Dynamic Taint Propagation for Java

Vivek Haldar
Deepak Chandra
Michael Franz

University of California
Irvine, CA 92697
+1-949-824-7308
{vhaldar,dchandra,franz}@uci.edu

ABSTRACT

Improperly validated user input is the underlying root cause for a wide variety of attacks on web-based applications. Static approaches for detecting this problem help at the time of development, but require source code and report a number of false positives. Hence, they are of little use for securing fully deployed and rapidly evolving applications. We propose a dynamic solution that tags and tracks user input at runtime and prevents its improper use to maliciously affect the execution of the program. Our implementation can be transparently applied to Java classfiles, and does not require source code. Benchmarks show that the overhead of this runtime enforcement is negligible and can prevent a number of attacks.

1. MOTIVATION

"The impact of using unvalidated input should not be underestimated. A huge number of attacks would become difficult or impossible if developers would simply validate input before using it. Unless a web application has a strong centralized mechanism for validating all input… vulnerabilities based on malicious input are very likely to exist."

- The Ten Most Critical Web Application Security Vulnerabilities, 2004, Open Web Application Security Project.

In the "old internet", machines and services communicated with each other using a variety of protocols that were processed largely by programs written in C. The full range of common UNIX remote services falls in this category – mail servers, finger daemons, scheduled job execution services etc. The most common way to attack these services was to exploit buffer-overrun vulnerabilities that stemmed from the fundamental lack of memory safety in the underlying implementation language, C.

The trend now is towards a model of web-based applications that communicate using the HTTP protocol, that are implemented in a type- and memory-safe language such as Java, and executed in a safe runtime such as the Java Virtual Machine or the .NET Common Language Runtime.

Such code platforms offer several advantages over native code. The virtual machine performs a number of static and dynamic checks to ensure a basic level of code safety—type-safety, and control flow safety. Type safety ensures that operators and functions are applied only to operands and arguments of the correct types. A special case of type safety is memory safety, which prevents reading and writing to illegal memory locations—for example, beyond the bounds of an array—and thereby also provides separation between different processes without the need for hardware-based memory management. Control flow safety prevents arbitrary jumps in code (say, into the middle of a procedure, or to an unauthorized routine). These basic properties of safe code are enforced by a combination of static (e.g. bytecode verification) and dynamic (e.g. array bounds checks) techniques. Thus, safe code does away with a major source of errors and vulnerabilities in current systems that stem from unsafe memory operations in C—such as buffer overruns and format string attacks.

Despite the fact that the safe execution environments in which web-applications typically execute are not vulnerable to buffer-overrun attacks, a wide variety of new attacks specifically targeting them have recently surfaced [3]. Instead of exploiting the weak-typing of the underlying language, attacks now focus on exploiting the presence of *logic errors* in the application. Since the interface web-applications provide to the world is simply an HTML page, they can be attacked from any client capable of issuing HTTP requests, and very often the only tool needed is a browser.

One large class of such errors is using untrusted user input in security-sensitive commands without proper validation and sanitization. An overly simplistic example of this is using a user-input string as argument to the `System.exec()` call in Java. If this string is not properly checked, it allows the user to execute arbitrary commands on the hosting system. User input consists not just of data entered into HTML forms, but the full range of data that originates from untrusted sources external to the web-application. This includes sources such as data read from cookies on the client and HTTP parameters encoded in a URL. Identifying, tracking and preventing the improper use of such untrusted data is the domain of the *taint problem.*

Various approaches have been explored to attack the taint problem (see section 6 for an overview). Broadly, these fall into two categories – *statically* analyzing code for the presence of taint vulnerabilities, and *dynamic approaches* that track tainted data at runtime. Each has its own advantages and disadvantages, and is applicable in different scenarios.

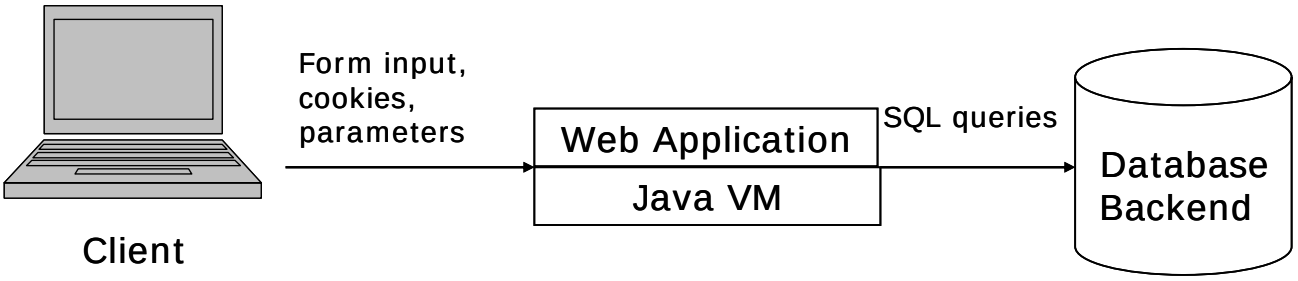

Figure 1: Architecture of a web application

Static analysis is useful at the time of application development, when potential vulnerabilities found by the analysis can be fixed by the programmer in source code. Some human intervention is also needed because static approaches, in order to be conservative, typically also report a number of false positives. The programmer must then manually examine the reported errors to determine which are actual vulnerabilities and which are not.

There are two problems that need to be dealt with. Firstly, the problem must be *specified* correctly. This means getting all the rules and corner cases for validating user input right. Secondly, this specification must be *implemented* faithfully. Static approaches can catch implementation errors, but not bugs of specification. If a dynamic approach independently also performs its own checks, it may be able to catch more errors than only static checking.

However, static approaches do provide more accurate reports than runtime approaches, enable fixing vulnerabilities *before* an application is deployed, and have no runtime performance overhead.

But most web-applications deployed in the real world do have bugs in them. A study [5] estimates that nearly 60% of deployed applications are vulnerable. For the large majority of these applications, the source code is not available. Moreover, web-applications also rapidly change and evolve. Here, static approaches fall short.

A dynamic, runtime technique that can be *transparently* applied to deployed applications is very useful in such scenarios. This explains the popularity of Perl's taint mode [4]. It is not guaranteed to prevent attacks, but it significantly raises the bar for exploiting taint vulnerabilities in Perl CGI scripts.

In this paper, we present a technique and our implementation for dynamically tracing tainted user input in the Java Virtual Machine. Our technique tracks the taintedness of untrusted input throughout the lifetime of the application. Taintedness is propagated in the obvious way – strings derived from tainted strings are also considered tainted. Our technique is completely transparent – the application is completely unaware of it. It can be applied to an existing Java classfile, and does not need source code.

We allow the separate specification of *sources* of tainted data, as well as sensitive methods that should not use tainted data (also called *sinks*). This separation of mechanism and policy gives our technique great flexibility. We need specify these sources and sinks only once per library. For example, once we specify the sources and sinks in the J2EE library, all applications using that can benefit from dynamic taint propagation. Sources are usually methods that get input from outside the program, and sinks are usually methods that either write output outside the program, or execute some form of code (SQL, shell commands). We track taintedness from sources to sinks, and prevent tainted data from being passed into sinks.

Our technique uses a fairly simple policy to *untaint* tainted data. This is needed because otherwise *all* data that depends on user input would always be considered tainted. Note that our policy for untainting data is a heuristic, and trusts that the programmer performed meaningful validation checks.

The rest of this paper is organized as follows: Section 2 provides an overview of the taint problem, and the various attacks that can be mounted against web-applications because of improperly validated input; Section 3 explains how we dynamically trace taintedness in the Java Virtual Machine; Section 4 presents implementation details and the results of some benchmarks; Section 5 discusses avenues for future work; Section 6 gives an overview of other approaches for dealing with the taint problem; and Section 7 concludes.

2. THE TAINT PROBLEM

The taint problem in web applications stems from using improperly validated user input in commands that are security-sensitive. This is the underlying cause for a wide variety of attacks on web-applications. Many authors [1, 2, 3] have given excellent overviews of attacks on web-applications, and in particular, how improperly validated user input can be used to mount these attacks. We borrow heavily from them and provide a short overview of these attacks here.

Figure 1 shows the architecture of a typical web-based application. It presents an HTML interface to users, and having got some input from them, queries a database backend, formats the result and presents a new HTML page. The backend need not always be a database, but could also be any other data source, such as another web application.

An attacker's goal is to manipulate user input such that it can be used to affect the execution of the program maliciously. For example, an attacker could provide input that is then used to construct malicious queries to the backend to extract data that she was not authorized to see. Another goal might be to *insert* information into the database to pollute it, or plant misinformation in it.

2.1 EXAMPLES OF ATTACKS

We illustrate with an example from WebGoat [13], a collection of web applications designed to demonstrate attacks on them. Consider a web form with a textbox where the user fills in her account number, and after pressing "OK", the resulting page displays her credit card information. The information is looked up in the database using the following query:

```
SELECT * FROM user_data WHERE userid =
<string input by user in textbox>
```

Here the string used to construct the SQL query is not properly checked before being sent to the database backend and a malicious input string can easily leak sensitive data. For example, if the user inputs:

```
101 OR 1
```

Then the resulting SQL query becomes:

```
SELECT * FROM user_data WHERE userid =
101 OR 1
```

In this query, the boolean condition evaluates to "true" always because of the additional "OR 1". Thus the query will match *all* records, and the resulting HTML page will display all credit cards in the database. Such attacks, where user input is used to affect the execution of a command on the local host, are called *command injection* attacks.

For another attack, consider a web forum with a text box where users enter new messages. A user could enter arbitrary JavaScript content between <SCRIPT> and </SCRIPT> tags in this text box, and the message would then be part of the webpage. Other users who load the same page would now be unknowingly executing this inserted JavaScript. This is an example of a *cross-site scripting* attack.

2.2 CLASSES OF ATTACKS

Attacks on web-applications can target both the hosting server, as well as clients that access the application. Some of the most prevalent attacks on web-based applications are:

- ***Command injection attacks:*** user input is manipulated to insert a maliciously constructed executable command into the program. The most common case of this attack, SQL injection, happens when user input is used in some way to construct an SQL query for the database backend. If this input is not properly validated, it could be used to construct a malicious SQL query.

- ***Cross-site scripting*** (also called output attacks) [11]: a maliciously crafted URL can insert executable scriptable content into a dynamically generated webpage. Thus, a user may unknowingly execute scripts when she visits a URL given to her. This script could leak local data, or redirect information to a malicious server rather than the original host of the webpage. Typically, such malicious URLs are found in spam emails. When clicked, the malicious script is executed. The underlying problem is that the URL, which is also a form of untrusted user input, is not properly validated. The earlier example of a malicious message on a web forum also falls under this category.

- ***Hidden Field Tampering:*** websites often use hidden fields to communicate persistent session data such as user ID, pricing information etc. The problem is that very often the value of these hidden fields is not properly validated at the server end. If these fields are tampered with, they could be used for malicious purposes, such as buying items for a price other than that published, or forging identities.

- ***Cookie Poisoning:*** malicious data is inserted into cookies that are used by the web-application. For example, often a website will skip authentication based on data stored in a cookie. If the cookie is modified, it could be used to present a forged identity to a website.

Of the above attacks, command injection, field tampering and cookie poisoning are attacks on the hosting server. Cross site scripting, on the other hand, targets clients that use web-applications. Note that though all these attacks use different avenues of attack, the root cause of all of them is improperly validated user input.

3. DYNAMICALLY TRACKING TAINTEDNESS

In order to track tainted user input, we need to specify the following:

- ***Sources:*** A *source* is a method that returns user input. Usually these are methods that get HTML form input, or read cookies stored on the client, or parse HTTP parameters. All strings emanating from sources must be marked tainted.

- ***Propagation:*** Strings from sources are usually manipulated to form other strings such as queries, or scripts, or filesystem paths. Strings that are derived from tainted strings also need to be marked tainted.

- ***Sinks:*** A *sink* is a method that consumes input or derivative of user input. This includes methods that execute some form of code (such as a script or SQL query), or methods that output data (such presenting a new HTML page). Tainted strings must be prevented from being used as parameters to sinks.

Sources and sinks need to be specified once per library or framework that a web application uses[1]. For our benchmarks, we needed to specify sources and sinks for the J2EE library.

To track the taintedness of strings, we associated a *taint flag* with every string. This taint flag is set when a string is returned by a source method. We propagate this taint flag to strings that are derived from tainted strings through operations such as concatenation, case conversion etc.

[1] We adopt the terms "source" and "sink" from [1].

3.1 UNTAINTING

Once we have a mechanism to mark strings tainted, we also need a way to untaint strings. This is needed because in the absence of a way to untaint strings, all strings that are derived from tainted strings will still be marked tainted. This includes strings that have been put through a sanitizing procedure and should not be marked tainted anymore.

The problem is to determine which procedures are sanitizing procedures. Since our technique applies transparently to existing Java bytecode, we have no programmer input telling us which methods sanitize and validate user input. Thus, we have to use a heuristic to determine this. Choosing this heuristic is one of our major design decisions.

We assume that methods of `java.lang.String` that perform checking and matching operations are used to untaint strings. For example, a tainted string that is passed through a regular expression match, or been tested for the presence of a particular character is not tainted anymore. Note that here we trust the programmer to have performed a meaningful check that accounts for all cases that might be exploitable in an attack. It is entirely possible that the programmer wrote a faulty input-validation routine that lets through user-input strings with malicious content in them.

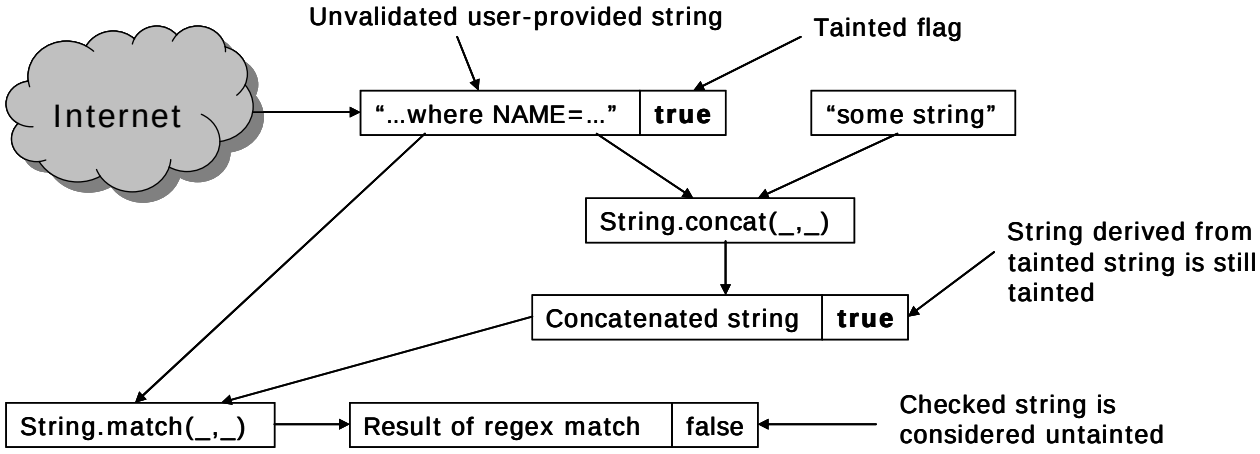

Figure 2: Overview of tainting and untainting

3.2 DEALING WITH TAINT ERRORS

A *taint error* occurs when a tainted string gets used as an argument for a sink method. When this happens, we could take one of a number of actions:

- *Raise a Java exception* indicating a runtime taint error: Since this is an exception the application is unaware of, this particular exception will not be caught, but if the application has a mechanism to deal with unknown runtime exceptions, it may be able to recover. In any case, tainted data will not be allowed into a sink.
- *Abandon the particular session* that caused a taint error.
- The weakest option is to let tainted data be used as an argument to a sink, but *make a full log* of the arguments, the sink, and the path the tainted data took from source to sink. This seems insecure, but is useful when auditing, doing penetration testing, debugging, or if used in a honeypot.

4. IMPLEMENTATION AND RESULTS

We have implemented our taint propagation scheme for the Java Virtual Machine, and tested it on a number of applications. Our implementation is independent of the particular JVM being used. We use bytecode instrumentation, and use Javassist [6] for this.

Our implementation needs to do the following:

- Specify sources and sinks.
- Mark strings emanating from sources as tainted.
- Propagate taintedness of strings.
- Mark strings untainted according to our heuristic.
- Raise an exception when a tainted string is used as an argument to a sink method.

The way we specify sources and sinks is straightforward. We simply list out every source method (say `Form.getValue()`) in a text file, one per line. We do the same for sink methods.

We instrument the `java.lang.String` class to propagate taintedness information, as well as untaint strings. Some methods are instrumented to propagate taintedness of strings, whereas some others make strings untainted. This instrumentation is done once off-line. This is because the JVM prohibits the load-time modification of system classes such as `java.lang.String`. System classes must be loaded by the primordial system class loader, while load-time instrumentation requires the installation of a custom class-loader.

We instrument the `java.lang.String` class as follows:

- Add a boolean field to the class that indicates whether it is tainted or not
- Instrument all methods in the class that have some String parameters and return a String, so that the return value is tainted if at least one of the parameters is tainted.
- The above is done for all but a number of string checking and matching methods, which *untaint* data. For example, `foo.match(regex)` will untaint `foo`.

Strings are immutable in Java. The java compiler compiles string operations such as concatenation into operations on the StringBuffer class, which implements mutable strings. For example, the expression

```
string1 + string2
```

will actually be compiled to

```
(new StringBuffer(string1)).append(string2)
.toString()
```

Because of this inter-conversion between Strings and StringBuffers, we also instrument the `java.lang.StringBuffer` class in much the same way as the `java.lang.String` class, by adding a tainted flag, and modifying its methods to propagate taintedness.

The StringBuilder class is also used internally to manipulate strings. It is like the StringBuffer class, except its methods are not thread-safe. We instrument the StringBuffer class too.

All other classes are instrumented at load-time using a custom class loader, as follows:

- *If the method is a source:* we mark the returned string tainted.
- *If the method is a sink*: we check if any of its arguments is a tainted string. If so, we raise an exception indicating a taint error.

Note that we only instrument classes that have sources or sinks in them, and not all classes. Currently, due an incompatibility between the class loader hierarchies of Javassist and Tomcat (the servlet container that executes our benchmark web applications), we are unable perform this instrumentation at the time of class loading. Instead, we instrument these classes offline.

We wrote a micro benchmark to measure the overhead of instrumenting the `java.lang.String` class to handle tainting information. The benchmark consisted of a number of string operations repeated in a loop, and was run with strings of length varying from 1 to 10000. It was run on a PentiumM 1.5 GhZ laptop with 512 MB of RAM, running Windows XP SP2, using version 1.5 of the Java runtime. Our measurement showed no noticeable difference in execution time of the benchmark between using the original and instrumented String class.

To test our taint propagation framework, we ran it with the WebGoat [13] set of web applications. WebGoat is a collection of applications designed to teach secure programming for web applications, and has a range of vulnerabilities in it by design. One application demonstrates a command injection attack, where user-supplied command can be executed on the host by tampering with HTTP parameters. Another demonstrates an SQL injection attack, where supplying a malicious string in an HTML form results in a query being executed on the host that reveals secret data.

We specified a list of sources and sinks specific to the J2EE framework, and ran WebGoat under our taint propagation framework. Our implementation flagged a taint error for both the applications mentioned above, and prevented the attack from being successfully carried out.

5. DISCUSSION AND FUTURE WORK

This work grew out of our broader attempt to bring strong mandatory access controls (MAC) to the Java Virtual Machine [12]. Our objective in that work was to explore how MAC can be integrated into a JVM, and at what granularity it is meaningful to do so, with the aim of providing greater assurance for applications that require strong data partitions, and that need to track the permissions and ownership of data throughout the lifetime of the program. Current access control mechanisms in Java can only control initial access to a resource, but fail to track data throughout execution, or limit how they are used once access was granted. We implemented a prototype JVM that performed MAC at the granularity of objects. Every object had a MAC tag associated with it. Based on the policy in place, this tag regulated how and if other objects were allowed to access it. Taint propagation can be seen as a special case of using MAC in the JVM. Taint tags associated with strings are in effect a kind of access control tag.

There are a number of avenues for future work:

Currently we have only tested our implementation with the WebGoat [13] sample applications. This is not a very realistic benchmark, as it was designed to demonstrate how web applications can be attacked, and has vulnerabilities by design. We are currently in the process of finding other realistic web applications, and would like to test our taint propagation framework with them.

Another direction for future work is to use our tool for logging of attacks and penetration testing. For this, it would be useful to have additional information carried along with tainted strings, such as which source method it came from, and what path (in terms of method calls) it followed from source to sink.

We would also like to explore a declarative approach to specifying valid inputs. Valid inputs for the large majority of web applications follow well-known rules, such as an expected format and the absence of certain special characters that could be used in an attack. In spite of this, every application developer rewrites these from scratch for a given application, often leaving holes and bugs. If these validation rules could be attached to sources and sinks and executed at runtime, they would form an additional layer of security, independent of and in addition to the checks the application already has. We do not expect this additional checking to impose a significant performance overhead as most web applications are I/O bound, and CPU time is usually not a bottleneck.

Extending this approach even further, we could attach to sources and sinks an operation that established an *invariant*. This may require source code modification, but only of the library, not

the application. It may even be possible to do this transparently at the bytecode level. The application will still be unaware of this, and not need to be modified.

Currently we have only two levels of tainting associated with a string – it is either tainted or not. However, a large web application deals with a number of data sources other than just users, such as other web applications, off-site databases etc. Input from these sources may not be untrusted to the same extent as input from a remote user on a client. Extending our work on MAC at the object level, we would like to explore if having a finer granularity of taint levels can improve the security of web applications. With multiple taint levels, we could also enforce policies and invariants about how and when data from various taint levels are allowed to mix, and what level of tainting the resulting data is marked with. This might be particularly useful in light of recent regulations [14] that mandate how information from various departments within an organization, and among organizations, is allowed to mix.

6. RELATED WORK

The original inspiration for this work is Perl's *taint mode* [4]. When in taint mode, the Perl runtime explicitly marks data originating from outside a program as *tainted*. This includes user input, input from environment variables and and file input. Tainted data is then prevented from being used as arguments for certain sensitive functions that affect the local system – such as running local commands, creating and writing files and sending data over the network. Doing so results in a runtime exception and termination of the program. Perl also provides a mechanism to *untaint* tainted data. Results of a regular expression match are always considered clean. Hence, if a tainted string is matched against a regular expression, the resulting match is clean. The programmer is trusted to have adquately checked a tainted string if she wrote a regular expression to filter it. Thus, taint mode is not a 100% guarantee for catching taint bugs. Its goal is to catch unintentional programmer errors, such as passing a user-input string directly to a shell command.

Ruby [7] has finer-grained taint levels than Perl. It has safe levels ranging from 0 to 4, each successively more stringent. Level 0 has no checks on tainted data, whereas level 4 partitions program execution into two sandboxes, one with tainted objects, and one without. Tainting is done at the level of *objects*, not just strings. Any object that had tainted data in it at any point during execution is marked tainted.

Our work essentially brings the idea of taint propagation to the Java runtime. The important difference is that our approach is more flexible and extensible because the list of sources and sinks is not hard-coded into the runtime, but separately specified. This allows our mechanism to be used for taint checking applications that use various libraries, after having specified sources and sinks for each library once. Moreover, we can run different instances of the same application, each with different source and sink specifications.

Nguyen-Tuong et al [2] have implemented taint propagation for the PHP interpreter. PHP is a widely used web scripting language. Their technique mostly mirrors Perl's. However, their technique for sanitizing data is different. Rather than have an operation that untaints strings, they never untaint strings, and put strings through their own sanitizing functions before they are passed as arguments to sensitive functions. Once again, the list of these sensitive functions is not separately specified, but built into the PHP interpreter.

A great deal of work has been done on static approaches to analyzing code security [8], and the taint problem in particular [1, 9, 10].

Taint propagation is an information flow problem[17]. Static checking approaches such as Myer's JFlow system [16] type-check source code for secure information flow. However, the programmer needs to insert source code annotations explicitly labeling sensitive data.

The WebSSARI [15] project analyzes information flow in PHP applications statically. It inserts runtime guards in potentially insecure regions of code. It differs from approaches such as JFlow in that it does not require source annotations.

Static analysis has also been applied to C programs [9, 10]. Evans' Split static analyzer [10] takes as input C source code annotated with "tainted" and "untainted" annotations. This is accompanied by rules for how objects can be converted from one to the other, and which functions expect which kinds of arguments. Shankar et al [9] use a similar approach in which C source code is annotated, but they use type-qualifiers instead.

The major disadvantage of all these approaches is that they require source code, and while useful at the time of development (even though they might report a number of false positives requiring manual examination to clear), they cannot be applied transparently to already deployed applications that are only available as binaries.

7. CONCLUSIONS

The most prevalent attacks on web applications – command injection, parameter tampering, cookie poisoning, cross-site scripting – all have the same root cause: improperly validated user input. Static approaches for detecting the presence of these vulnerabilities require the presence of source code. But this is unrealistic for deployed applications that still have bugs in them.

In this paper, we have proposed a framework for tagging, tracking and detecting the improper use of improperly validated user input (also called *tainted* input) in web applications. We mark data originating from the client as tainted, and this attribute is propagated throughout the execution of the program. Data derived from tainted data is also marked tainted. Finally, we prevent tainted data from being used improperly in security-sensitive contexts.

Our implementation runs on the Java Virtual Machine, and is able to prevent the improper use of tainted data. We associate a *tainted flag* with strings. Data originating from methods that get user input, called *sources*, is marked tainted. Strings derived from tainted strings are also marked tainted. Certain string checking operations mark data untainted. Here we trust the programmer to have made a meaningful check. Finally, methods that consume input or execute some form of code (scripts, SQL), called *sinks*, are prevented from taking in tainted arguments.

Our technique applies to Java classfiles and does not require source code. Hence it can be transparently applied to deployed web applications and increase their security in the face of attacks.

8. ACKNOWLEDGEMENTS

This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-05-2-0216. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

9. REFERENCES

[1] V. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In USENIX Technology Symposium, 2005.

[2] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Green, Jeffrey Shirley, David Evans. Automatically Hardening Web Applications using Precise Tainting. In IFIP Security Conference, May 2005.

[3] Open Web Application Security Project. Top Ten Most Critical Web Application Security Vulnerabilities. January 2004. http://www.owasp.org/documentation/topten.html

[4] Larry Wall, Tom Christiansen, Jon Orwant. Programming Perl, 3rd ed. O'Reilly.

[5] Moran Surf and Amichai Shulman. How safe is it out there? Imperva. June 2004. http://www.imperva.com/application_defense_center/papers/how_safe_is_it.html

[6] Shigeru Chiba. Javassist: Java Bytecode Engineering Made Simple. Java Developer's Journal, vol. 9, issue 1, January 8, 2004

[7] Dave Thomas, Chad Fowler and Andy Hunt. Programming Ruby: The Pragmatic Programmer's Guide, 2nd ed.

[8] B. Chess and G. McGraw. Static Analysis for Security. IEEE Security and Privacy, 2(6), 2004.

[9] U. Shankar, K. Talwar, J. S. Foster and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. USENIX Security Symposium. 2001.

[10] D. Evans and D. Larochelle. Improving Security using Extensible Lightweight Static Analysis. IEEE Software. Jan/Feb 2002.

[11] CERT Advisory CA-2000-02. Malicious HTML tags embedded in Client Web Requests. February 2000.

[12] V. Haldar, D. Chandra and M. Franz. Practical, Dynamic Information Flow for Virtual Machines. Technical Report 05-02, Departmnet of Information and Computer Science, University of California, Irvine. February 2005.

[13] Open Web Application Security Project. The WebGoat Project. http://www.owasp.org/software/webgoat.html

[14] K. Beaver. Achieving Sarbanes-Oxley Compliance for Web Applications through security testing. http://www.spidynamics.com/support/whitepapers/WI_SOXwhitepaper.pdf

[15] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. Proceedings of the Thirteenth International World Wide Web Conference (WWW2004). May 2004.

[16] A. C. Myers. JFlow: Practical mostly-static information flow control. In Symposium on Principles of Programming Languages, pages 228–241, 1999.

[17] A. Sabelfeld and A. Myers. Language-based information-flow security. 21(1), 2003.