Report: Measuring the Attack Surfaces of Enterprise Software

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

[1] Carnegie Mellon Univeristy, Pittsburgh, PA, USA
[2] SAP Research, Palo Alto, CA, USA

Abstract. Software vendors 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 apply a method for measuring attack surfaces to enterprise software written 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. We envision our measurement method and tool to be useful to software developers for improving software security and quality.

1 Introduction

There is a growing demand for secure software as we are increasingly dependent on software in our day-to-day life. 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 security vulnerabilities in software. In practice, however, building large and complex software devoid of vulnerabilities remains a very difficult task. 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 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.

Michael Howard of Microsoft introduced the notion of Relative Attack Surface Quotient (RASQ) for the Windows operating system [1]. Pincus and Wing generalized Howard's notion and measured the attack surfaces of seven versions of Windows [2]. Their measurement method, however, was ad-hoc in nature, required a security expert (e.g., Michael Howard for Windows), and was focused on operating systems. Manadhata and Wing of Carnegie Mellon University (CMU)

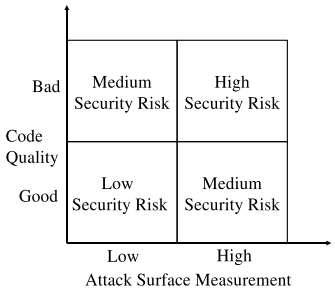

Fig. 1. Attack Surface Reduction and Code Quality Improvement are complementary approaches for improving software security.

formalized Howard's notion and proposed an abstract but systematic attack surface measurement method that does not require a security expert and is applicable to a wide range of software [3].

Intuitively, a system's attack surface is the set of ways in which an adversary can enter the system and potentially cause damage. 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.

To see how well our attack surface method works on enterprise-scale software, SAP and CMU collaborated to apply CMU's attack surface measurement method to SAP's platforms and business applications. Henceforth, we collectively refer to SAP's platforms and business applications as SAP software systems. This collaboration suggested ways to integrate the measurement process with software development process, not just for SAP, which we discuss in Section 6. We describe the collaboration in the rest of this report.

2 Abstract Attack Surface Measurement Method

We briefly describe Manadhata and Wing's abstract measurement method in this section. Please see their technical report for details [3].

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. 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. Not all resources, however, are part of the attack surface. Manadhata and Wing use the entry point and exit point framework to identify the resources that are part of a system's attack surface.

Entry Points Each system has a set of methods. A method receives arguments as input and returns results as output. Examples of methods are the API of a system. A system's methods that receive data items from the system's environment are the system's entry points. For example, a method that receives input from a user or a method that reads a configuration file is an entry point. A method m of a system s is a *direct entry point* if either (a) a user or a system in s's environment invokes m and passes data items as input to m, or (b) m reads from a persistent data item, or (c) m invokes the API of a system in s's environment and receives data items as the result returned. An *indirect entry point* is a method that receives data from a direct entry point.

Exit Points A system's methods 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 m of a system s is a *direct exit point* if either (a) a user or a system in s's environment invokes m and receives data items as results returns from m, or (b) m writes to a persistent data item, or (c) m invokes the API of a system in s's environment and passes data items as input to the API. An *indirect exit point* is a method that sends data to a direct exit point.

Channels Each system also has a set of channels; the channels are the means by which users or other systems in the environment communicate with the system. Examples of channels are TCP/UDP sockets, RPC end points, and named pipes. 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.

Untrusted Data Items An attacker uses persistent data items either to send data indirectly into the system or receive data indirectly from the system. Examples of persistent data items are files, cookies, database records, and registry entries. A system might read from a file after an attacker writes into the file. Similarly, the attacker might read from a file after the system writes into the file. Hence the persistent data items act as another basis for attacks on a system. An *untrusted data item* of a system s is a persistent data item d such that a direct entry point of s reads from d or a direct exit point of s writes into d.

Attack Surface Definition By definition, the set, M, of entry points and exit points, the set, C, of channels, and the set, I, 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, we define s's attack surface as the triple, $\langle M, C, I \rangle$.

Attack Surface Measurement Method Not all resources contribute equally to a system's attack surface. Manadhata and Wing 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.

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` and hence causes damage to the system. 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 attacker effort in terms of the resource's access rights. We assign numeric values to the attributes to compute a numeric damage potential-effort ratio. We describe a specific method of assigning numbers in Section 4.2.

Our abstract measurement method has the following three steps.

1. Given a system, s, and its environment, 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 Measurement Method for SAP Software Systems

In this section, we walk through the steps of our method for measuring the attack surfaces of software services written in `Java`. We keep our discussion general, bringing in the specifics of the SAP application only where necessary.

In our SAP collaboration, we chose a component of the SAP NetWeaver platform as the system whose attack surface is to be measured [4]. The component is a core building block of the platform; henceforth, we refer to the chosen component as *the service*. The service does not use any persistent data items and opens only one channel, i.e., a `TCP` socket. Hence we only considered the method dimension of the attack surface in our measurement. We would, however, consider the three dimensions of the attack surface for a generic `Java` system.

3.1 Identification of Entry Points and Exit Points

An entry point of a system is a method that receives data items 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 an entry point if m invokes the `read` method of the `java.io.DataInputStream` class.

An exit point of a system is a method that sends data items 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 an 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 entry points and exit points.

3.2 Estimation of the Damage Potential-Effort Ratio

We estimate a method's damage potential using the method's *sources of input data (destinations of output data)*. 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 whereas the method receives data items from the data store in case of File Existence Check attacks. We do not use method privilege to estimate damage potential because the entire code of the NetWeaver platform runs with the same privilege and hence we can not make any meaningful suggestions to reduce the attack surface.

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.

We assign numeric values to sources of inputs and access rights levels to compute numeric damage potential-effort ratios. The choice of the numbers depends on a system and its environment. We discuss a specific way of assigning numeric values in case of the service in Section 4.2.

4 Implementation of a Measurement Tool

In this section, we describe a tool we implemented to measure the attack surfaces of software systems implemented in `Java`. We implemented our tool as a *plugin* for the *Eclipse* Integrated Development Environment (IDE) so that software developers can use the tool inside their software development environment [5]. We show a screen shot of our tool in Figure 2.

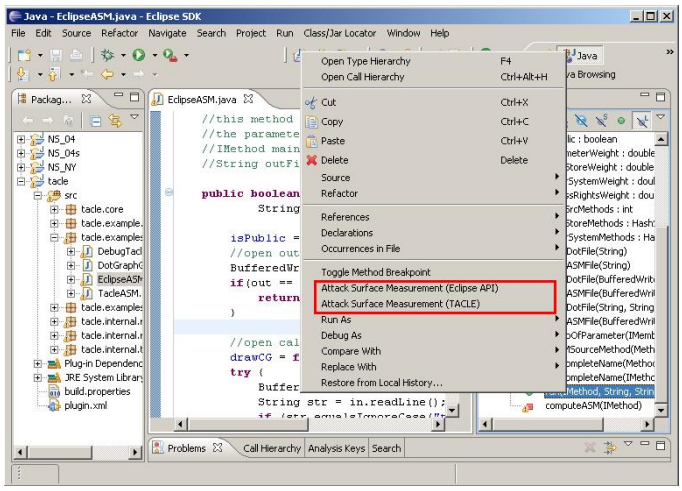

Fig. 2. Screenshot of the Attack Surface Measurement tool implemented as an Eclipse plugin.

4.1 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, which gives a very precise call graph, but does not scale well to large programs [6]; and an Eclipse API, which gives a less precise call graph, but scales [7].

The TACLE based approach results in a precise attack surface measurement whereas the Eclipse API based approach results in an over-approximation of the measurement. The two approaches are complementary; software developers can use the Eclipse API based approach to produce an imprecise measurement of a large software system, and then use the TACLE based approach to obtain precise measurements of the components that make large contributions to the measurement. The imprecise measurement guides the developers to the relevant components of a system and the precise measurement guides the developers in reducing the attack surface.

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 we provide our tool the list of interface methods of other systems and the list of `Java I/O` library methods through two configuration files. Our approach of identifying entry points and exit points is applicable to generic `Java` systems and is independent of SAP software systems.

4.2 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. 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. Users of our tool would choose the numbers based on the knowledge of their system and its environment.

The methods of the service have three different sources of input: `parameter`, `data store`, and `other system`. As discussed in Section 3.2, 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 an internal 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 [8]. 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	Value
`other system`	1	1
`data store`	3	18
`parameter`	5	35

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

Access Rights	Value
`public`	1
`internal`	18

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

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.

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 $73/18 = 4.05$.

4.3 Usage of the Tool and Measurements

Software developers can use our tool to measure and reduce a system's attack surface. The tool generates detailed output 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. 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. We are currently working on a visualization tool to guide the developers to the relevant parts of the code base.

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.

5 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 versions 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.

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. We estimated

the damage potential-effort ratio of each entry point (exit point) to compute the attack surface measurements; we show the measurements in Table 4.

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

Table 3. The number of entry points and exit points.

Version	Attack Surface Measurement
S3	5298.44
S2	4687.00
S1	4649.00

Table 4. Attack surface measurements.

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 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.

The measurement results show that addition of new features will increase a software system's attack surface measurement. Software developers should, however, aim to minimize the increase in the attack surface. SAP's developers made a good design decision by introducing internal interfaces that reduced the increase in the attack surface measurement.

6 Potential Usage of Attack Surface Measurements

We envision three potential uses of attack surface measurements in the software development process. First, software developers and architects can use attack surface measurements to prioritize their software testing effort. For example, if a system's attack surface measurement is high, they should invest more in testing efforts to identify and remove vulnerabilities from the system. Similarly, if the measurement is low, they can reduce their testing effort.

Second, software developers can use attack surface measurements as a guide while implementing patches of security vulnerabilities. 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.

Third, software consumers can use attack surface measurements to guide their choice of software configuration. Choosing a suitable configuration, especially for complex enterprise-scale software, is a nontrivial task. Since a system's attack surface measurement is dependent on the system's configuration, software consumers would choose a configuration that results in a smaller attack surface exposure.

7 Summary and Future Work

In summary, we introduced a method to measure the attack surfaces of SAP software systems implemented in `Java` and implemented a tool to measure the attack surface in an automated manner. We 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 also learned important lessons on how to improve the method and the tool to make them more useful in practice.

Based on the feedback received from SAP developers, we identify four possible avenues of future work. First, we plan to extend the tool to measure the attack surfaces of software implemented in other languages such as `JavaScript`. Second, Manadhata and Wing performed three empirical studies to validate the abstract measurement method and the measurement results of systems implemented in `C` [8]. A possible direction of future research is to explore validation ideas in the context of SAP business applications. Third, we plan to extend our work by developing a method to estimate the minimum and the maximum possible attack surface measurements of a system given the system's functionality. The minimum and maximum estimates will be useful in guiding attack surface reduction and test effort prioritization. Fourth, we plan to investigate code analysis techniques to identify a system's indirect entry points and indirect exit points in an automated manner.

References

1. Howard, M.: Fending off future attacks by reducing attack surface. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure02132003.asp` (2003)
2. Howard, M., Pincus, J., Wing, J.: Measuring relative attack surfaces. In: Proc. of Workshop on Advanced Developments in Software and Systems Security. (2003)
3. Manadhata, P.K., Kaynar, D.K., Wing, J.M.: A formal model for a system's attack surface. Technical Report CMU-CS-07-144, CMU (July 2007)
4. AG, S.: NetWeaver. `http://www.sap.com/platform/netweaver/index.epx`
5. Eclipse: Eclipse - An open development platform. `http://www.eclipse.org/`
6. Sharp, M., Sawin, J., Rountev, A.: Building a whole-program type analysis in Eclipse. In: Eclipse Technology Exchange Workshop at OOPSLA. (2005) 6–10
7. Eclipse: Eclipse package org.eclipse.jdt.internal.corext.callhierarchy. `http://mobius.inria.fr/eclipse-doc/org/eclipse/jdt/internal/corext/callhierarchy/package-summary.html`
8. Manadhata, P.K., Tan, K.M., Maxion, R.A., Wing, J.M.: An approach to measuring a system's attack surface. Technical Report CMU-CS-07-146, CMU (2007)