Measuring the Attack Surfaces of SAP Business Applications

Pratyusa K. Manadhata[1] **Yuecel Karabulut**[2]
Jeannette M. Wing[1]

May 2008
CMU-CS-08-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]School of Computer Science, Carnegie Mellon Univeristy, Pittsburgh, PA
[2]SAP Research Palo Alto, Palo Alto, CA

This research was sponsored by the US Army Research Office (ARO) under contract no. DAAD190210389, SAP Labs, LLC under award no. 1010751, and the Software Engineering Institute. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity. This material was based on work partially supported by the National Science Foundation, while the third author was working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Keywords: Attack Surface, Attack Surface Metric, SAP Business Applications, Security Metrics, Security Risk Mitigation, Software Security, Software Quality

Abstract

Software vendors such as SAP are increasingly concerned about mitigating the security risk of their software. Code quality improvement is a traditional approach to mitigate security risk; measuring and reducing the *attack surface* of software is a complementary approach. In this paper, we introduce a method for measuring the attack surfaces of SAP business applications implemented in `Java`. We implement a tool as an Eclipse plugin to measure an SAP software system's attack surface in an automated manner. We demonstrate the feasibility of our approach by measuring the attack surfaces of three versions of an SAP software system. SAP's software developers can use the tool as part of the software development process to improve software quality and security. SAP's customers can also use the tool to mitigate their security risk.

1 Introduction

There is a growing demand for secure software as we are increasingly dependent on software in our day-to-day life. Software industry has responded to the demand by increasing effort for creating "more secure" products and services (e.g., Microsoft's Trustworthy Computing Initiative and SAP's Software LifeCycle Security efforts). How can industry determine whether this effort is paying off and how can consumers determine whether industry's effort has made a difference? We need security metrics and measurements to gauge progress with respect to security; software developers can use metrics to quantify the improvement in security from one version of their software to another and software consumers can use metrics to compare alternative software that provide the same functionality.

While it is very difficult to devise security metrics that definitively measure the security of software [9], prior work has shown that a system's *attack surface measurement* is an indicator of the system's security [11, 15]. Michael Howard of Microsoft informally introduced the notion of a system's attack surface [11]; Manadhata and Wing of Carnegie Mellon University (CMU) formalized and generalized Howard's notion and proposed an abstract but systematic attack surface measurement method [13]. Manadhata and Wing also defined a concrete method for measuring the attack surfaces of systems implemented in `C` and applied the method to two open source FTP servers and two open source IMAP servers [16, 14].

SAP and CMU collaborated to apply CMU's attack surface measurement method to SAP's business applications and platforms. SAP is the world's largest enterprise software company with more than 46,100 customers worldwide [2]. SAP provides a comprehensive range of enterprise software and business applications covering all aspects of the customers' businesses. The business applications have a varied customer base of small, medium, and large enterprises and support a wide range of business functionalities; hence the applications are large in size and complex in their design [3].

1.1 Motivation

Software vendors have traditionally focused on improving code quality for improving software security and quality. The code quality improvement effort aims toward reducing the number of design and coding errors in software. An error causes software to behave differently from the intended behavior as defined by the software's specification; a vulnerability is an error that can be exploited by an attacker. In principle, we can use formal correctness proof techniques to identify and remove all errors in software with respect to a given specification and hence remove all its vulnerabilities. In practice, however, building large and complex software devoid of errors, and hence security vulnerabilities, remains a very difficult task. First, specifications, in particular explicit assumptions, can change over time so something that was not an error can become an error later. Second, formal specifications are rarely written in practice. Third, formal verification tools used in practice to find and fix errors, including specific security vulnerabilities such as buffer overruns, usually trade soundness for completeness or vice versa. Fourth, we do not know the vulnerabilities of future, i.e., the errors present in software for which exploits will be developed in the future.

Software vendors have to embrace the hard fact that their software will ship with both known and future vulnerabilities in them and many of the vulnerabilities will be discovered and exploited. They can, however, minimize the risk associated with the exploitation of these vulnerabilities. One way to minimize the risk is by reducing the attack surfaces of their software. A smaller attack

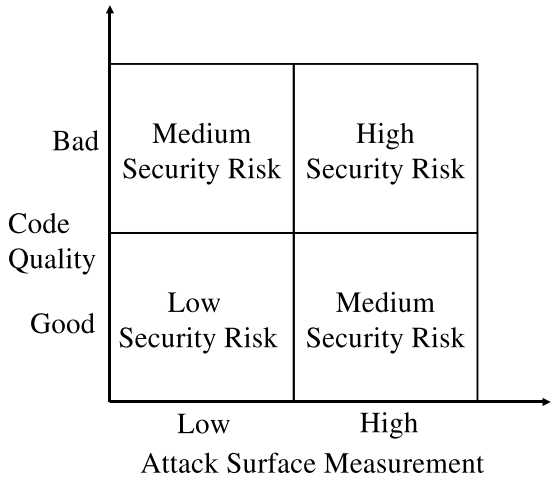

Figure 1: Attack Surface Reduction and Code Quality Improvement are complementary approaches for improving software security.

surface makes the exploitation of the vulnerabilities harder and lowers the damage of exploitation, and hence mitigates the security risk. As shown in Figure 1, the code quality effort and the attack surface reduction approach are complementary; a complete risk mitigation strategy requires a combination of both.

There is anecdotal evidence from the software industry to demonstrate that the reduction in a software system's attack surface mitigates the system's security risk. For example, in case of Microsoft, the Sasser worm, the Zotob worm, and the Nachi worm did not affect some versions of Windows due the reduction in the attack surface of Windows [12]. Similarly, both Firefox 2.0 and Firefox 1.5 contained a buffer overflow vulnerability in the Mozilla Network Security Services (NSS) code for processing the SSL 2 protocol. Firefox 2.0, however, is immune to attacks that exploit the vulnerability due to the reduction in its attack surface [17].

The motivation behind the collaboration is two-fold. First, we demonstrate that the attack surface measurement method scales to enterprise-scale software. Second, we have the opportunity to interact closely with SAP's software developers and architects and get their feedback on our measurement process. There are two goals of the collaboration.

1. The short-term goal is to instantiate Manadhata and Wing's abstract method to obtain a measurement method for SAP software systems and to apply the method to SAP business applications. The measurement process will also identify possible ways to reduce the business applications' attack surfaces.

2. The long-term goal is to evaluate the possibility of integrating the measurement process with SAP's software development process so that SAP's software architects and developers can use the measurement process as a tool in a regular basis.

1.2 Contributions and Roadmap

We make the following key contributions in this paper.

1. We introduce a systematic method for measuring the attack surfaces of SAP business applications implemented in `Java`.

2. We implement a tool to measure a business application's attack surface from the application's source code in an automated manner.

3. We demonstrate the utility of the measurement method and the tool by measuring and comparing the attack surfaces of three versions of an SAP software system.

4. We identify future avenues of research to make the measurement method and the tool more useful in practice.

The rest of this paper is organized as follows. We briefly describe Manadhata and Wing's abstract attack surface measurement method in Section 2. In Section 3, we describe our choice of the SAP software system used in the collaboration. We introduce a method to measure the attack surfaces of SAP software systems in Section 4. We describe the implementation of a tool to measure the attack surface in Section 5. We discuss the measurement results in Section 6 and our recommendations in Section 7. In Section 8, we describe various lessons learned from the collaboration and discuss possible avenues of future work. We conclude with a summary in Section 9.

2 Abstract Attack Surface Measurement Method

We know from the past that many attacks, e.g., exploiting a buffer overflow, on a system take place by sending data from the system's operating environment into the system. Similarly, many other attacks, e.g., symlink attacks, on a system take place because the system sends data into its environment. In both these types of attacks, an attacker connects to a system using the system's *channels* (e.g., sockets), invokes the system's *methods* (e.g., API), and sends *data items* (e.g., input strings) into the system or receives data items from the system. An attacker can also send data indirectly into a system by using data items that are persistent (e.g., files). An attacker can send data into a system by writing to a file that the system later reads. Similarly, an attacker can receive data indirectly from the system by using shared persistent data items. Hence an attacker uses a system's methods, channels, and data items present in the system's environment to attack the system. We collectively refer to a system's methods, channels, and data items as the system's *resources* and thus define a system's attack surface in terms of the system's resources (Figure 2).

Not all resources, however, are part of the attack surface. A system's attack surface is the subset of the system's resources that an attacker can use to cause damage to the system. Manadhata and Wing introduce an *entry point and exit point framework* to identify these relevant resources. Moreover, not all resources contribute equally to the measure of a system's attack surface. A resource's contribution to the attack surface reflects the likelihood of the resource being used in attacks. For example, a method running with `root` privilege is more likely to be used in attacks than a method running with `non-root` privilege. Manadhata and Wing introduce the notion of a *damage potential-effort ratio* to estimate a resource's contribution to the attack surface. A system's attack surface measurement is the total contribution of the resources along the methods, channels, and data dimensions; the measurement indicates the level of damage an attacker can potentially cause to the system and the effort required for the attacker to cause such damage.

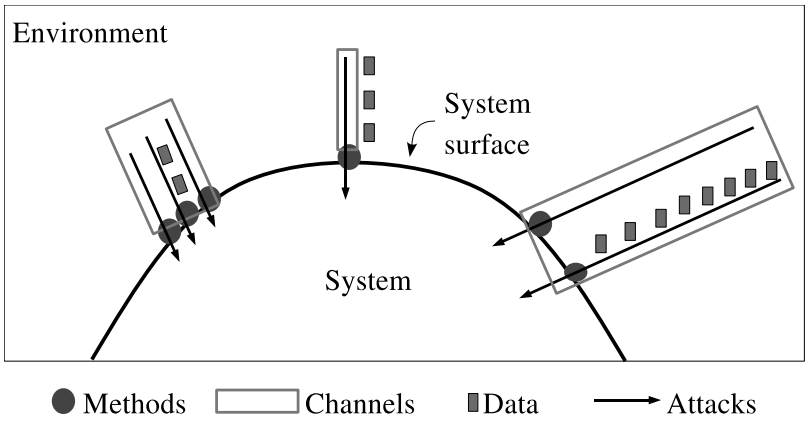

Figure 2: Intuitively, a system's attack surface is the subset of the system's resources (methods, channels, and data) used in attacks on the system.

A system's attack surface measurement does not represent code quality; hence a large attack surface measurement does not imply that a system has many vulnerabilities and few vulnerabilities in a system does not imply a small measurement. Instead, a larger attack surface measurement indicates that an attacker is likely to exploit the vulnerabilities present in the system with less effort and cause more damage to the system. Since a system's code is likely to contain vulnerabilities, it is prudent to reduce the system's attack surface measurement in order to mitigate the security risk.

2.1 Attack Surface Definition

Consider a set, S, of systems, a user, U, and a data store, D. For a given system, $s \in S$, we define its environment, $E_s = \langle U, D, T \rangle$, to be a three-tuple where $T = S \setminus \{s\}$ is the set of systems excluding s. The system s interacts with its environment E_s; hence we define the entry points and exit points of s with respect to E_s. Figure 3 shows a system, s, and its environment, $E_s = \langle U, D, \{s_1, s_2, \}\rangle$. For example, s could be a web server and s_1 and s_2 could be an application server and a directory server, respectively.

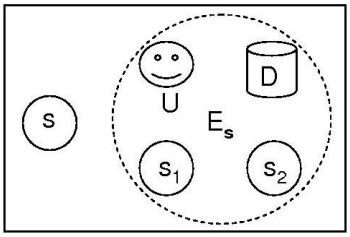

Figure 3: A system, s, and its environment, E_s.

Every system in S has a set of methods. A method of a system receives arguments as input and returns results as output. Examples of methods are the API of a system. Every system also has a set of communication channels. The channels of a system s are the means by which the user U or any other system in the environment communicates with s. Examples of channels are

TCP/UDP sockets, RPC end points, and named pipes. The user U and the data store D are global with respect to the systems in S. The data store is a collection of *data items*. Examples of data items are strings, URLs, files, and cookies. We model the data store D as a separate entity to allow sharing of data among all the systems in S. For simplicity, we assume only one user U is present in the environment. U represents the adversary who attacks the systems in S.

2.1.1 Entry Points

The methods of a system that receive data items from the system's environment are the system's entry points. A method of a system can receive data directly or indirectly from the environment. A method, m, of a system, s, receives a data items *directly* if either (a) the user U or a system, s_1, in the environment invoke m and passes data items as input to m, or (b) m reads data items from the data store, or (c) m invokes the API of a system s_1 in the environment and receives data items as results returned. A method is a *direct entry point* if it receives data items directly from the environment. Few examples of the direct entry points of a web server are the methods in the API of the web server, the methods of the web server that read configuration files from the file system, and the methods of the web server that invoke the API of an application server.

A method, m, of s receives data items *indirectly* if either (a) a method, m_1, of s receives a data item, d, directly, and either m_1 passes d as input to m or m receives d as result returned from m_1, or (b) a method, m_2, of s receives a data item, d, indirectly, and either m_2 passes d as input to m or m receives d as result returned from m_2. A method is a *indirect entry point* if it receives data items indirectly from the environment. For example, a method in the API of the web server that receives login information from a user might pass the information to another method in the authentication module; the method in the authentication module is an indirect entry point. The set of entry points of a system is the union of the set of direct entry points and the set of indirect entry points.

2.1.2 Exit Points

The methods of a system that send data items to the system's environment are the system's exit points. For example, a method that writes into a log file is an exit point. A method of a system can send data directly or indirectly into the environment. A method, m, of a system, s, sends a data items *directly* if either (a) the user U or a system, s_1, in the environment invoke m and receive data items as results returned from m, or (b) m writes data items to the data store, or (c) m invokes the API of a system s_1 in the environment and passes data items as input to s_1's API. A method m of s is a *direct exit point* if m sends data items directly to the environment. A method, m, of s sends data items *indirectly* if either (a) m passes a data item, d, as input to a method, m_1, of s or m_1 receives d as result returned from m , and m_1 passes d directly to s's environment, or (b) m passes a data item, d, as input to a method, m_2, of s or m_2 receives d as result returned from m, and m_2 passes d indirectly to s's environment. A method m of s is a *indirect exit point* if m sends data items indirectly to the environment. The set of exit points of a system is the union of the set of direct exit points and the set of indirect exit points.

2.1.3 Channels

An attacker uses a system's channels to connect to the system and attack the system. Hence a system's channels act as another basis for attacks. An example of a channel of an IMAP server is

the TCP socket opened by the IMAP server.

2.1.4 Untrusted Data Items

The data store D is a collection of persistent and transient data items. The data items that are visible to both a system s and the user U across different executions of s are the persistent data items. Specific examples of persistent data items are files, cookies, database records, and registry entries. The persistent data items are shared between s and U, hence U can use the persistent data items to send (receive) data indirectly into (from) s. Hence the persistent data items act as another basis for attacks on s. An *untrusted data item* of a system s is a persistent data item d such that a direct entry point of s reads d from the data store or a direct exit point of s writes d to the data store. The configuration files of an IMAP server are examples of the IMAP server's untrusted data items.

Notice that the attacker sends (receives) the transient data items directly into (from) s by invoking s's direct entry points (direct exit points). Since the direct entry points (direct exit points) of s act as a basis for attacks on s, we do not consider the transient data items as a basis for attacks on s.

2.2 Attack Surface

The set of entry points and exit points, the set of channels, and the set of untrusted data items are the resources that the attacker can use to either send data into the system or receive data from the system and hence attack the system. Hence given a system, s, and its environment, E_s, s's attack surface is the triple, $\langle M, C, I \rangle$, where M is the set of entry points and exit points, C is the set of channels, and I is the set of untrusted data items of s.

2.3 Attack Surface Measurement Method

A naive way of measuring a system's attack surface is to count the number of resources that contribute to the attack surface. This naive method is misleading as it assumes that all resources contribute equally to the attack surface. In real systems, not all resources contribute equally to the attack surface. For example, a method, m_1, running as `root` is more likely to be used in an attack than a method, m_2, running as `non-root`; hence m_1 contributes higher to the attack surface than m_2.

We estimate a resource's contribution to a system's attack surface as a *damage potential-effort ratio* where *damage potential* is the level of harm the attacker can cause to the system in using the resource in an attack and *effort* is the amount of work done by the attacker to acquire the necessary access rights in order to be able to use the resource in an attack. The higher the damage potential, the higher the contribution; the higher the effort, the lower the contribution.

In practice, we estimate a resource's damage potential and effort in terms of the resource's attributes. Examples of attributes are method privilege, access rights, channel protocol, and data item type. In case of systems implemented in `C`, we estimate a method's damage potential in terms of the method's *privilege*. An attacker gains the same privilege as a method by using a method in an attack. For example, the attacker gains `root` privilege by exploiting a buffer overflow in a method running as `root`. The attacker can cause damage to the system after gaining `root` privilege. Similarly, we estimate a channel's damage potential in terms of the channel's *protocol* and a data

item's damage potential in terms of the data item's *type*. The attacker can use a resource in an attack if the attacker has the required *access rights*. The attacker spends effort to acquire these access rights. Hence for the three kinds of resources, i.e., method, channel, and data, we estimate the effort the attacker needs to spend to use a resource in an attack in terms of the resource's access rights.

We assume that we have total orderings among the values of each of the six attributes, i.e., method privilege and access rights, channel protocol and access rights, and data item type and access rights. In practice, we impose these total orderings using our knowledge of a system and its environment. For example, an attacker can cause more damage to a system by using a method running with `root` privilege than a method running with `non-root` privilege; hence `root` $\succ$ `non-root`. We use these total orderings to assign numeric values to the attributes and estimate numeric damage potential-effort ratios using the numeric values.

Our abstract attack surface measurement method consists of the following three steps.

1. Given a system, s, and its environment, E_s, we identify a set, M, of entry points and exit points, a set, C, of channels, and a set, I, of untrusted data items of s.
2. We estimate the damage potential-effort ratio, $der_m(m)$, of each method $m \in M$, the damage potential-effort ratio, $der_c(c)$, of each channel $c \in C$, and the damage potential-effort ratio, $der_d(d)$, of each data item $d \in I$.
3. The measure of s's attack surface is the triple $\langle \sum_{m \in M} der_m(m), \sum_{c \in C} der_c(c), \sum_{d \in I} der_d(d) \rangle$.

3 Choice of an Enterprise Software System

Choosing an appropriate software system was a vital step in our collaboration with SAP. The choice of a system was guided by three requirements. First, the chosen system should be a heavily used system so that reduction in its attack surface benefits a large number of SAP customers by reducing their security risk. Second, the chosen system should be representative of SAP's software systems; measuring the system's attack surface will guide us in measuring the attack surfaces of other SAP systems. Third, the product development group in charge of the system should be committed to the collaboration.

We had discussions with six different SAP product development groups before making our choice. We identified a component of the SAP NetWeaver platform as the enterprise software system to be used in our collaboration [4]. SAP NetWeaver is the common technology platform for SAP business applications; the platform provides the run time environment for all SAP applications. The platform enables development, life cycle management, composition, and integration of SAP business applications [5]. Our chosen component is a core building block of the NetWeaver platform and provides a critical service inside the platform. Henceforth, we refer to the chosen component as *the service*.

4 Measurement Method for SAP Software Systems

In this section, we introduce a method to measure the attack surfaces of SAP software systems implemented in `Java`. A majority of SAP systems are implemented in `Java`, `JavaScript`, and `ABAP` (a proprietary SAP language). We leave the measurement methods for systems implemented in `JavaScript` and `ABAP` for future work.

Manadhata and Wing instantiated the abstract method discussed in Section 2.3 to obtain a measurement method for systems implemented in `C` [14]. Figure 4 shows the steps in their `C` measurement method.

We instantiate the abstract method of Section 2.3 to obtain a measurement method for `Java`. Our measurement method has the same steps as the `C` measurement method. The implementations of the steps, however, are different for the method dimension; the channel dimension and the data dimension remain unchanged.

Recall that the two key steps in measuring the attack surface along the method dimension are (1) the identification of the entry points and the exit points (Section 4.1), and (2) the estimation of the damage potential-effort ratio of the entry points and the exit points (Section 4.2). We describe these two steps of our new method in the following paragraphs.

4.1 Identification of Entry Points and Exit Points

A direct entry point of a system is a method that receives data items directly from the system's environment. A method, m, of a system, s, implemented in `Java` can receive data items in three different ways: (a) m is a method in s's public *interface* and receives data items as input, (b) m invokes a method in the interface of a system, s', in the environment and receives data items as result, and (c) m invokes a `Java I/O` library method. For example, a method, m, is a direct entry point if m invokes the `read` method of the `java.io.DataInputStream` class.

A direct exit point of a system is a method that sends data items directly to the system's environment. A method, m, of a system, s, implemented in `Java` can send data items in three different ways: (a) m is a method in s's public interface and sends data items as result, (b) m invokes a method in the interface of a system, s', in the environment and sends data items as input, and (c) m invokes a `Java I/O` library method. For example, a method, m, is a direct exit point if m invokes the `write` method of the `java.io.DataOutputStream` class.

Given a system, s, we generate s's call graph starting from the methods in s's public interface. From the call graph, we identify all methods of s that invoke either a method in the interface of a system, s', in s's environment or a `Java I/O` library method. These methods are s's direct entry points and direct exit points.

We implemented a tool, described in Section 5, to measure the attack surfaces of SAP systems in an automated manner. The tool identifies only direct entry points and direct exit points of a system. No existing code analysis technique enabled us to identify the indirect entry points and the indirect exit points; we leave the identification of the indirect entry points (exit points) as future work. Hence our measurement is an under-approximation of the measure of the attack surface.

4.2 Estimation of the Damage Potential-Effort Ratio

In case of systems implemented in `C`, we estimate a method's damage potential using the method's privilege and the attacker effort using the method's access rights. In case of the SAP systems, however, a method's privilege is not a useful estimate of the method's damage potential. The entire code base of the NetWeaver platform runs with the same privilege, i.e., the privilege of the application server hosting the platform. If we were to estimate the damage potential using the privilege, we could not make any meaningful suggestion to reduce the attack surface. Hence we do not use a method's privilege in estimating the method's damage potential. Instead, we estimate a method's damage potential using the method's *sources of input data (destinations of output data).*

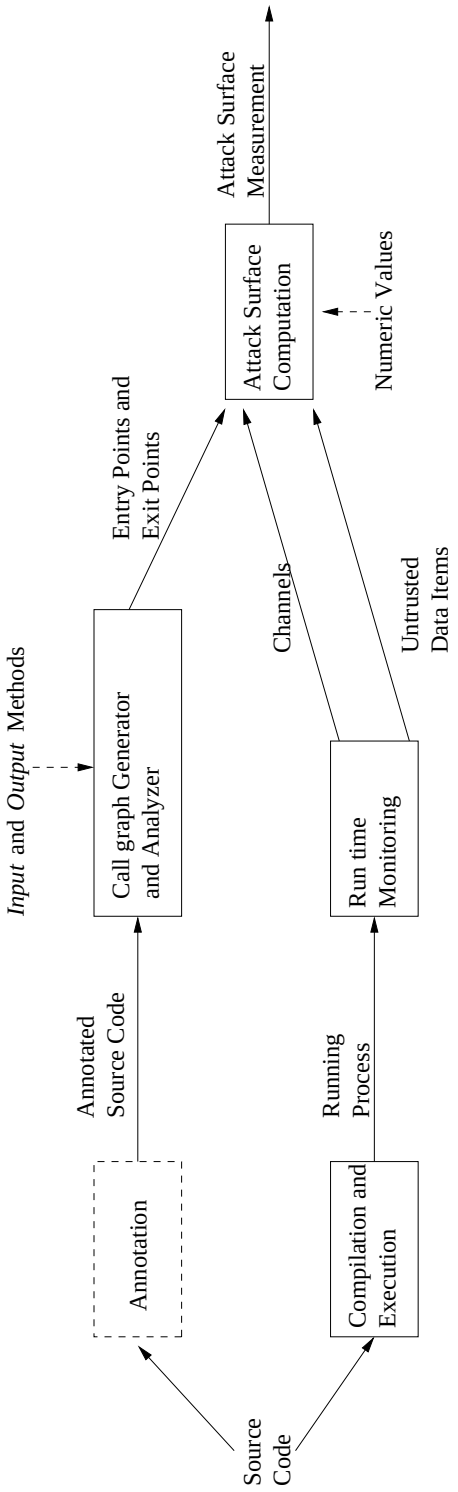

Figure 4: Steps in the attack surface measurement method for systems implemented in `C`.

A method can receive (send) data items from (to) three sources: an input parameter, the data store, and other systems present in the environment. For example, a method receives data items from an attacker as an input parameter in case of SQL injection attacks and Cross Site Scripting (XSS) attacks whereas the method receives data items from the data store in case of File Existence Check attacks. Methods in SAP systems can have three different sources of input: `parameter`, `data store`, and `other systems`.

Similar to systems implemented in `C`, we use a method's access rights level to estimate the attacker effort. A typical SAP system has two different types of interfaces: (1) public interfaces that can be accessed by all entities belonging to any NetWeaver *role* and (2) internal interfaces that can be accessed by only other components of the NetWeaver platform. Hence the methods in SAP systems can be accessed with two different access rights levels: `public` access rights level for methods in public interfaces and `internal` access rights level for methods in internal interfaces.

In case of an SAP system, we annotate the system's code base to indicate the access rights levels of the system's interfaces. We generate the call graph of the annotated code base and determine the sources of input and the access rights levels of the methods from the call graph. Notice that a method may have multiple sources of input. Similarly, a method may be accessible with multiple access rights levels. We identify such a method as a direct entry point (direct exit point) multiple times.

Similar to systems implemented in `C`, we impose total orderings among the sources of input and the access rights levels and assign numeric values to the sources of input and access rights levels in accordance to the total orderings. The exact choice of the numeric values is subjective and depends on a system and its environment. We discuss a specific way of assigning the numeric values in case of the service in Section 5.3. We estimate a method's damage potential-effort ratio from the numeric values assigned to the method's source of input and access rights level.

5 Implementation of a Measurement Tool

In this section, we describe a tool we implemented to measure the attack surfaces of SAP systems implemented in `Java`. There are two key objectives that guided the implementation of the tool: (1) the tool should be an integral part of the software development process, and (2) the software developers and architects can use the tool easily and frequently without spending too much time and effort. The software developers and architects at SAP are already under time pressure; hence it is crucial for the adoption of the tool that we do not burden them with more work and we do not require them to go out of their normal routine to use the tool.

SAP's software developers use a customized version of the *Eclipse* Integrated Development Environment (IDE) for their software development activities [7]. We implemented our tool as a *plugin* for the Eclipse IDE so that the developers can use the tool inside their software development environment. We show a screen shot of our tool in Figure 5 and describe the tool in details in the following paragraphs.

5.1 Call Graph Generation

A key component of our tool is the generation of a system's call graph from the system's source code. We use two different techniques to generate the call graph to provide a precision-scalability tradeoff to the software developers: the TACLE Eclipse plugin developed at the Ohio State University,

Figure 5: Screenshot of the Attack Surface Measurement tool implemented as an Eclipse plugin.

which gives a very precise call graph, but does not scale well to large programs [10]; and an Eclipse API, which gives a less precise call graph, but scales [8].

The TACLE plugin provides us an off-the-shelf implementation of the *Rapid Type Analysis (RTA)* algorithm to perform type analysis and construct call graphs [6]. The plugin performs the operations on a *Java project* inside a *Java Development Tools (JDT)* environment. The plugin implements the RTA algorithm using an *Abstract Syntax Tree (AST)* analysis [18]; for each `Java` class included in the source code, the algorithm constructs and stores the AST of the class in main memory. Hence the algorithm does not scale well to large software systems that contain a large number of `Java` classes. Moreover, to construct a complete call graph, the plugin requires the source code of all the `Java` libraries and `Java Archives (JAR)` used for the compilation of the code base. In practice, we do not expect the software developers to possess the source code of the libraries required by their software. For example, the service requires 23 SAP JARs for its compilation. It is unrealistic to expect the developers of the service to include the source code of these 23 JARs in their `Java` development project.

The Eclipse IDE has an internal API for the construction of call graphs from `Java` source code. The API is included in the `org.eclipse.jdt.internal.corext.callhierarchy` package of the IDE [8]. The API does not require the source code of the libraries and JARs used by a system's code base; hence scales well to large code bases. The constructed call graph, however, is less precise than the call graph generated by the TACLE plugin. For example, the API does not resolve an interface method to a class method that implements the interface method. We extended the API by finding all class methods that implement an interface method and then including the implementing methods in the call graph. The resulting call graph, however, is an over-approximation of the actual call graph as only one of the class methods will be invoked in place of the interface method. Hence our measurement is an over-approximation of the measure of the attack surface.

5.2 Entry Points and Exit Points Identification

Our tool identifies a system's entry points and exit points from the system's call graph. A method of a system is an entry point (exit point) if the method invokes a method in the public interface of another system present in the environment or a `Java I/O` library method. Hence our tool needs the list of interface methods of other systems and the list of `Java I/O` library methods. We provide the list of methods to the tool through two configuration files: `osmethods.txt` and `dsmethods.txt`.

The service invokes the methods of many SAP systems included in the NetWeaver platform. Invocation of an interface method, however, is only *relevant* to the attack surface measurement if the invocation results in data transfer from the service to other systems in its environment or vice versa. We used the javadoc description of the interfaces and interface methods included in the 23 SAP JAR libraries needed by the service and identified 25 interfaces and 126 interface methods to be relevant to the attack surface measurement. We discussed the methods with the developers of the service to avoid any error and included the methods in the `osmethods.txt` configuration file of our tool.

The service does not read or write any untrusted data items from the data store. Hence there were no `Java I/O` library methods to identify as relevant. But in general, we should provide the list of relevant methods through the `dsmethods.txt` configuration file.

5.3 Numeric Value Assignment

Another key component of our tool is the estimation of the numeric damage potential-effort ratios of a system's entry points and exit points. We estimate the damage potential and the attacker effort in terms of the sources of input and the access rights level, respectively. The tool determines the sources of input and the access rights levels from the system's call graph; the tool, however, requires the numeric values assigned to the different sources of input and the access rights levels to estimate numeric damage potential-effort ratios. We provide these numeric values through a configuration file, `weights.txt`.

The methods of the service have three different sources of input: `parameter`, `data store`, and `other system`. As discussed in Section 4.2, different types of attacks on the service require the methods to have different sources of input. We assign numeric values to the sources of input by correlating the sources with possible attacks on the service.

SAP conducted a threat modeling process for the service. The process identified possible attacks on the service and assigned severity ratings to the attacks. We correlated the sources of inputs with the possible attacks identified by the threat modeling process. For each source of input, we computed the average severity rating of the attacks that require the source of the input. We show the sources of input in the first column and the average severity ratings in the second column of Table 1.

We assigned numeric values to the sources of input in proportion to the average severity ratings. Manadhata and Wing's parameter sensitivity analysis suggests that the difference between the numeric values assigned to successive damage potential levels should be in the range of 3-14 [14]. Hence we chose the midpoint, 8.5, of the range as the difference. For example, we assigned 1 to the source `other system`, and $1 + (3 - 1) \times 8.5 = 18$ to the source `data store`. We show the numeric values in third column of Table 1.

Source of Input	Average Severity Rating	Numeric Value
`other system`	1	1
`data store`	3	18
`parameter`	5	35

Table 1: Numeric values assigned to the sources of input.

The methods of the service can be accessed by two different access rights level: `public` and `internal`. We imposed the following total ordering among the access rights level: `internal` > `public`. The parameter sensitivity analysis suggests that the difference between the numeric values assigned to successive access rights level should be high (15-20). Hence we chose a difference of 17. We show the numeric values assigned to the access rights level in Table 2.

Access Rights Level	Numeric Value
`public`	1
`internal`	18

Table 2: Numeric values assigned to the access rights levels.

We use the numeric values shown in Table 1 and Table 2 to compute the numeric damage

potential-effort ratios. For example, consider an entry point, m, of a system, s. m is a method in s's public interface and has two input parameters; m also invokes three interface methods of a system, s', in the environment. Then m's damage potential is $2 \times 35 + 3 \times 1 = 73$. If m is accessible with the `public` access rights level, then m's damage potential-effort ratio is $73/1 = 73$. Similarly, if m is accessible with the `internal` access rights level, then m's damage potential-effort ratio is $73/18 = 4.05$.

5.4 Usage of the Tool

The software developers can use our tool to generate a system's call graph rooted at any method of the system and to compute the attack surface measurement from the call graph. The developers can choose the method in the *Outline* view window of the Eclipse IDE. Figure 5 shows the two options for measuring the attack surface in the right context menu of the Outline window; the developers can choose the option appropriate for their scalability and precision requirements.

SAP's software systems have multiple interfaces and each interface typically contains multiple methods. To compute the attack surface of an SAP system, we create a new `Java` class with only a `main` method. The `main` method invokes all the methods included in the interfaces of the system. We generate a call graph rooted at the `main` method and compute the attack surface of the system.

The tool generates its output in the form of a text file, containing (1) the system's attack surface measurement, (2) a list of the system's entry points and exit points, and (3) for each entry point (exit point), a list of input sources, the access rights level, and its contribution to the attack surface measurement. The software developers can use the detailed output as a guide in reducing the attack surfaces of their software. For example, they can focus on the top x% of the entry points and the exit points to reduce the attack surface. They can also focus on the top contributing interfaces and components instead of considering the entire code base of the system.

Our tool allows the developers to perform *incremental* measurements. They can measure the increase in the attack surface due to the addition of a new interface method by generating a call graph rooted at the method; they do not have to measure the attack surface of the entire system. They can also measure the potential reduction in the attack surface due to the removal of an interface method.

The tool also allows the developers to consider many *what-if* scenarios during software development. For example, the developers can easily determine the effect of adding a new feature to the system on the system's attack surface. Similarly, while reducing the attack surface, they can consider the removal of different features and the effect of the removal on the attack surface measurement. They can use the incremental measurements to make an informed decision.

6 Results and Discussion

We measured the attack surfaces of three different of the service included in three different versions of the NetWeaver platform. We identify the three version of the service as `S1`, `S2`, and `S3`. The `S1` version is the first released version of the service, followed by `S2` and `S3` versions, respectively.

We only considered the method dimension of the attack surface in our measurement. The three versions of the service do not use any persistent data items and open only one channel, i.e., a `TCP` socket. Hence we did not measure the attack surface along the channel dimension and the data dimension.

The S3 version of the service implements 8 public interfaces and 2 internal interfaces. The S2 and S1 versions implement 9 and 8 public interfaces, respectively, and no internal interfaces. We show the number of entry points and exit points of the three versions for each access rights level in Table 3.

Version	Count	
	Public	Internal
S3	71	4
S2	67	0
S1	63	0

Table 3: The number of entry points and exit points of the three versions of the service for each access rights level.

We estimated the damage potential-effort ratio of each entry point (exit point) as described in Section 5.3; the ratio is the contribution of the entry point to the attack surface. We summed up the contributions of the entry points and the exit points to obtain the attack surface measurement along the method dimension. We show the attack surface measurements in Table 4. The measurements indicate that the S3 version has the highest security risk along the method dimension followed by S2 and S1.

Version	Attack Surface Measurement
S3	5298.44
S2	4687.00
S1	4649.00

Table 4: Attack surface measurements of the three versions of the service.

The S1 version is the first version of the service released to the customers. The S2 version is backward compatible with S1 for the convenience of the customers. Moreover, S2 added new features to S1 resulting in an increase in the number of public interfaces. Hence the set of methods of S2 is a superset of the set of methods of S1 and as shown in Table 4, the attack surface measurement of S2 is greater than S1.

The S3 version is the latest version of the service released to the customers. The S3 version differs from the S2 version in two significant ways: (1) S3 converted a public interface of S2 to an internal interface to mitigate security risk, and (2) S3 added new features to the service resulting in an increase in the number of public interfaces and internal interfaces. If no new features were added, the attack surface measurement of S3 would have been smaller than S2 due to the conversion of a public interface to an internal interface. The increase in the number of total interfaces due to the addition of new features, however, increases the attack surface measurement of S3. Hence as shown in Table 4, the attack surface measurement of S3 is greater than S2.

Notice that S3's attack surface measurement would have been greater than its current measurement if all its interfaces were public; the presence of internal interfaces results in a minimal attack surface measurement. Hence the addition of the internal interfaces was a good design decision that reduced the attack surface measurement and hence mitigated the security risk of the service.

7 Recommendations

The results discussed in the previous section show that our new attack surface measurement approach is feasible for SAP's complex business applications. The relative ordering among the service's three measurements conforms to the expected ordering. The measurement results can be used as a guide to reduce the service's attack surface measurement. We, however, need further research before we can integrate the attack surface measurement method with SAP's software development life cycle; we discuss possible directions for further research in section 8.

A system's attack surface measurement can be used in other contexts besides the obvious, i.e., as an indication of the system's security risk. We describe four possibilities below; a useful direction of future work is to explore other usage of the attack surface measurement.

7.1 Usage by SAP Developers

We envision two possible uses of attack surface measurements by SAP's software developers. First, software developers and architects can use the minimum and the maximum attack surface measurement estimates discussed in Section 8.3 to prioritize software testing effort. For example, if a system's attack surface measurement is closer to the maximum, then they should invest more in testing efforts; if the measurement is closer to the minimum, they can reduce their testing effort.

Second, SAP's software developers can use attack surface measurements as a guide while implementing patches of security vulnerabilities in SAP software systems. A good patch should not only remove a vulnerability from a system, but also should not increase the system's attack surface. Software developers can use our tool to ensure that their patches do not increase the attack surface.

7.2 Usage by SAP Customers

We also envision two possible uses of attack surface measurements by SAP's consumers. First, SAP's customers often customize SAP software by adding new code to SAP software. The customers can use our tool to measure their customized software's attack surfaces. They can get a sense of their security risk by comparing their measurements with the measurements of the original software released by SAP.

Second, software consumers often have to make a choice between several possible configurations of software. For example, SAP business applications can be configured in many different ways; SAP customers choose the configuration best for them. Configuring large enterprize-scale software is a complex process; hence choosing an appropriate configuration is a non-trivial and error-prone task. SAP's customers could use a system's attack surface measurement as a guide in choosing an appropriate configuration. Since a system's attack surface measurement is dependent on the system's configuration, they would choose a configuration that results in a smaller attack surface exposure.

8 Future Work

In this section, we discuss possible avenues of future work.

8.1 Validation of the Method

A key challenge in security metrics research is the validation of the metric. Manadhata and Wing performed three empirical studies to validate the abstract measurement method and the measurement results of systems implemented in `C` [14]. A possible direction of future research is to explore validation ideas in the context of SAP business applications. Validation of SAP business applications' attack surface measurements may turn out to be easier than validation in the general context. We, however, need more research to identify suitable variables to correlate with the business applications' attack surface measurements.

A possible validation approach would be to integrate the attack surface measurement and reduction process with the software development process of an SAP business application and then observe the benefits of attack surface reduction after the release of the application. If the newer version of the application undergoes attack surface reduction, then we would expect to see fewer exploitable vulnerabilities in the newer version than the older versions over similar time periods.

8.2 Improvements of the Measurement Tool

Based on the feedback received from the SAP product development group, we have identified three possible extensions of the tool to make it more useful for software developers. First, the tool currently outputs its result in the form of a text file. We could improve the tool by presenting the results in a graphical window inside the Eclipse IDE so that the developers can access the measurement results within the IDE. Second, we could improve the usability of the tool by implementing a Graphical User Interface (GUI) to update the configuration information required by the tool. Third, the tool currently measures the attack surfaces of systems implemented in `Java`. The tool would be more useful in practice if we were to extend the tool to measure the attack surfaces of software implemented in other languages such as `JavaScript` and `ABAP` [1].

8.3 Attack Surface Range Analysis

The result of our attack surface measurement method guides the software developers to focus on a system's relevant parts to reduce the system's attack surface. For example, the developers can analyze the top contributing entry points and exit points instead of the entire code base to reduce the attack surface. The result, however, does not help in deciding when to stop the reduction process. In order to address this issue, a possible extension of our work is to develop a method to estimate the minimum and the maximum possible attack surface measurements of a system given the system's functionality. We briefly describe such a method in the following paragraph.

In order to estimate the minimum and the maximum attack surface measurements, we need to estimate an entry point's (exit point) minimum and maximum contributions to the attack surface, i.e., we need to estimate the minimum and the maximum damage potential-effort ratios. We can estimate the minimum and the maximum damage potential-effort ratios from the range of numeric values assigned to damage potential and effort. We also need to estimate the appropriate number of entry points and exit points required to implement the system's functionality. In the absence of such an estimate, we can simplify our analysis by assuming that the appropriate number is the same as the observed number of entry points and exit points. Hence we can estimate the minimum and the maximum attack surface measurements by multiplying the number of entry points and exit points with the minimum and the maximum damage potential-effort ratios, respectively.

9 Summary

In summary, we have introduced a method to measure the attack surfaces of SAP software implemented in `Java`. We have implemented a tool as an Eclipse plug-in to measure the attack surface in an automated manner. We have demonstrated the use of the method and the tool by measuring and comparing the attack surfaces of three versions of an SAP software system. We have also learned important lessons on how to improve our measurement method and our measurement tool to make the measurement process more useful in practice.

We view our work as a useful and pragmatic approach for quantifying the security risk of SAP business applications. We believe that our understanding over time will enable us to further refine our measurement approach.

Acknowledgments

We would like to thank Effrat Keren for her enthusiastic collaboration on the project. The project results would not have been possible without her support.

We would like to thank Sherman Yahali and his team for hosting us. We would like to thank the following members of the team for patiently answering our never-ending list of questions: Eyal Gal, Robert Krien, Shani Ozeri, Oren Ronen, Yael Schuldenfrei, and Vitaly Vainer.

We would like to thank Daniel Clemens for his help on issues related to project management.

We would like to thank Ike Nassi, Paul Hofmann, and Anne Hardy for their leadership roles and encouragement.

We would like to thank Michael Hartmann and Sachar Paulus of SAP Product Security for sponsoring the project and for their support and encouragement throughout the project.

Finally, we would like to thank all SAP folks, especially the NetWeaver security group, with whom we had many fruitful discussions during the course of the project.

References

[1] SAP AG. Abap development. `https://www.sdn.sap.com/irj/sdn/abap`. 8.2

[2] SAP AG. SAP - business software solutions applications and services. `http://www.sap.com/index.epx`. 1

[3] SAP AG. SAP - enterprise solutions technology leader. `http://www.sap.com/about/index.epx`. 1

[4] SAP AG. SAP NetWeaver. `http://www.sap.com/platform/netweaver/index.epx`. 3

[5] SAP AG. SAP NetWeaver products. `https://www.sdn.sap.com/irj/sdn/nw-products`. 3

[6] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM. 5.1

[7] Eclipse. Eclipse - an open development platform. `http://www.eclipse.org/`. 5

[8] Eclipse. Eclipse package org.eclipse.jdt.internal.corext.callhierarchy. `http://mobius.inria.fr/eclipse-doc/org/eclipse/jdt/internal/corext/callhierarchy/package-summary.html`. 5.1

[9] Seymour E. Goodman and Herbert S. Lin, editors. *Toward a Safer and More Secure Cyberspace*. The National Academics Press, 2007. 1

[10] PRESTO Research Group. TACLE project. `http://presto.cse.ohio-state.edu/tacle/`. 5.1

[11] M. Howard, J. Pincus, and J.M. Wing. Measuring relative attack surfaces. In *Proc. of Workshop on Advanced Developments in Software and Systems Security*, 2003. 1

[12] Michael Howard. Personal communication, 2005. 1.1

[13] P. K. Manadhata, D. K. Kaynar, and J. M. Wing. A formal model for a system's attack surface. In *Technical Report CMU-CS-07-144*, July 2007. 1

[14] P. K. Manadhata, K. M.C. Tan, R. A. Maxion, and J. M. Wing. An approach to measuring a system's attack surface. In *Technical Report CMU-CS-07-146*, August 2007. 1, 4, 5.3, 8.1

[15] P. K. Manadhata and J. M. Wing. Measuring a system's attack surface. In *Technical Report CMU-CS-04-102*, January 2004. 1

[16] P. K. Manadhata, J. M. Wing, M. A. Flynn, and M. A. McQueen. Measuring the attack surfaces of two FTP daemons. In *ACM CCS Workshop on Quality of Protection*, October 2006. 1

[17] Gervase Markham. Reducing attack surface. `http://weblogs.mozillazine.org/gerv/archives/2007/02/reducing_attack_surface.html`. 1.1

[18] Mariana Sharp, Jason Sawin, and Atanas Rountev. Building a whole-program type analysis in Eclipse. In *Eclipse Technology Exchange Workshop at OOPSLA*, pages 6–10, 2005. 5.1