (setting up the virtual machine, starting the virtual machine, executing the exploit, and shutting down the virtual machine) to occur without human intervention. This would permit the development of automated experiments that can run over the entire corpus of vulnerabilities, such as experiments that measure the effectiveness of an intrusion prevention system. Some experiments would also require annotations marking which web request should not have been permitted, in the case of multi-request exploits.

Currently, we have developed Selenium scripts for a portion of vulnerabilities in the corpus. We will then develop Selenium scripts for the rest of the corpus, as well as scripts on the virtual machine host that coordinate the management of the virtual machines with the execution of the scripts, with hooks for custom experiments to be inserted.

2.2 A framework to encode and compute attack surface metrics

Because there has not yet been a large volume of research studying attack surface metrics, a model allowing many such metrics can be expressed in the same formulation has not yet been created. One goal of this research is to develop universal formulations for expressing various attack surface metrics, along with tools which can compute an attack surface metric on any relevant program, as long as the attack surface metric can be expressed formally. Such a tool is an integral step toward evaluating attack surface metrics (either in isolation or comparatively).

2.2.1 A universal format to represent attack surface metrics

Status: Design complete

In order to express and encode attack surface metrics for the purposes of computing and evaluating them, we have designed a representation for attack surface metrics. In our representation, attack surface metrics are defined as a sum of *elements* which individually and independently contribute to the attack surface metric. Elements can be defined by any computable function (which is defined by the individual attack surface metric being represented), and elements can be localized in the program's source code, being tied to file(s), statement(s), or other units. When expressed in this manner, attack surface metrics can be computed (regardless of the representation of the element generating function, as long as it is computable) and evaluated against a corpus of vulnerabilities. All previously developed attack surface metrics that are known to us can be expressed with this representation.

We have formally described this attack surface metric representation in our related work [8] and plan to develop a source code interface so attack surface metrics can also be expressed in code.

2.2.2 A tool to compute attack surface metrics on existing applications

Status: Planned

Once attack surface metrics are represented in a standard format and implemented in code, the last step is to perform measurements on existing software systems. We plan to develop a tool that can automatically analyze a software system and compute its attack surface, based on an attack surface metric which is loaded into the tool. It is important to note that the attack surface metric is computed without a need for expert judgement or manual review of the source code. This measurement tool is used as part of our method to evaluate attack surface metrics. The tool could also be used by developers to gauge the security impact of their own design decisions and system administrators who are concerned about the security impact of new software or configuration decisions.

2.3 A method and tool to evaluate attack surface metrics

Once attack surface metrics can automatically be computed, the last step is to evaluate them. We evaluate attack surface metrics by measuring, under some corpus of vulnerabilities, the degree that program features which resulted in more vulnerabilities had larger attack surface metrics.

Our method is summarized here and described in our related work [8].

2.3.1 A method to evaluate attack surface metrics

Status: Design complete

We have designed an algorithm for evaluating an attack surface against a corpus of known vulnerabilities. The algorithm yields a numerical score for the attack metric, allowing attack surface metrics to be compared as long as they were evaluated against the same corpus.

For every program containing vulnerabilities in the corpus, for every feature, a data point is generated plotting the attack surface contribution of the feature against the number of vulnerabilities introduced by the feature. The attack surface contribution of a feature is computed by performing a control-flow analysis on the feature, determining which attack surface elements are associated with units of code that become unreachable when the feature is disabled. The vulnerabilities are tied to features in a similar way by using the trace set for each vulnerability. A vulnerability is considered to have been introduced by a feature if all traces in a vulnerability's trace set become infeasible (at least one element or branch is unreachable in each trace) when the feature is disabled. A vulnerability may be introduced by multiple features, in the case where an interaction between features introduces the vulnerability. Security roles are also considered in the analysis, discounting system states which can only be entered by users possessing valid administrator credentials, which an attacker exploiting a vulnerability probably lacks.

The correlation between these two variables of the data points is then computed, measuring the tendency for vulnerabilities to fall inside features with larger attack surfaces. Note that this evaluation can be performed without having to assume that all vulnerabilities for a given program are included in the analysis. Making such an assumption would require our corpus to be *complete* (contain all vulnerabilities for each application), which would make the procedure much more difficult and threaten its validity, because not all vulnerabilities are known and some vulnerabilities may have been patched without any record of such in the corpus. This drives our decision to correlate the presence of vulnerabilities with features of high attack surface, rather than correlating the absolute number of vulnerabilities in a program with the program's attack surface as a whole.

2.3.2 A tool to perform control-flow analysis on existing applications

Status: Design complete

In order to implement this method to evaluate attack surface, we need a tool which can perform static analysis and control-flow analysis on a program. Some form of program analysis (which will depend on the element generating function for each metric) will be required to identify and localize attack surface elements in a program, allowing its attack surface to be computed. Control-flow analysis will be required to determine which code artifacts (such as statements, expressions, functions, or branches) become unreachable when program features are disabled.

We plan to implement our program analysis, attack surface computation, and attack surface evaluation for programs in PHP, because our corpus of security vulnerabilities is also in PHP. The control-flow analysis problem to be solved is determining the reachability of code artifacts based on constraints that are imposed and then specializing the program under such constraints (where constraints are logical predicates that declare features to be disabled, by negating the configuration condition that would have caused the features to be enabled, given the previous definition of features and configuration). We plan to investigate existing PHP static analysis tools, such as Pixy [2], to determine if they could be used to perform such an analysis. If no such tools exist, we intend to implement a control flow analysis algorithm that we have developed, which is described below:

Analyze the source code: Parse the program's source code and resolve features such as inclusions and references to non-local variables

Build control flow graph: Build a graph for the program to be analyzed, where nodes represent statements and edges represent possible successor statements (including possible targets for function calls and function call returns)

Annotate control flow graph: For each constraint to be applied (as defined earlier in this section), annotate the node of the control flow graph where the constraint is to be applied with the predicate for the constraint, in relation to the variable(s) constrained

Propagate annotations: Propagate constraints across the control flow graph, using a standard data flow analysis algorithm for monotonic transfer functions (in our case, the transfer function propagating constraints of the program's variables)

Program specialization: Locate branch expressions on the control flow graph where constraints prevent the branch expression from ever evaluating to "true". Fix such graph expressions and locate nodes which have become unreachable. The program has now been specialized under a set of constraints.

2.3.3 A tool to evaluate an attack surface metric against a corpus

Status: Planned

Once tools are available to compute attack surfaces metric and analyze on the program in the corpus, we plan to develop a tool that automatically evaluates the attack surface metrics. Such a tool will use the programs found in the vulnerability corpus, but does not actually need to execute the programs, instead relying on control-flow analysis to gauge the relationship between vulnerabilities and attack surface.

To facilitate this, we plan to export the vulnerability database's data (currently stored in the semantic wiki) into a format that our attack surface evaluation tool can read, supplying data on vulnerability traces and program feature annotations.

2.4 An evaluation of new and existing attack surface metrics

Status: Planned

Once we produce a tool to evaluate attack surface metrics, we plan to perform a comprehensive evaluation of existing metrics and new metrics that we develop. To our knowledge, this kind of formal evaluation has

never been performed on an attack surface metric. Along with allowing attack surface metrics to be compared, our evaluation will provide evidence that supports (or discredits) the concept of quantitative attack surface metrics in general, by demonstrating that such metrics do (or do not) point to vulnerable sections of code. This evidence may direct us and other researchers toward the most promising security metrics research areas in the future.

Aside from evaluating attack surface metrics, we also plan to develop our new attack surface metrics as part of this subtask. When developing such metrics, we plan to pay special attention to the concept that attack surface metrics can measure the power of an adversary to influence the execution of the code (loosely, quantifying the attacker's "privilege") by driving the program execution toward statements or branches that the attacker may desire to exercise. Such a concept is not explored in most of the existing attack surface metrics, and quantifying this privilege of an attacker may reveal how *internal* architectural decisions can reduce the amount of privilege (quantified as attack surface) that an attacker gains when "*external* features" (such as open ports) are added to a system. (In other words, the effective attack surface of a system may depend on the internal, "hidden", architecture of a system, aside from the visible surface features of said system.)

3 Framework for distributed intrusion detection

Numerous technologies have been developed that monitor a system or network and attempt to determine when a malicious user is attempting to compromise (or has compromised) the security of the system. Such technologies include network intrusion detection systems, anomaly detectors, host-based intrusion detection systems, compiler enhancements that monitor the integrity of the program's data structure, and server frameworks that attempt to detect suspicious web requests.

Out of the existing intrusion detection technologies that explicitly monitor user activity for signs of intrusion, the vast majority operate by observing and analyzing the *externally observable* behavior of the system in question. For example, network-based intrusion detection systems monitor the network packets that enter and leave the system, while traditional host-based intrusion detection systems monitor how processes interact with their environment, such as through system calls. We hypothesize that improved intrusion detection performance can be realized by monitoring activity which is *internal* to a system. In addition, implementing a *distributed* intrusion detection method, where each part of a system detects intrusions affecting the other parts, would allow a system to internally react and prevent an attacker from gaining control over the entire system when one part of the system has been compromised.

Our ultimate goal is to prototype an intrusion detection and prevention method that can be applied to existing software systems without expending additional development effort, by transforming applications with automated source code annotators or instrumentors. The following sections describe the intrusion detection and prevention method that we plan to prototype, along with implementation details and status. Our prototype works on Java applications. We chose Java for this purpose due to the wide availability of test applications in Java, the availability of Java program analysis tools, and the highly structured Java class loading system which makes it easy to instrument and manipulate the execution of programs.

A previous prototype that we have developed implementing a distributed intrusion detection method is described in our related technical report [6].

3.1 Componentization and analysis of applications

For this intrusion detection and prevention method, the first step is to separate the software system into well-defined components and determine how each component is capable of interacting with other components and entities outside the system. Components may not exchange information through channels other than those which are tracked by this security method. Componentizing an application in this way serves several purposes:

1. Componentizing an application prevents information from entering any component without passing through previously determined channels. This allows intrusion prevention modules to be built into

components that prevent the component's behavior from being influenced by any information that has not been scanned for illegal or anomalous characteristics.

2. Some static analysis techniques, which can analyse the intended behavior of components to aid in determining if the component's behavior has been subverted by an intruder, are exponential in complexity. By splitting a system into smaller components and analyzing each component separately, the complexity of the analysis can be reduced.

3. Componentizing an application allows each component to check the behavior of each other component using a different method or algorithm, depending on the suitability of any given algorithm in relation to the component in question.

3.1.1 Separation of components

Status: Placeholder implemented

To implement our intrusion detection and prevention method ("security method"), it is first necessary to identify the components of an application. In our prototype, we identify two types of components: *internal components* and *external components*. External components are portions of a system implemented as separate Java Virtual Machines (JVMs) which communicate through network sockets. Internal components are sets of classes within a JVM that interact with other classes in the same JVM in a limited enough way that the interfaces between the two sets of classes can be identified and isolated. This situation will most often arise when an existing library is integrated into an application, communication with the library is done in the form of API calls, and the library and the rest of the application do not share data structures that can be manipulated by means other than calling the API's methods.

In our prototype, we are treating the Derby database engine (an SQL database engine written in Java) as an external component. Applications that use the Derby database application (in our case, a test web application and Derby's test suite) are considered to be external components. These client applications are then divided into internal components, such that Derby's JDBC driver is in one internal component and the rest of the application's logic is in another internal component.

In a full implementation of this security system, a formal way of identifying and modeling an application's components would be required, either by providing a tool for system administrators that allows them to identify an application's components, or by analyzing the externally observable behavior of a running system to determine where the components lie. To facilitate the development of the prototype, we currently hardcode references to the application's components into the intrusion detection and instrumentation code that we are developing.

3.1.2 Identification of external interfaces

Status: Placeholder implemented

Once external components of an application have been identified, ways that the external components can interact with other components must be identified. These encompass both expected means of interaction (such as through network sockets or data files) and unexpected means of interaction (such as if one component overwrites cache files read by another component). It is important that these interfaces be identified so external components cannot interact through means of interaction not monitored through intrusion detection.

In a full implementation of this security system, Java programs would be analyzed for references to system classes which are known to provide access to the external environment, such as files and network sockets. Places where methods on these classes are invoked would then be identified as external interfaces. In our current prototype, we have determined that the components of our test application can only communicate with network sockets, so we identify the use of Java's network socket classes as external interface access.

3.1.3 Identification of internal interfaces

Status: Placeholder implemented

Once internal components have been identified, the interfaces through which they can interact must also be identified. In the case where Java applications are componentized by separating them into sets of classes, interfaces exist where code in one class calls code in another class, or when code performs some other kind of interaction which is associated with a class in the other set. The latter case includes situations where code from one class interacts with the fields of an object of a class in the other set, as well as situations where code manipulates objects associated with Java's API (such as arrays or collections) that are also manipulated by code in the other set. A full discussion of possible internal interfaces in a Java program is outside the scope of this report.

In our currently prototype, we have determined that the client application code only interacts with Derby's JDBC driver by calling standard database access methods. For this reason, we are not yet performing any additional, automated analysis to determine how internal components communicate.

3.2 Instrumentation of applications

After separating an application's components and locating interfaces between components, it is necessary to add *instrumentation* to the interfaces. We consider instrumentation to be a shim that can be applied at runtime or compile time that mediates communication across the interface. Once all of the interfaces of each component are instrumented, the components can then be isolated from each other at will, and communication between the components can be monitored for intrusions.

As we use the term in this report, *instrumentation* code does not necessarily collect data. Instrumentation code acts as a dynamic reconfiguration point where communication between components can be split, recorded, examined, permitted, or denied as needed at runtime.

3.2.1 Instrumentation of external interfaces

Status: Prototype completed

As mentioned above, external interfaces are points where components can interact with their environment, and by extension, with other components in the system. Because our current prototype is in Java, our external interfaces are the methods provided by the JVM that allow programs to perform operations such as file and network access. In a full implementation of the security system, instrumentation would be performed by transforming the Java class files and replacing references to Java's API classes (that perform external interfacing functions) with classes provided by the security system which allow for instrumentation to be dynamically added or removed as needed. In our current implementation, because we have determined that the only external interfaces which would allow components to communicate are network interfaces, we have developed a replacement for Java's network socket classes.

Our replacement socket classes take the place of Java's default socket implementation, running on the sides of both parties of the socket communication. A single, physical socket connection is established between the two components, and any additional communication takes place by creating *logical* connections which are multiplexed over the physical socket connection. From the perspective of the components, the two components are communicating through ordinary TCP sockets as they expect; however, all network-related calls are intercepted by the logical socket manager, which translates them into messages which are passed across the physical connection. By maintaining only a single physical socket connection, it is possible to preserve the order of messages that may have been reordered during physical transport (such as messages sent across multiple TCP connections). This is important if the other component will verify if it is receiving a valid message sequence (as described later), because the messages and metadata (also described later) can be assured to have been received in the same order as they were sent.

Because Java contains a built-in mechanism to accept an alternative socket implementation, this external component instrumentation in our prototype can be performed without performing significant transformations of the program beforehand. A single method call must be inserted to switch socket implementations upon initialization.

3.2.2 Instrumentation of internal interfaces

Status: Placeholder implemented

As mentioned above, internal interfaces are boundaries where it may be useful to treat existing components as two separate components. Once internal interfaces are identified, the component must be instrumented by rewriting the Java class so method calls from one interface to another go through a proxy class which provides the opportunity to perform instrumentation-related actions when needed. Aspect-oriented programming may be a useful way to implement such instrumentation in a full implementation of the security system. If the two parts of the component internally communicate through the use of shared Java objects or data structures, refactoring is required to first convert these operations into method calls from one part of the component to the other.

In our prototype, the only internal interfaces are when the application's code calls JDBC functions. To facilitate the prototype, we manually added instrumentation in the appropriate spots by adding an instrumentation statement before each such method call. These instrumentation statements collect metadata (as described later) which is transmitted by the alternative network socket implementation.

3.2.3 Preparing interfaces of static, dynamic, and host components for verification

Status: Placeholder implemented

After the interfaces between components have been identified and instrumented, some analysis may be required prior to intrusion detection, depending on the configuration of the application and the intrusion detection strategy to be used.

From the perspective of one component of a multi-component application which is performing intrusion detection, applications are made of three kinds of components:

Host component: The host component receives messages from static and dynamic components, and performs intrusion detection by verifying these messages.

Static components: Messages sent by static components are checked based on the results of a static analysis which was performed at deployment time. The static component is analyzed in order to derive specifications or constraints on the component's behavior. The host component checks these specifications or constraints at runtime by determining if messages received by the host component exceed the bounds computed by the static analysis. To prepare static components for verification, the static analysis must be performed ahead of time, and the data derived from the static analysis must be stored in the host component that will do the runtime checking.

Dynamic components: Messages sent by dynamic components are checked by the host component, which performs dynamic analysis at runtime. When performing this dynamic analysis, the host component has a copy of the dynamic component's code, and the entire sequence of messages that the two components have exchanged is checked for consistency against the code that produces them, determining if a malicious user has subverted the normal execution of this code. As implemented in our prototype, this kind of analysis does not require anything to be precomputed; however the dynamic component must collect metadata (as described later) on interactions it has with components other than the host component. Such metadata can be collected through the instrumentation mechanism.

In our current prototype, we have not implemented a generalized static analysis or component specification tool. In this prototype, the Derby database server is a host component and the client applications are split into two internal components, with the JDBC driver being a dynamic component and the database application code being a static component. This application topology is hardcoded into the instrumentation and verification mechanisms. The JDBC driver code has been isolated into a JAR file which is supplied to the verifier in the host component. The database application code was analyzed by hand and a list of valid method call signatures between the application and the JDBC driver has been prepared. These signatures

can be used to generate or verify metadata when performing verification inside the JDBC driver or Derby server components.

3.3 Verification of inter-component messages

Once the application has been componentized, instrumented, and analyzed, verification can be performed at runtime. Through verification, intrusions can be detected and prevented by observing when one component of an application is no longer behaving as intended.

3.3.1 Developing a verification strategy

Status: Placeholder implemented

Host components verify messages they receive from static and dynamic components to detect and prevent intrusions. This distributed intrusion detection framework allows for multiple verification strategies to be used, even in the same application. Verification strategies fall under two types:

- Specification-based methods: Specification-based methods use some kind of information on a component, derived at deployment time, to check if the component's behavior is consistent with this information. This checking can be done with static or dynamic analysis (which can in turn be performed at either deployment or run time). The information derived on the component may consist of specifications, static analysis results, the component's entire source code (acting as a stand-in for a complete specification), or anything else that can provably describe the intended behavior of the component.
- Non-specification-based methods: Traditional intrusion detection methods can also be applied when verifying inter-component traffic. These include anomaly detection and signature-based detection.

A full implementation of this security system would allow system administrators to choose verification strategies for each component. Our prototype currently hardcodes a verification strategy in the verification components. Under this strategy, the behavior of the dynamic component (the JDBC driver) is verified by the host component (the Derby server) through dynamic analysis. The dynamic component records its interactions with the static component (the application) and sends them as metadata to the host component, which uses the metadata in the simulation and then verifies the metadata against a manually provided set of valid metadata signatures (which we will later extract automatically with static analysis on the static component). This has the effect of checking for intrusions that originated either in the static component or the dynamic component.

3.3.2 Initiating verification on the host component

Status: Prototype completed

When a host component participates in an interaction with a component that is being verified, the verification process must be initiated, in order to determine if the interaction reflects malicious behavior on the part of the verified component. This verification must occur before the host component processes the message, to prevent a possible intrusion from spreading between components before it is detected. To begin the verification process, we use the instrumentation mechanism incorporated into the host component's interface with the verified component. Because interactions can take on many different forms (such as network traffic, reading from files, or reading user input), the instrumentation mechanism provides a universal hook that allows verification to take place whenever it is needed.

In our prototype, we have added logic to the replacement network socket implementation that initiates verification when a message is received by the host component from a verified component.

3.3.3 Transmitting metadata from dynamic components

Status: In progress

In order to verify a dynamic component through simulation, any data entering the component from the outside (or from other components) must be known. The verifying component has no direct knowledge of how the dynamic component has interacted with other components (other than with the verifying component itself), so the dynamic component must store and transmit such information to the verifying component. We refer to such information as *metadata*, because it is transmitted to the verifying component as an augmentation or annotation of the other information that the dynamic component is transmitting.

Metadata must be collected whenever a dynamic component receives information from the outside or from another component other than the verifying component. Whenever the dynamic component sends information to the verifying component, it must send all metadata in the order collected. The instrumentation mechanisms are used to collect the metadata, send the metadata, and ensure that the events being recorded are ordered (especially in the case of multithreaded components) so they can be simulated in the same order during the verification.

Although information sent from the verifying component to the dynamic component need not be re-sent to the same verifying component as metadata, the time that the information was received (in relation to when other metadata was collected) may need to be recorded as metadata, in cases where the information is received through a blocking system call. For example, if the dynamic component and the verifying component are communicating through network sockets, the dynamic component must record if it received messages from the verifying component before or after receiving messages from other components, so the exact sequence of events can be replicated on the client.

In our prototype, whenever the database client application makes a method call to the JDBC driver, metadata is collected identifying the method that was called and the parameters that were used in the call. This metadata is collected by statements manually inserted in the client application before each method call. When the JDBC driver sends a network message to the Derby server, the metadata buffer is examined by the alternative network socket implementation and metadata is attached to the message when present. This method successfully collects metadata related to the parameters of single database connections and statement executions; however, additional work is required to properly identify the target objects of method calls (and the objects' counterparts in the verification simulator) when multiple database objects are manipulated simultaneously.

3.3.4 Building a simulator to verify a dynamic component

Status: In progress

By simulating a dynamic component, information received by the verifying component from the dynamic component can be verified to ensure that it is consistent with what the developer originally intended. Inconsistencies would indicate that the security of the dynamic component has been compromised and a malicious user is attempting to induce unwanted behavior in the verifying component.

For each component of the system, note that all of the component's interactions with other components and the component's environment were identified during the interface identification phase. These interfaces were then instrumented during the instrumentation phase. The instrumentation mechanism provides us with a way to run components in an isolated simulation setting, outside their original environment, in order to measure any observable differences between their simulated behavior and their actual behavior. This provides a verification method.

In our prototype, we are implementing the simulator by storing the dynamic component's code (JDBC driver) in a JAR file and then implementing a custom classloader to run the code from a simulating verifier class. The only interactions that this code has with the environment are through incoming API calls and network socket communication. Network socket communications are intercepted by the replacement network socket implementation already present in the JVM (the host component's JVM where the verification is being performed), which examines the call stack upon being invoked to determine if the host component or the simulated dynamic component is invoking it. API calls are initiated by a simulation manager which reads

the incoming metadata from the dynamic component, which contains method call targets and parameters, and makes the same method calls on the simulated components.

In order to verify that an incoming network message is not anomalous, the verifier must perform two steps:

1. Determine that the externally observable behavior of the dynamic component is consistent with the metadata
2. Determine that the metadata (which encodes the behavior of the static component) is consistent with the static component's specification

To perform the first task, the simulator examines the network messages leaving the simulated dynamic component and ensures that they are identical (and in the same order) as the network messages leaving the actual dynamic component. Recall that several logical network connections are multiplexed into one physical network connection. The messages being verified are those which were transmitted over the physical connection, which encode both data transmission – through multiple logical connections – and control activities such as establishing new connections. Therefore, the verifier ensures that all activities that may influence the host component's behavior – even control activities such as closing a connection – are first verified. Because the simulated dynamic component received the same inputs as the actual dynamic component, the outputs will be identical if the component's behavior was not subverted by an attacker (assuming that true environmental inputs and source of nondeterminism, such as reordering of network packets, are replicated through metadata).

To perform the second task, the simulator ensures that the metadata is consistent with the specification of the static component. This covers the case where the internal component corresponding to the database client's application logic has been compromised. (This verification could also have been performed within the JDBC driver's internal component.) In the current implementation of our prototype, a list of valid JDBC method calls and parameters is manually supplied. Therefore, the metadata can be easily verified by comparing the reported method calls with those on the list.

3.4 Exploit response and mitigation

Status: Design complete

After detecting an intrusion through an inconsistency or anomaly in some information received by the host component, the verifier can decide how it will react to the intrusion. Possible responses include denying passage of the undesired information, shutting down the host component, ignoring further communications from the source of the undesired communication, reverting all system components to a known "safe state", or notifying the system administrator of the intrusion.

In our prototype, we will ignore any further messages from a component which has sent a message that was associated with an intrusion. This can be done by closing the physical network connection with this component, preventing any logical connections from being created or used.

4 Evaluation of methods to mitigate vulnerabilities

Previous researchers have developed numerous methods to improve the security of a software system by detecting and preventing intrusions. As we are developing intrusion detection and prevention methods, we are also interested in *evaluating* the performance of our methods as compared to those that already exist. However, quantitative methods for evaluating and comparing intrusion detection and prevention methods are few, and those that do exist are specialized for evaluating a single class of intrusion detection products (such as packet inspection devices). Therefore, we also plan to investigate methods for evaluating these technologies.

4.1 Direct testing through a corpus of vulnerabilities

One method to test intrusion detection and prevention tools is to install the tools on a collection of applications with known vulnerabilities and then to attempt to exploit the vulnerabilities in these applications. We have collected a corpus of security vulnerabilities in PHP web applications, and we are creating automation tools to easily perform experiments on the corpus. Testing existing intrusion prevention and detection tools is one possible application for this corpus.

We summarize the task of evaluating intrusion prevention systems here. A more detailed description of this task can be found in our related work. [7]

4.1.1 An exploit verification tool

Status: Prototype completed

To perform an experimental evaluation of intrusion detection and prevention tools using a corpus of existing vulnerabilities, it is necessary to automatically attempt exploits against known vulnerabilities and then detect if the exploits succeeded while the security tool was active. To do this, an automated method of determining if the exploit succeeded is necessary.

Exploit success detectors can either check for evidence that the exploit's payload was successfully executed (by verifying that a system's data was successfully stolen or modified) or for evidence that the delivery mechanism of the exploit succeeded (by looking for a telltale signature in the application's output or database calls). We chose the latter method because it eliminates the need to develop tailored exploit payloads for each application. In the case of SQL injection attacks, we detect exploit success by searching for an injected SQL string in a log of database activity. In the case of cross-site-scripting attacks, we detect exploit success by searching for an injected script tag in the web application's HTML output.

4.1.2 Comparing the performance of intrusion detection and prevention methods

Status: Planned

Once a vulnerability corpus has been compiled and an exploit verification mechanism has been added to it, the performance of various intrusion detection and prevention methods can be compared. We plan to perform such an evaluation to compare the distributed intrusion detection method with existing intrusion detection methods. To do this, we will update our exploit verifier and intrusion prevention testing tools for the latest version of our corpus, because they were developed for a previous version of the corpus. We will also ensure that the intrusion prevention tools are compatible with the applications in our corpus, finding new vulnerabilities or tools if they are incompatible (for reasons such as using an incompatible programming language).

4.2 Indirect measurement by measuring attack surface reduction

Status: Planned

The direct method of evaluating intrusion detection technologies suffers from the shortcoming of requiring a corpus with a representative and valid mix of vulnerabilities. Because software development trends and vulnerabilities change over time, and because some intrusion detection tools are narrowly tailored for the vulnerabilities prevalent when they are written (making the performance of the tools especially sensitive to the mix of vulnerabilities in the corpus), it is difficult to give a tool a numerical score that remains stable over time. However, a good attack surface metric will measure code structures and designs which tend to result in security vulnerabilities, and such characteristics may remain applicable over time as software development trends change.

We plan to investigate the possibility of evaluating intrusion prevention technologies based on their potential to reduce attack surface. Technologies such as firewalls have an obvious and direct impact on attack surface, at least when open ports contribute to an attack surface metric. However, technologies that make it harder to subvert the execution of a system, such as hardened versions of libraries or frameworks, may have a measurable impact on the system's security by reducing the system's attack surface. More sophisticated

attack surface metrics, such as those that measure an attacker's ability to drive program execution toward desired functions or branches, are more suitable for measuring this kind of reduction. We plan to evaluate several security tools against the corpus of security vulnerabilities, and then investigate if evaluated attack surface metrics are able to predict which security tools are better at mitigating vulnerabilities in applications in the corpus. (The question of predicting which applications will contain vulnerabilities in the first place is investigated separately.) This may provide evidence supporting the notion that attack surface metrics could be used to predict the effectiveness of intrusion prevention technologies.

References

[1] T. Heumann, S. Türpe, and J. Keller. Quantifying the attack surface of a web application. In *ISSE/Sicherheit 2010: Information Security Solutions Europe - Sicherheit, Schutz und Zuverlässigkeit.*, volume P-170 of *Lecture Notes in Informatics (LNI)*, pages 305–316. Gesellschaft für Informatik (GI) e.V., Bonner Köllen Verlag, 2010.

[2] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, May 2006.

[3] P. Manadhata and J. Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371 –386, May-June 2011.

[4] Selinux. http://selinuxproject.org.

[5] M.-O. Stehr, C. Talcott, J. Rushby, P. Lincoln, M. Kim, S. Cheung, and A. Poggio. Fractionated software for networked cyber-physical systems: Research directions and long-term vision. In G. Agha, O. Danvy, and J. Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 110–143. Springer Berlin / Heidelberg, 2011.

[6] J. Stuckman and J. Purtilo. Detecting runtime anomalies in AJAX applications through trace analysis. Technical Report CS-TR-4989, University of Maryland, College Park, 2011.

[7] J. Stuckman and J. Purtilo. A testbed for the evaluation of web intrusion prevention systems. In *Proceedings of MetriSec 2011 International Workshop on Security Measurements and Metrics*, pages 66 –75. IEEE, Sept 2011.

[8] J. Stuckman and J. Purtilo. Comparing and applying attack surface metrics. In *(to appear) Proceedings of MetriSec 2012 International Workshop on Security Measurements and Metrics*, 2012.