Dynamic taint-tracking:

Paper Discussed Today:

Today’s lecture dealt with the paper discussion on the An Efficient Black-box Technique for Defeating Web Application Attacks by R. Sekar.

Today’s lecture dealt with the background building before moving onto the actual discussion of paper. It involved discussion on various injection attacks and the taint tracking paper. Black box technique paper will be discussed in next class.

Command injection: Not discussed in class. So will start from different techniques including command injection.

As we know, Buffer overflows are popular for quite some time. Biggest threats we have come across in the applications. Even today they are big threats. But if we try see the number of vulnerabilities that exist today have definitely change over the course of time. So question comes up in our mind

What comes after buffer overflows?

Most vulnerabilities reported early part of decade were due to memory corruption. 2/3rd to 4/5th of security advisories.

Web related vulnerabilities dominate today:

- Increased use of web
- Hybrid nature of web application, with server client components and mix of trusted and untrusted data.
- Less sophisticated developers.

Two application communication with each other, having one of them as client browser having JavaScript’s, Html forms and rest of the application runs on the server side. So trusting one of the things either server or client code would be there. If we are server side administrator, we gonna
trust the server code more than client side code which is on the remote side of the user which he/she should not trust and vice versa so leaving one of them untrusted and worthy of attack. So, trusting one of them makes vulnerabilities left in the system.

7% of sites found using a search technique were vulnerable!

Larger fraction was susceptible to cross site scripting attacks which we will discuss later.

**SQL injection**:-

We see time and again password being exposed. Most of the time that’s because of SQL injection vulnerability. Point being that often SQL injection is used to defeat authentication, access to data that we shouldn’t have access to. Large set of confidentiality data revealed have been occurred due to SQL injection. For example, most recent is in South Carolina couple of years ago, tax payee information in government system were compromised.

What is SQL Injection?

Attacker provided data used in SQL queries. Most of us know, web application make use of Data backend to store the data. So application interface can query data at backend that is our database to get the persistent data because web itself is made up of http which is a stateless protocol. That means in c program we tend to keep the state of our program in program variables whereas in stateless program dies each time request is submitted, so for each request we basically have a new program so we have to store it in the persistent storage to retrieve it back which is done using databases. So to look up data we make use of SQL queries to get the data and the web application construct this query on fly and submit the request.

Here is the example of PHP code which constructs the query and try send it to database:-

```php
$cmd="Select price from products where name=".
$name.
"
```

Web application trying to look at price in database. Product name is coming from browser side. $name coming from client side which is done by filling
the form at the client side browser and we have shown it in red because it is untrusted data.

- Name if provided legitimate then its fine, but attacker sitting on the other side might provide something strange like

  xyz’; update products set price=0 where name=’diamond ring’

Here, we see xyz has a close quotes not the open one, when concatenated, it gets the other one from the original query making it a legitimate attack.

**Resulting query** (quote in red. Untrusted from user)

- Select price from products where name=’ Xyz’; update products set price=0 where name=’diamond ring’

- **Well-formed query is formed, and typical SQL injection takes place. By doing this, he can change the meaning of the query which he does here by updating the price of precious expensive diamond ring to 0.**

- Attacker doesn’t always get to use the quotes in this way. Sometimes they make use of common labels so that they can use the common strings in order to construct a valid SQL injection

Let’s look at another Example:-

**Command injection:**

This attack was common to Squirrel mail which was common may be 5 years ago. IT again make use of PHP.

Here, Attacker provided data used in creation of command that is passed to the Operating system

Example: - Squirrel Mail

They make use of variable, to tap there emails by using the recipient variable send to as depicted in the below example.

Eg:
$_send_to_list=$_GET['send to']  // in php it is used to easily pull out form fields based on the label on the form field. It makes it stored in the variable send_to_list

$command="gpg –r $send_to_list" 2>&1"  // then it constructs the command by making use of program gpg. It is an encryption decryption program. It takes argument is recipient details.

popen(command)  //command needs to pull out the credentials related to the user letting us some information about the user. It executes the shell command by first constructs shell command. Which is done by popen. If you are a normal user, it works as simple email address which runs fine. 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.

Attack: user fills in the following information in the “send” field of mail

xyz@abc.com; rm –rf*

**Script injection:**

- Similar to command injection
- Attacker provide input used to create a string that is interpreted as a script.
  Common in dynamic languages since these allows string values.
  Example: PHP, python etc
- **Format string** attacks are also the same class of the attacks as we are working with command line there. Conceptually similar. Therefore technique used for detecting and preventing the script injection attacks can be used for format string attacks too due to similarity in the concepts of the attack

**Cross site Scripting (XSS):**

- Attacker provide data used as script embedded in generated web pages.
Based on the Javascript call, browser has security policy that allows it to access the javascript to all parts of the browser/web page. That’s the one aspect how the attack works.

Other aspect is that we want to use the same browser to browse different websites. As in visiting the bank website and at the same time visiting some website that we don’t really trust and we trust our browser to keep these two sites separate by that we means there should be no access to steal other site info like it should not steal information of bank. That’s what we want.

This policy is called same origin policy. Only information a script can access from the site is valid to that original source website. Not from different origin.

How the attack is made is by insert script in third party webpage. How let’s see?

Suppose we are finding a nearby branch of the bank on the website. We fill in zip code. Submit the form. How form submission looks like is below statement.

Example: http://www.xyzbank.com/findATM?zip=90100

By doing the GET in PHP, we can get the value of the variable which is 90100. It comes back if it is invalid zip code. Server sends back html response.

Normal
<HTML> Zip Code not found : 90100 </HTML>

Attacker wants to inject script in the page, so instead of providing like above, it can inject script on the web page like below and can steal the cookies to hack the accounts.

ATTACK:-
Browser is now, going to interpret it as script and fetch it and then run this malicious script. It is coming from diff site, but that script is present in the webpage of the bank, so it assumes that bank has given this authorization to script to run, so it will sent the cookie to attacker and then he can come back and login into the site with credential to perform attack.

It is also called reflected cross site scripting. As it is based on the fact that it is simply reflecting user provided data, and if it is done without data checking then it might affect the security policy.

**Directory traversal: -**

There is another style of attack known as directory traversal. The idea of it is that website might be having some policy regarding the access to the file system and certain files.

- Attacker provide path names contains directory traversal strings (example: - “/../”)
- May be disguised by various encodings. Server might classify certain directory are accessible, certain are not.

Example of the code which signifies use of public directory root for the web server, it should not be able to pass that directory if some malicious string is inputed:

Let’s us the code:

```c
void check_access(char* file){
    If((strstr(file,'/cgi-bin')==file)&&(strstr(file,”/../”)==NULL))   //can lead to root directory
    {
        //check if it is null, not present then only allow access to f
        // After doing this check problem is it is calling the function url_decode,
        So on the web , number of encoding are used, for example in this case,
```
we are using some kind of notation to send the parameters, we can use equal to. In order to that we can use something special. Or if we can use question mark as it is a special we can use encodings in order to make it possible to pass the data that can confuse the underlying encoding scheme. We can do that.

```
Char *f=url_decode(file); // This not only deals with equals but it also has some general calls using their ascii code.
/*allow access to f..*/
```

Attacker might provide file:-

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

Policy is by passed and the match code above is not matched. This is done after match, url_decode was after it, so it was not found and attack can be done by bypassing the check. If it was after that, it could have converted these special char into ../.. So it would have caught. So programmer error. Not careful programming. Order was not correct.

So the Check implemented have bugs.

**Distribution of vulnerabilities (as per 2006) CVE : 1500 odd vulnerabilities covered**

- SQL injection- 21%
- Memory error(buffer overflow etc)- 16%
- Format string 1%
- Cross site scripting- 28%
- Command injection -27% and so on.

These three are large in 20’s that means SQL injection, XSS and Command injection.

2 points can be inference from here:-

1) There is a bunch of attacks 2/3rd fall into well define categories.
2) Application specific bugs like path in application which fails to check the password authentication.

In 2009 as time passed. These three categories still seem to be biggest drivers.

- Memory - 18%
- Path traversal same
- Command injection reduced to 10%
- SQL injection still issue with 30%.

Classes of attack are not 2/3\textsuperscript{rd} anymore. They are now nearly 60%. People have become aware about these attacks. More of application specific bugs are present.

**Unified view of attack: -**

How do we detect these attacks which involve use of untrusted input?
It is where taint tracking is useful. We know from where data is coming, so we can do something with it, if we know from where it is coming.

General way of doing this is:

- Target: program mediating access to protected resources/services.
- Attack: use maliciously crafted input to exert unintended control over protected resource operation
- Resource/service access uses

As depicted in the figure above there are OS resources, command interpreters, database servers at the response ends. Data structures and interfaces.

Let’s see example: Squirrel command injection example

```
$send_to_list = S_GET['sendto']

$sendto="nobody; rm -rf "

$command = "gpg -r $send_to_list 2>&1"

$command="gpg -r nobody; rm -rf 2>&1"

popen($command)
```

Incoming Request (untrusted Input)

Program

Outgoing Request/Response (Security Sensitive operations)
(To databases, command line interpreters, backend servers etc)

popen($command)

Attack:
Removes files
Tracking from where information coming from. Variable is coming from browser. So mark it as untrusted. Red color. Program construct string. So we see two colors. Red and black. Finally command gets executed, we know something bad has happened. Intuitively we can see command from untrusted source has done some problem to the code. So it is visible with popen command.

Attack is using maliciously untrusted crafted input to exert unintended control over output operations. This is how they take control over the resources.

Even benign user is trying to control the output sequence. Question is ‘Is it an intended or intended control?’ So these are two key concepts here.

Based on taint: Degree to which output depends on input. It detects or answers the question on exertion of control. You still need to find whether it is intended or not. So for that we need policies.

Detect if control is intended: Requires policy

**Taint enhanced policy enforcements**

Input interface → program (fine grained taint tracking) → security sensitive operation
Input is coming in. Taint sources: where it comes from. Mark data as untrusted. Marking using wrapper functions. Usually marking network inputs 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 of where data come from.

Example:

- Source code transformation
- Binary translation/emulation
- Static analysis (it is not able to give level of details at runtime to make distinction between benign and malicious attack), so other two are used. Black box analysis can also be done to be covered in next lecture.

Taint sinks: where it is used. Enforce taint policy provide certain parameters and policy are applied that whether those parameters are tainted or not.

Instrumentation of taint tracking

- Fine grained tracking. Track if each byte of memory is tainted. Program can have multiple sources, so we have to keep in mind this factor. More general formulation might be needed more bit to store \( \log(n) \) bits to store taint.
- Bit array tagmap to store taint tags of every memory byte. It is a simple case.
- Tag(a): taint bits in tagmap for memory bytes at address a.

Simple arithematic operation

\[ X = y + z; \rightarrow \text{Tag}(&x) = \text{Tag}(&y) \cup \text{Tag}(&z); \]

//if tag of y or z id=s tainted, then x is tainted. Take memory address and look into tag map whether it is tainted or not.

Dereference operation

\[ X = *p \rightarrow \text{Tag}(&x) = \text{Tag}(p). \]
//p is memory location, so that’s why we are using p not &p. x is tainted if location pointed by p is tainted, then mark it as tainted.

Enabling Fine grained Taint tracking

- Source code transformation (on C program) to track information at runtime

**Taint transformations: for expression**

<table>
<thead>
<tr>
<th>S.no</th>
<th>E</th>
<th>T(E)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>C</td>
<td>0</td>
<td>Constants are tainted</td>
</tr>
<tr>
<td>2.</td>
<td>V</td>
<td>tag(&amp;v, sizeof(v))</td>
<td>tag(a,n) refers t n bits starting at tagmap[a]</td>
</tr>
<tr>
<td>3.</td>
<td>&amp;E</td>
<td>0</td>
<td>An address is always untainted</td>
</tr>
<tr>
<td>4.</td>
<td>*E</td>
<td>tag(E, sizeof(*E))</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>(Cast)E</td>
<td>T(E)</td>
<td>Type cast don’t change taint</td>
</tr>
<tr>
<td>6.</td>
<td>Op(E)</td>
<td>T(E)</td>
<td>for arithmetic/bit op, 0 otherwise</td>
</tr>
<tr>
<td>7.</td>
<td>E1 op E2</td>
<td>T(E1)</td>
<td></td>
</tr>
</tbody>
</table>

**Statements we can infer like this:**

1) V=E  v=E;  \text{tag(&v, sizeof(v))} = T(E)
2) S1;S2  Trans(S1);Trans(S2)
3) If(E) S1 else S2  if(E) Trans(S1) else Trans(S2)
4) While (E) S  While(E) Trans(S)
5) Return E  return (E, T(E))
6) F(a){S}  f(a,ta){tag(&a, sizeof(a)=ta; Trans(S)}
7) V=f(E), v(*f)(E) (v,tag)
Issues in taint tracking:-

1) Expensive especially on modern CPU. Memory access quite slow. Statements are added for each assignment. Efficiency- almost every statement is instrumented. Can introduce 4X to 40X slowdown

2) Accuracy- implicit flows, untransformed libraries.

3) Dealing with malicious code

Implicit flows accuracy issue
- Control dependence: - decoding using if then else/switch (no actual copy of data takes place in assignment.)
  If (x==”+”) y=”;

Example:-
If (E), then y=1 → // based on assignment
// Control dependence example

If (E){
  Y=1;
  Tag(y)=T(E) //if T is tainted then mark y as tainted.
}

- Negative control dependence
  y=1;
  if (x==0)
  y=0.
  So if x is tinted, but equals 1, then y is tainted at end?
  If x=1, condition fails it go out of loops. Does nothing.

- Operation involving tainted pointers.
  Char transtab[256]; ... x= transtab[p]
  Here if p is tainted, is x is tainted?
  What about following case:
  *p=’a’
Or the case:
\( X = \text{hash	able\_lookup}(p) \)

Other side of problem is, if you could track tainted, is it something you would like to do. Answer is, you don’t want to track the tainted. See some piece of code that checks:

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

This type of code is present frequently. Program gets some piece of information, and error message is logged to some file which is insecure. Not considered safe. In that case, intuitively there is control dependency in \( x \) and log message printed. This might not be confidential as it is not the exact data.

So in general if you try track the implicit flows then you tend to find everything in the program becomes tainted and you lose discrimination power between what is tainted and what is not tainted.

Discussion on the dynamic taint analysis will be dine in next class.

Class concluded.