Practical Dynamic Taint Analysis for Countering Input Validation Attacks on Web Applications

Wei Xu, Sandeep Bhatkar, and R. Sekar
Department of Computer Sciences
Stony Brook University, Stony Brook, NY 11794-4400
{weixu,sbhatkar,sekar}@cs.sunysb.edu

Abstract

Errors in validation of user inputs lead to serious security vulnerabilities. Many web applications contain such errors, making them vulnerable to remotely exploitable input validation attacks such as SQL injection, and cross site scripting. In this paper, we present a dynamic taint analysis technique to detect the input validation attacks. More specifically, our technique is based on tracking flow of taint information from untrusted input into the parts of the generated output (or commands). A unique benefit of our approach is that it can be applied to all of the web application development languages whose interpreters are implemented in C. We demonstrate this ability by applying our technique on web applications which use PHP, and bash scripts. Our technique is implemented as a fully automatic source-to-source transformation. We present experimental evaluation to establish effectiveness of our approach, paying particular attention to its attack detection ability. Experiments demonstrate that our technique detects all the attacks accurately with no false alarms.

Key Words: Input Validation Errors, Fine-grained Information Flow Tracking

1 Introduction

Web applications have been playing a critical role in many areas such as financial transactions, commercial business, cyber community services. As a consequence, web applications also become interesting targets for attackers. Most research efforts have been put on detecting and preventing buffer overflow attacks, however, there are less research work on attacks that exploit input validation vulnerabilities, which have been shown as a more significant problem in web applications. In fact, according to [15], input validation attacks are the No. 1 type of web attacks.

Web applications usually take inputs from users through form fields, cookies, or some other standard channels, and use these input data in further processing operations, such as querying databases, generating web pages, or executing commands. Because the input data are from the remote users and may contain malicious values, they need to be validated before use. Once a web application fails to do so, attacks can exploit the vulnerabilities to launch particular attacks. Examples of popular input validation attacks are SQL injections, cross site scripting, and command injections. These attacks can cause many serious problems, such as leak of sensitive information and corruption of critical data.

One way to detect the above types of attacks is to properly sanitize user input, for example, to ensure that no special characters or SQL keywords are used in untrusted input. This can work, but it requires the developers to perform the checks. As illustrated by the buffer overflow problems, developers often do not have the time, know-how or willingness to check their code for all security vulnerabilities. Even if they did introduce security checks in their code, these checks may be incomplete, still leaving the door open for attackers to sneak through. Consequently, it is important to develop techniques that can largely automate the process.

Several static analysis approaches have been proposed to address this problem. These approaches use type analysis or pointer analysis on the source code to statically examine whether a user input will be used in security sensitive operations along some execution paths in a program. Any such instance will be reported as a possible vulnerability. Because static analysis always need to make approximations, the analysis results are usually not very accurate and false alarms can be very high. Furthermore, it is legitimate for web applications to use user inputs in even security sensitive operations as long as these inputs are properly sanitized. It just makes the analysis results even less welcome by reporting every link from a user input to a security operation as a vulnerability.

Recently dynamic information flow tracking techniques have been proposed to detect control flow hijacking attacks such as buffer overflow attacks and format string attacks [18, 14]. In this paper, we present a runtime information flow tracking technique to detect input validation attacks in web applications.

We have made the following contributions with our approach:

- We present a practical runtime technique which can accurately detect input validation errors in web applications. This technique can be used as the second line of defense to account for programmer's negligence. In addition, our automatic technique can also address the unknown input validation vulnerabilities.
- We present an information flow tracking technique that is much more comprehensive than previous similar approaches. Indirect flows such as assignments through control flows, or assignments through translation tables are taken care of in our approach. We also carefully handle external functions.
- Our technique can be applied to various scripting languages that are used in web application developments. We achieve this by providing a transformation tool that works programs written in C, and using this tool to transform interpreter programs for the scripting languages.
- Our approach support versatile policy enforcement. Because information flow tracking results are available at the time of policy enforcement, the checking functions can make use of these tracking results, and provide much more accurate and reasonable checking results.
- We have implemented a prototype tool, and evaluated our technique on two scripting languages and a number of web applications. The results show that our technique has accurate attack detection and reasonable performance overheads thankful to our optimization techniques.

In the rest of the paper, we first give a background description on input validation attacks in Section 2. Then we give an overview of our approach in Section 3, and present transformations in Section 4 and optimizations in Section 5. In Section 6 we provide our implementation and experiment results. After that, we discuss some of related work in Section 7 and conclude the paper in Section 8.

2 Background: Input Validation Vulnerabilities

A web application is typically a client/server software that handles user requests coming from clients such as web browsers. To serve the user requests, it often requires accessing system resources such as databases and files at the server end. Figure 1 shows a simplified architecture of a web application. The system resources are a part of trusted environment and often contain security-critical data. Hence the resources need appropriate protection to maintain their confidentiality and integrity. User input, however, cannot be trusted. It cannot be directly used to access the system

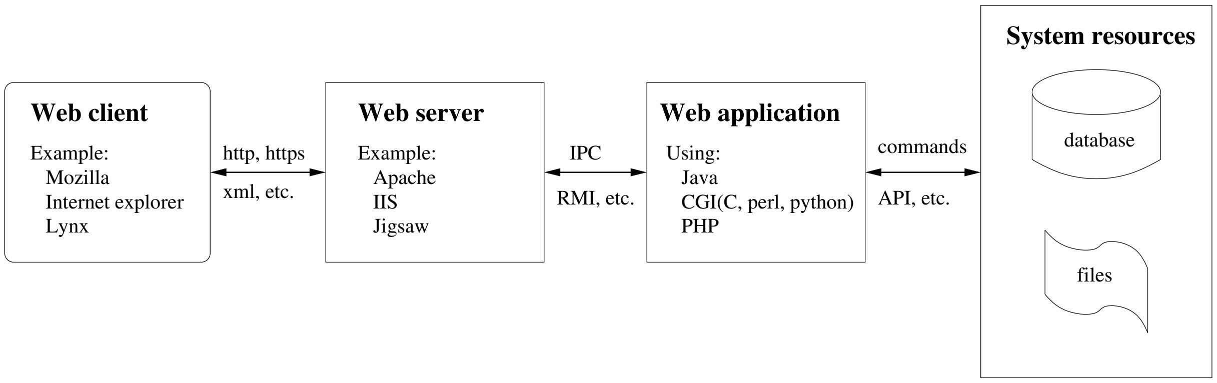

Figure 1: Web application architecture

resources. Checks should be performed to validate the user input before it can be used in the trusted environment. Lack of validity checks or inadequate checks can result in exploitable vulnerabilities. In this section, we present only a few examples of such vulnerabilities. Comprehensive information on the vulnerabilities can be obtained from [7, 17, 15].

2.1 SQL Injection

SQL Injection vulnerabilities occur when untrusted user input is used to construct SQL commands. Malicious input can alter the semantic meaning of an SQL command, which on execution performs actions different from the one intended. Consider the following statement in PHP application used for constructing a SQL command:

```
$cmd = "SELECT COUNT(*) FROM users
        WHERE name = '" . $name . "'
        AND passwd = '" . $pwd . "'";
```

The variables `$name` and `$pwd` respectively hold user name and password provided by the user. The SQL command authenticates a user by returning 1 for a valid user name and password combination, otherwise it returns 0. An attacker can enter the same value: `' 1=1` for both `$name` and `$pwd`. The resulting SQL command would be:

```
SELECT COUNT(*) from users
WHERE name = '' OR 1=1
AND passwd = '' OR 1=1
```

The above statement always returns 1, thereby authenticating the attacker. The attacker can also use `' OR 1=1 --` for the user name. The two hyphens (`--`) in Microsoft SQL server is the comment symbol, and therefore the part of SQL query after the symbol is commented out. The attack still succeeds. Worse attack can be launched by using `' ; DROP TABLE users --` as the value for user name. The semicolon symbol (;) divides the command string into two different SQL statements, both of which get executed. As a result of this, the table `users` is dropped. Possibilities of attacks are numerous !

In order to detect SQL injection attacks accurately, first we need to identify parts of the SQL query string which depend on the user input. If any of the part contains non-permissible SQL keywords, or meta-character such as `;` and `--`, the SQL query should be rejected.

2.2 Cross-Site Scripting (XSS)

Many web sites have options that allow users to enter data and then dynamically generate web pages based on the input. A cross-site scripting vulnerability may occur if the user input is not properly validated. Attackers exploit this vulnerability by embedding malicious script into the generated page. The script is automatically executed on the machine of any user who views the generated page.

Consider an example of a web service which allows viewing of remote documents by including the document name in the query string as shown below:

```
http://www.trustedsite.com/searchdoc.cgi?file=designDoc
```

If the requested document does not exist, the web site returns an error message such as:

```
<HTML>
Document does not exist: designDoc
...
</HTML>
```

Note in the above output that the search string `designDoc` is directly included from the URL of query string. This may seem harmless. However, an attacker can construct a XSS attack in the following way. First, he lures the victim into clicking the below URL:

```
http://www.trustedsite.com/searchdoc.cgi?file=<script>
window.open("http://www.attacker.com/malicious_script.js")
</script>
```

A link to the URL can be included in an e-mail to the victim, or it can be included in a web page that the victim visits. Once the victim opens the URL, the error output containing the malicious script is delivered to the victim on behalf of the trusted web server. However, the victim's browser now interprets the output in a different way. Since `<script>...</script>` are the HTML tags for executing scripts, the browser executes the malicious script instead of literally displaying the argument of the query string. The trust level associated with the malicious script will be the same as that with the trusted web server, which means that script can at the minimum read the cookies set by the server. Cookies can be effectively used to steal the identity of the victim. Note that depending on the trust level, the script may be have additional privileges such access to file system, etc.

A cross-site scripting seems to be an easy attack to detect. The web server should just check the input for existence of HTML tags such as `<script>`, and not let them through where it is not allowed. However, checking every input is tedious and error-prone job. Also, this particular type of check can be easily bypassed by using character encodings on the malicious part of the input string.

2.3 Command Injection

Web applications often use functions to manipulate system resources. Examples of functions include `fopen()`, `unlink()`, `system()`, `exec()`, etc. Command injection vulnerabilities occur when the arguments to these functions are constructed from untrusted data.

Recently a command injection vulnerability was discovered in ePing plugin [6]. It contains a PHP script for executing ping command. Vulnerable portion of the PHP code is shown below:

```
$eping_do = $eping_cmd . " " . $eping_count .
            " " . $eping_host;
If (eping_validaddr($eping_host))
{
  ...
  system($eping_do, ..);
}
```

The function `eping_validaddr` checks the validity of the input string corresponding to the IP address of the host being pinged. However, it has a bug due to which even improper IP addresses are validated. Attackers can exploit this bug by injecting malicious command in the input `$eping_host`.

For example:

```
?eping_host=127.0.0.1;cat%20/etc/passwd
```

Thus, on execution of the PHP `system` function, password file is displayed (`%20` is the HTML encoding for the space character).

Strict validation of every input that is used in the arguments of vulnerable functions can prevent command injection types of attacks. For the above example, we can specifically check if the input contains any meta-characters (e.g., `;`) or any other commands.

3 Overview of Approach

All the attacks described in Section 2 are caused by the same type of bugs in web applications: failure to validate untrusted input data prior to using them in security sensitive operations. By studying these attacks, we have made the following important observation: *if special characters (e.g. SQL keywords) are found in those parts of security sensitive operation arguments (e.g. SQL queries) that are directly copied from untrusted input, then it is a very strong indicator of an attack.* Based on this observation, we develop the following approach to detect input validation errors:

- *Identify untrusted input and security-sensitive operations.* Untrusted input may include any input that is read from a remote client by a server. Security-sensitive operations may include database queries, execution of shell commands, invocations of important system calls or library calls such as `system` and `popen`; and so on. Both untrusted input and security-sensitive operations are identified by generic annotations on relevant functions, and this does not require much application-specific knowledge.
- *Use information flow tracking to identify, at a fine level of granularity, how arguments to sensitive operations are derived from untrusted user inputs.* We use runtime tracking of information flows (rather than static analysis) to obtain the degree of accuracy necessary for a preventive approach.
- *Express and enforce security policies that can use the content of sensitive arguments, together with information flow.* A simple example of such a policy is: "only those characters in the query string that weren't derived from untrusted input can be special characters." More complex polices can be defined based on knowledge of security sensitive operations, For example, a policy may parse the query to identify SQL keywords, and then require that no component of any keyword is derived from untrusted input.

The most important and challenging step is to correctly and efficiently track flows of untrusted data at a fine level of granularity. In our approach, we developed a source-to-source transformation technique to instrument programs for tracking flows of untrusted data at the byte level. There are two significant advantages of the source code transformation technique:

- *Efficiency.* Tracking flows of untrusted data in transformed programs can be made efficient, especially when static program analysis is used to eliminate unnecessary tracking statements.
- *Wide applicability.* We notice that although web applications are developed in various scripting languages such as PHP, Perl, Bash, most of these languages are interpretation-based and their interpreters are written in a common language, namely C. By developing a program transformer for the common language and using the transformer to transform the interpreters, our technique can be easily applied to all the languages. Furthermore, the transformation can be made transparent to web applications, and thus make the deployment much smoother.

In addition, source code transformation makes it much easier to track indirect flows. Tracking such flows, as described later, are very important to improve the precision of tracking and attack detection.

4 Transformation

4.1 Basic transformation for tracking untrusted data

Our approach is based on precise "taintedness" tracking on program memory at runtime. A memory unit is marked as *"tainted"* when the stored value is copied from untrusted data. Annotations are provided to mark as "tainted" the untrusted input data from untrusted sources. Programs are transformed to insert additional code for tracking the propagation of "tainted" input data.

4.1.1 Taint `tagmap` We use a memory "mirror" to track the propagation of untrusted input data in programs. Each byte in the program memory is associated with a *taint tag* indicating whether this byte is *"tainted"* or *"untainted"*. The taint state of each byte is represented as a bit in the "mirror", a global bit array `tagmap`, which is indexed by the program memory address of each byte. A bit value of 1 represents "tainted", while 0 represents "untainted". In a 32-bit architecture, a program may allow to use up to 4GB virtual memory. So the array `tagmap` is defined as large as 512MB in virtual memory. However, the actual memory usage of `tagmap` depends on how much actual memory is used by the program and it is usually much smaller than 512MB.

Accessing the `tagmap` involves two steps. The first step is to locate the tag word in the array using the highest bits in an address, while the second step is to locate the tag bit within the tag word using the lowest bits in the address. A number of accessor functions are defined to facilitate reading, writing, and copying tags in `tagmap`. These functions are frequently called in the instrumentation code to update the "taintedness" state of the program memory. These functions are specialized for different sizes of memory blocks, and most of them are implemented as macros to avoid overheads due to function calls.

Initialization of `tagmap`. The taint `tagmap` is initialized as all *"untainted"*. Taint tags for global variables that have initial values are initialized in module constructor functions. The tag values depend on whether the initial data values are untrusted.

4.1.2 Tracking explicit assignments An assignment statement `lhs = rhs` updates the `lhs` variable value with the value of the expression `rhs`. The taintedness tag of the `lhs` variable should also be updated according to that of the expression `rhs`. To track this taintedness propagation, the transformation inserts additional statements into the program. These additional statements are in the form of:

```
tag = taint_get_tag(&rhs);
taint_set_tag(&lhs, tag);
```

An expression could be a constant, a variable, or a complex expression formed by sub-expressions and operators. However, when computing the taint tag for an expression, we only consider the sub-expressions of which the values are directly used in evaluating the whole expression. Sub-expressions for addressing or indexing are excluded. This is because what we want to track is how untrusted values propagate from one location to another. However, sometimes indirect expressions can also be used to propagate values. We will discuss this later.

4.1.3 Tracking Functions. Function argument passing in C follows the "copy-by-value" style, in which values of actual arguments are copied to the corresponding formal parameters upon each function invocation. To pass the taintedness tags of actual arguments into the callee function, two auxiliary stacks are used. The first stack, called *taint_in_args*, stores the taint tag of each actual input argument of invoked functions, while the second stack, called *taint_out_args*, stored the taint tag of return values of functions.

At each function call site, the tags of actual arguments are pushed onto the taint tag stack *taint_in_args* before the call, and the tag of return value is popped off from *taint_out_args* and is set to the tag of the actual receiving variable.

```
taint_push_in_arg(&arg1);
taint_push_in_arg(&arg2);
ret = foo(arg1, arg2);
tag = taint_pop_out_arg();
taint_set_tag(&ret, tag);
```

When transforming each function, at the function entry the taint tag of each formal parameter is initialized to the passed taint tag of the corresponding actual argument.

```
tag1 = taint_pop_in_tag();
taint_set_tag(&form1, tag1);
tag2 = taint_pop_in_tag();
taint_set_tag(&form2, tag2);
```

Before each `return` statement, the taint tag of the return value is pushed onto the *taint_out_args* stack so that this taint tag can be passed back to the call site.

```
taint_push_out_arg(&ret);
```

4.2 Improvements on taint tracking

4.2.1 Tracking Indirect Data Propagation Although most propagation of untrusted input data are through explicit assignments, there are certain cases where untrusted input data are propagated in an indirect manner. To precisely track the flows of untrusted input data, we must track these kinds of indirect propagation as well. However, tracking indirect flows may introduce false dependencies among program variables, and thus cause false taintedness. The biggest challenge is to find the trade-off between the precision of tracking and false alarm rates.

Propagation through translation tables. Input and output strings in web applications are usually encoded in various formats. Web applications need to decode these strings before processing them, and encode them before sending them back to users. Many encoding and decoding functions make use of translation tables. In these functions, each character in the input string, or a computed value from the input character, is treated as an offset to the translation table to get a translated value. This value may be used as the final translation result or for further translation computations. For example, to decode a `base64` encoded string, a translation table lookup as shown below is used.

```
res = reverse_table[ch];
```

To correctly track the taintedness propagation indirectly through `ch` in the above assignment, the taintedness tag of `res` is defined as depending not only on the taintedness of `(reverse_table[ch])`, but also on `ch`.

C permits uses of pointers, however, the above assignment can be rewritten slight differently by using pointers:

```
p = reverse_table;
p += ch;
res = *p;
```

Although the above statements achieve the same effect, the way to track the taintedness propagation due to `ch` is completely different. Now the pointer arithmetic involving `ch` should taint `p`. Because `p` is tainted, the value obtained by the dereference of `p` is then tainted. As a result, `res` is tainted. The method requires that pointer arithmetic and dereferences should be tracked. While this is usually enforced in taint analyses for detecting control hijacking attacks, we do not take this approach for the fear of introducing too much false positives.

Propagation through control flows. Sometimes the encoding and decoding of strings take advantage of control flows. For example, the `space` characters in URLs are usually represented as `+` signs. The following code snippet decodes such `+` signs into `spaces`:

```
if (*data == '+')
    *dest = ' ';
```

Another similar example is to convert `dots` and `spaces` in variable names to `_` characters for better compatibility. The code below achieves this:

```
switch (ch) {
    case ' ':
    case '.':
        res = '_';
        break;
}
```

In both examples, the result strings are completely determined by the input strings with the help of control flow constructs. Consequently the taint tags of the result strings should also reflect the taint tags of input strings. To track such kinds of taintedness propagation, a notion of `pctag`, the taint tag for the current program counter, is introduced. `pctag` is initialized to "untainted". Before the program steps into one branch of a control flow construct such as `if`, `switch`, `while`, the current `pctag` value is saved, and the taint tag of the condition expression is set as the new `pctag` value. When computing the taint tag value for an expression within the branch, not only tags of variables in the expression should be considered, but `pctag` should also be counted. Once the program exits from the branch, the saved `pctag` value is restored. Using `pctag`, the first example will be transformed as follows:

```
oldpctag = pctag;
pctag = taint_get_tag(data);
if (*data == '+') {
    *dest = ' ';
    taint_set_tag(dest, pctag);
}
pctag = oldpctag;
```

It is easy to see that the taintedness dependency between `*dest` and `*data` is well captured by `pctag`. Unfortunately, unrestricted uses of `pctag` will cause far too many dependencies among variables which in turn will greatly degrades the quality of taint tracking. In our implementation, we only use `pctag` to track indirect assignments through controls flows which are based on a single equality check and constant assignment, and the assignment should appear in the immediate inner branch of the control flow construct where the check appears. This can sufficiently handle the above two examples.

4.2.2 String encoding/decoding vs hashing. It is very common to encode/decode strings in web applications. The encoding/decoding processes usually involve complex computations on input strings. For example, below is part of the code used for base64 decoding:

```
result[j++] |= ch >>2;
result[j] = (ch & 0x03) << 6;
```

Certainly we would like to track the value propagation from `ch` to `result`. However, sometimes the similar computation is used to compute a hash value from the input string:

```
while (arKey < arEnd) {
   h += (h << 5);
   h ^= (ulong) *arKey++;
```

To avoid unnecessary dependencies, and thus improve the precision of taint tracking, it would be desirable to treat hash values as "untainted" all the time. Unfortunately, it is hard for the transformer to distinguish these two cases unless additional annotations are provided. Our current implementation treats these two cases in the same way and tracks both.

4.3 Dealing with External Functions.

External library functions can also propagate untrusted data from one argument to another. Ideally we should also transform these library functions to track the data propagation happened inside them. However, for various reasons (e.g. the source code is not available) sometimes we cannot transform certain library functions. To account for the data propagation effects caused by these external functions, we use wrapper functions, or summarization functions, which summarize how the associated external function propagate data values between its arguments and global variables.

Because the transformer operates on each individual file and does not hold the whole program, it has no knowledge about which functions are external. We create a small tool that traverses all the source files of a program and generates a list of all external functions used by the program. This list will then be used by the transformer as a guideline to report warnings whenever an external function needs a summarization function but the summarization function is not present.

Summarization functions are created manually based on the semantic of each function. They are compiled into standalone libraries which will be provided to the transformer for linking against transformed programs. The summarization functions are invoked in the transformed program whenever the associated library function is called.

When an external library function has no return value, its summarization function takes exactly the same arguments as the library function. When an external library function has return values, the variable that receives the return values will additionally be passed as the first argument to the summarization function. Summarization functions can make use of the taint tagmap macros to read, write, or copy taint tags.

It is rather straight-forward to write summarization functions for standard C library functions as the semantics of these functions are well defined and well known. We write summarization functions for most library functions used in the test interpreter programs. Most of them are string library functions. Other library functions include `realloc`, `realpath`, `tolower`, and so on. Below is the summarization function for `memcpy`, one of the most used library functions in the test programs.

```
void memcpy_smr
(char *dest, const char *src, int n) {
    taint_copy_buffer(dest, src, n);
}
```

Writing summarization functions for non-standard library functions is more involved. This is largely because the semantic of each function is less well known compared to standard C library functions. In our implementation, we create a number of such summarization functions for Apache library functions. Here is an example:

```
void apr_pstrdup_smr
(char **ret, apr_pool_t *p, const char *s) {
    int sz = strlen(s);
    taint_copy_buffer(*ret, s, sz);
}
```

5 Optimizations

Although the transformation presented in Section 4 brings the comprehensive fine-grain taint tracking into transformed programs, the performance penalty caused by the inserted taint tracking statements is quite high. To reduce the performance overheads, we have designed and implemented several local optimizations. As we show later in Section 6, these optimizations are very effective in improving the performance of transformed programs.

- *Use local taint tag variables whenever possible.* One major source of taint tracking overheads is frequent accesses to `tagmap`. These accesses involves expensive memory load instructions and complex bit operations for locating tag bits within `tagmap`. To reduce such overheads, we use shadow taint tag variables to store taint tags of original program variables. A shadow variable has the same scope and lifetime as the original variable itself, and is updated along with updates to the original variable to reflect changes to its taint tag. Because shadow taint tag variables are of simple data types (more precisely integers) and tend to go in registers, accesses to these variables are usually much faster.

  We only use shadow taint tag variables for program variables that have no aliases. When a program variable is aliased by other variables, the memory location represented by the variable may be updated through different aliases. We then need to track all the aliases to make sure that updates to these aliases should result in updates to the same shadow taint tag variable. This is rather difficult, especially when the program variable is passed beyond the function scope.

  To be conservative and keep the transformation simple, we restrict the use of shadow taint tag variables to only local variables. Our criteria to generate a taint tag variable for a local variable is that the address of the local variable must not be assigned to other variables, i.e., there are no aliases to the local variable.

- *Do not track local variables that are independent of tainted inputs.* One important direction of optimizations is to minimize the number of variables to be tracked. The intuition is that if we statically know that a variable is never tainted, then there is no need to track such a variable.

  We use an intra-procedural flow-insensitive dependency analysis to find out all local variables that are independent of tainted inputs. Initially all global variables and formal parameters are marked as "tracked". After that, all variables whose values may come from the "tracked" variables are also marked as "tracked". We repeat this step until no more "tracked" variables can be added. In the end, all the unmarked local variables are independent variables and require no tracking at all.

  A special case is that if an assignment is self-updating (i.e., the only variable in the rhs expression is the lhs variable, for example, `p=p+1`), there is no need to insert taint tracking statements as the taint tag of the lhs variable remains unchanged.

Here we also outline some global optimization ideas. Although we have not implemented these global optimizations, we believe that they will further significantly reduce the overheads caused by taint tracking.

- *Use global dependency analysis.* In the second local optimization we assume that all formal parameters may be tainted, and thus should be tracked. However, this is an overly conservative assumption. With the help of a simple global analysis, we should be able to find out which formal parameters should be tracked.
- *Create multiple versions of functions.* Another approach to alleviate the above problem is to use polymorphic functions. In another word, to create multiple versions of the same function, each representing different taintedness of formal parameters.

6 Evaluation

6.1 Implementation

To evaluate our technique, we have implemented a program transformation tool that instruments C programs with taint tracking statements. The transformer is written in Objective Caml and uses a nice toolkit CIL [10] as the front end to manipulate C constructs. For greater scalability, each source file is transformed separately. There are no changes required to `Makefiles`. Only the C compiler is replaced with the transformer by redefining the `CC` variable.

Using this transformer, we have successfully transformed the interpreters of two popular scripting languages, PHP4 and Bash, which are commonly used in web application development. For the purpose of evaluation, we use Apache 2 as the web server.

PHP. As an HTML-embedded scripting language, the goal of PHP is to allow web developers to write dynamically generated pages quickly. In the architecture of PHP4 interpreter, a component called *Zend Engine* acts as the language parser; an abstract layer *SAPI* is defined to allow PHP modules to communicate with web servers; and extensions such as interfaces to various databases are supported through *extension APIs*.

There are several ways by which PHP applications can get untrusted inputs: query strings (HTTP GET), form fields or hidden form fields (HTTP POST), and cookies. We identified these input points in the *SAPI* component of the PHP interpreter, and created annotations as well as marking functions for these untrusted input functions. The marking functions simply initialize all bytes of each untrusted input string as "tainted".

The PHP interpreter calls about 1,000 external functions. Many of these functions are part of the standard C library. However, there are also a number of functions belonging to other libraries, such as XML libraries and Apache libraries. We have also transformed some libraries such as the XML library `expat`. For other libraries, we carefully studied their functions, and provided summarization functions whenever needed. Most of these functions have a semantic similar to functions in the standard C library. So it is not difficult to write summarization functions for them. For example, `apr_pstrndup`, one of Apache portability library functions, is similar to `strndup`.

Bash. Not only `bash` is a handy UNIX/Linux shell, but it is a powerful shell programming language that can be used to develop advanced web applications. To get untrusted inputs, `bash` programs need to either read from standard inputs (HTTP GET), or parse the environment variables (HTTP POST and cookies). We marked all file read functions as input points where untrusted data ("tainted") may come in.

6.2 Effectiveness

We studied the effectiveness of our approach on detection of input validation attacks. Based on their popularity and availability of known vulnerabilities, three PHP web applications are selected for the experiments.

- `phpBB2` is an electronic bulletin board application. In one of its older versions, there is an input validation error which exists in `viewtopic.php` and is vulnerable to SQL injection attacks. The variable `$topic_id`, controlled by remote users, gets passed directly to SQL server in queries. Attackers can exploit this vulnerability to pass a special SQL query string which can used to see MD5 password hash for any user for `phpBB`.
- `SquirrelMail` is a web-based email client. One of its plugins, `calendar`, directly output values of user-controlled variables such as `dyear` when generating response HTML pages. Attackers can take advantage of this error and craft cross site scripting attacks.

- `phpSurveyor` is an on-line multi-user survey management system. Its user management component allows users to change their passwords by invoking the command `htpasswd` with arguments `$user` and `$pass`. Unfortunately, the values of these two variables are from untrusted inputs and not checked before being used in `htpasswd`. This makes command injection attacks possible.

We first successfully reproduced the attacks on these application in our testing environment. Then we deployed the transformed PHP interpreter module along with sensitive operation annotations and their associated security policies.

- *Detection of SQL injection attacks.* The PHP extension APIs that are used to send SQL queries to database servers are annotated as sensitive operations. The associated security policy *"no tainted strings should contain SQL keywords and meta characters"* is enforced to detect SQL injection attacks.
- *Detection of Cross Site Scripting attacks.* The `print_variable` family functions in *Zend Engine* are annotated as sensitive operations as they generate dynamic output HTML pages. The policy being enforced is *"no HTML tags (especially <script>) should appear in tainted strings"*.
- *Detection of command injection attacks.* The standard C library functions `popen` and `execve` are annotated as sensitive operations. The security policies are *"no commands should be tainted"* and *"no tainted strings should contain special characters (e.g. ';')"*.

By enabling tracking of explicit assignment and indirect propagations in the transformed PHP interpreter module, all the attacks are detected successfully, without any false alarms. Because our tracking is still conservative, sometimes data that are "untainted" are incorrectly marked as "tainted". However, only when such data are used as arguments to sensitive operations and their values violate the corresponding security policies, a false alarm will be alerted. According to our experience, the likelihood of this type of false alarms is small.

Indirect propagation plays a very important role in precisely tracking flows of tainted input strings. Once indirect propagation is disabled, we missed many attacks.

Tracking pointers, on the other hand, greatly increases the rate of incorrect taintedness, has minimal impact on improving the coverage of attack detection.

6.3 Performance

To study the performance overheads incurred by our technique, we ran the Apache server in single thread mode. We then compared the performance of the above testing applications under the original and transformed PHP interpreters. The experiments are performed on a PC with Pentium IV 1.7GHz CPU, 512MB RAM, Red Hat Linux 9.0.

Two measurements are used for performance comparison: CPU time of Apache server + PHP interpreter, and average client-side response time. The CPU time measurement can be thought of as micro-benchmark testing, while average response time reflects the real world end-user experiences. The experiments show that CPU overheads due to taint tracking and policy checking are about 100%, while overheads for client response time are about 30-40%.

Optimizations vastly improve the performance of transformed programs. The first optimization, using local tag variable whenever possible, reduces the CPU time overheads from about 400% to a little over 100%. The second optimization, only tracking variables dependent on untrusted inputs, also cuts the CPU time overheads by 20-30%. We expect to obtain an average CPU time overhead of 30-40% once the global optimizition is implemented.

7 Related Work

There have been a number of approaches to detect and prevent attacks on security vulnerabilities in web applications. These approaches can be broadly classified into three categories: testing, static analysis, and runtime monitoring. We briefly summarize some of representative approaches in each category below, and compare them with our approach.

Testing. Currently the most widely used approach to find out security vulnerabilities in web applications is to perform extensive penetration testings on the web applications [1]. In these testings, a set of random or malicious inputs are generated and sent to web applications in attempts to exploit vulnerabilities in such applications or simply crash them. Usually the inputs include the ones which can cause exploits on known vulnerabilities for regression tests. However, it is always hard for testing-based approaches to come up with a comprehensive list of test inputs. So these approaches are less effective in finding out security vulnerabilities.

Static analysis. Input validation vulnerabilities are basically the problem of insecure information flows. Information flow analysis has been researched for a very long time[2, 5, 4, 9, 19, 12, 11, 16]. Early research was focused on multi-level security, where fine-grained analysis was not deemed necessary [2]. For instance, the analysis will make the conservative conclusion that if a program read a file, then any output from the program is derived from this file. Subsequent works proposed information flow tracking at the level of variables in a program [5].

WebSSARI [7] uses a type inference based information flow analysis to analyze input validation vulnerabilities in PHP programs. [8] extends a intro-procedural pointer analysis to analyze flows from untrusted inputs to sensitive operations. Static analysis approaches are inherently less precise. They approximate program states by combining effects of multiple branches even if these branches are not all executed at runtime.

Runtime monitoring. Approaches such as application level proxy [17] use application level proxies or firewalls to validate untrusted inputs outside of the protected web applications. Although it is good that the input validation process is transparent to web applications, the two factors, semantic gaps between proxies and web applications, and multiple encodings of data, make this kind of approaches prone to both high false positives and false negatives.

Instruction set randomization techniques are proposed to detect attacks such as SQL injections [3] by hiding the actual encoding of instruction set from attackers. Unfortunately, the obscurity of instruction set is largely static and can be circumvented by attackers without too much efforts.

There have been a number of proposals to track information flow at a very low level, e.g., by using hardware tags for each memory location and register, and using the tag to track propagation of information (through assignments) at runtime [18]. The work of [14] is an independently developed, very similar approach, but using an instruction level emulator (Valgrind [13], in particular) instead of relying on hardware assistance. However, all these works were focused mainly on detecting control-flow hijack attacks. In contrast, our approach is targeting at input validation attacks on web applications, which are more data-flow dependent.

Perl has a taint mode [20] in which taintedness of variables are tracked and checked before executing security critical operations. Compared to this, our approach tracks program memory at a finer level (byte). In addition, our approach is a transformation technique for C and works for more scripting languages provided their interpreters are developed in C.

8 Summary and Future Work

In this paper, we presented a taint analysis based solution to combat input validation attacks. We demonstrated how precise taint tracking enables detection of various types of input validation attacks. We presented a fully automatic and practical implementation approach, which could be easily applied to web applications implemented in different languages.

Runtime overheads incurred by our approach are moderate: between 40 to 100 %. We have identified a few optimization strategies. As a part of future work, we will implement these optimizations to bring down the runtime overheads.

References

[1] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. In *IEEE Symposium on Security and Privacy*, 2005.

[2] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[3] Stephen W. Boyd and Angelos D. Keromytis. Sqlrand: Preventing sql injection attacks. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 292–302, 2004.

[4] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[5] J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2):143–147, May 1974.

[6] Security Focus. ePing remote command execution vulnerability. http://www.securityfocus.com/bid/13929.

[7] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D.T. Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of International WWW Conference*, New York, USA, 2004.

[8] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, 2005.

[9] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.

[10] Scott McPeak, George C. Necula, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Conference on Compiler Construction*, pages 213–228, 2002.

[11] A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[12] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy*, pages 186–197, May 1998.

[13] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Workshop on Runtime Verification (RV)*, Boulder, Colorado, USA, July 2003.

[14] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[15] OWASP. The ten most critical web application security vulnerabilities. http://www.owasp.org.

[16] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[17] David Scott and Richard Sharp. Abstracting application-level web security. In *Proceedings of*

International WWW Conference, pages 396–407, Honolulu, Hawaii, May 2002.

[18] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, Boston, MA, USA, 2004.

[19] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[20] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly, 1996.