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

Last class we talked about range of input validation attacks which we call injection attack, where attacker inject untrusted data / injecting request in such a way it violates the intended function of piece of software.

Example: General abstract/ Description of how the attack works as discussed earlier in case of Squirrel mail command injection attack.
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

So the basic idea is:

We take the program running here and we can automate it like we can do it with the help of compiler and transformed program can gauge where the request come from. If tainted, shown in red color, not tainted in black color as shown above in the example.

So we can figure out what is attack and what not by marking the tainted program. We showed it using C program in last class.

Drawbacks of this approach are there as discussed:

- Runtime overhead- slow down by factor of two, significant slowdown. Source to source transformation which is done by compiler also can significantly slowdown the program.
- Language dependence
- Intrusive instrumentation- transform every statement in target application.

Another approach we can look into this taint tracking is by looking at input and output, can we figure out that transformation took place or not? We are going to talk about that technique now. So, treating it as a black box and figure out/ estimate the information flow from input to output.

If we try to do that, it’s quiet clear, we have a situation like this where we are doing a simple string operation like concatenation, so we can see from
where input came from by simply observing what we see at the input and what comes out as output.

As long as it is a simple transformation of input to output, we can track such processing of data. Most of web application where this type of vulnerabilities are sever can be done this way. In that type of web application, simply observing input and output we can derive and benefit the instrumentation of program.

**Approach overview**

Web server talks to web application which may internal talk to database server. It is basically 3 tier web application architecture. Let us see the diagram for it.


Data input is coming from web server, protected resource in this case is database server, and so by observing data after and before web application in the above diagram, we can figure out the information flow in the data if any.

We will be looking briefly how we can do this. Before that we see what the advantages of this approach are.

- Efficient and language neutral and non-intrusive.
It consists of taint inference: black box technique to infer the taint by observing the input and output of protected apps.

Syntax and taint aware policy for detecting unintended use of tainted data.

For example:

- You take numbers and print them as strings and let say, we get the output of the program not as numbers but strings. So even that is a transformation which we simply compare and we wouldn’t see any relationship.
- So very simple transformation can create problems as it might not have any relationship.

**So, the major thing to deal with is various encoding and decoding operation that take place in web application.** As in last class we say how form data gets encoded into http request. Standard techniques like this should be included otherwise it will be next to impossible to figure out the transformations.

Once you decide that you simply going compare input and output. **One obvious way to compare is to look for output string and look for substring that might be from the input.** Or if we want it to be more general, we can look for subsequences.

For instance in case of squirrel mail that we used, if we see what actually goes in the program is if we specify list of recipients as Alice, bob and then attacker tries to inject this command after that then in the output it doesn’t appear that way.

```
“alice, bob; touch /tmp/a” produces an output“-r alice@ -r bob; touch /tmp/a”
```

- Little bit of processing is going on in the application. Output doesn’t appear exactly the same way.
- Application is adding –r flag before every email. And it is also putting add symbol. This is how this piece of code operates.
Before it calls gpg, it not only copy’s but also add few/make few small changes. If you look for small substring, and we see if it appears in the string. That’s not true. Answer is No. It is not a substring.

So if you have something like abcde, so bce is not a substring but it is a common subsequence. Mapping should be in order mapping way to be subsequence.

Is there a common subsequence between input and output? Yes, then we allow for these small transformation that take place. On strings these typical type of transformation do take place where we eliminate spaces, or add spaces, insert some characters in the middle. All of these can be used as subsequence formulation than substring formulation

So we can use the subsequence formulation then the algorithm you get is more robust and is able to find the taint in the minimum case than if we use substring based formulation

**Taint Inference algorithm**

- Standard approximate substring matching has quad time and space complexity
- Contribution of the paper is to introduce linear-time coarse-filtering algorithm ensuring taint identification

**Overview of syntax + Taint aware policies**

- Focus is on this.
- Policy question. Control intended or not?

We can come up with policy in various way, let’s look at example:-

**Policy for squirrel mail example (attack):**

Can we say something about the data which is tainted and what it is not tainted by looking at $command in the diagram. To distinguish between attack and no attack. Can we?
First is, semicolon is the thing I shouldn’t have, so tainted semicolon is an indication of attack. So it should be the case of benign attack.

What if it’s an escape semicolon, what if it’s inside the quote? It should be ok. So how to figure out which one is taint and which one is not.

I look for **commands**, which one is valid in the particular context.

So **application based policy. So lot of effort.** It would be better we give policy that do not depend on the application. So what about general policies. Which should not be differentiated on the basis of applications.

What we will do?

We will segment these request or we can say we going to parse these request at certain level and then look at the structure. So again, a valid command looks like `gpg -r` ... and if we look at syntactic structure of this request then it resembles of somewhat like figure below: -

![Diagram](image)

*Source: - taken from slides on An Efficient Black-box Technique for Defeating Web Application Attacks by R. Sekar, Stony Brook University*

So idea comes here is that tainted data is contained in a single leaf in a structure of a language. Tainted data is contained entirely into a single word when you have an attack, you have a tainted data that essentially looks like it is overflowing past the one particular word and passing the additional words in our language. So this is the essential property that we observe in the attacks.

Pattern seen in all languages? → Yes, it can have exactly same pattern based on single piece of input. Example: SQL injection. Looking price of the
product which is a literal in the query, so instead attack will have closing quote and bunch of other stuff so you will have the same characteristics which just creates multiple token from single piece of input.

So, we can summarize the syntax aware policy’s as: -

- **SpanNodes policy**: captures “lexical confinement”
  - tainted data to be contained within a single tree node
- **Straddle Trees policy**: captures “overflows.
- **Both are “default deny” policies**
  - Tainted data begins in the middle of one syntactic structure (sub tree), then flows into next sub tree.

**It is the change in the structure of query that is changing the meaning of the request. Hence we can identify.**

**One policy**— when you take untrusted input, it should go into certain individual field in output, not span multiple words. It should be contained in single words. Example: Cross site scripting (XSS) like zip code attack we saw, instead of providing bunch of tags, it consider single tag.

For most of the attacks, have this property they change the structure of command in some way, policy tends to work.

We have talked about Buffer overflow, stack smashing. Can taint tracking be used to gauge stack smashing?

- **What will you look for?**
  - See the **return address which is a memory location on the stack**, before you return, and see if it is tainted. No canary, no length to be check. Just check address, you need not know details, just get to know the change has been made. So it can be tracked.
  - Any indication of attack, corrupting a code pointer, it can be detected by simple policy. No code pointer should be tainted, function pointer etc. All the code injection attack corrupt these pointers.

Tainted pointer always tell you there is a problem.
**Reason:** Same property that thing should be contained within the particular field or unit of something ends up in something going past. In case of stack smashing attack, variable or array in the stack which may be tainted.

How well technique works is in the paper which is being assigned for the class which can be read in detail for the implementation.

Example raised in class is, if we have input of various email address having semicolon after each one of them, which is valid email address. There is nothing after the semicolon, so will the policy is considers it to be benign? So, if we try using the diagram of taint policy above, it will be consider as an attack. But actual term it is not an attack, but it is really meaningful to put the semicolon there. Is it the web user role to provide the value of parameter as in what should appear after? So, keep in mind these policy are supposed to catch most of the attack.

**How general be policy be? General at what level?**

- General at command injection, SQL injection attack level
- You have to know some of the basic, lexical roles in the language. So one should know how shell command works, or if it is a SQL, the lexical structure is different from shell language.
So in order to construct it, we should have some basic knowledge of the language.

**Attacks space of Interest (CVE 2006-07)**

All are exploits protection technique. Generalized injection attack. These are range of exploits which occurs. Most vulnerability falls in the general category and these category can be

- Command injection - 18%
- Memory errors - 10%
- XSS - 19%
- Directory traversal - 4% and so on

These are techniques like someone is running a network, web application server, applications. These are technique you will rely on. These are not tools that developers themselves find useful. They find much more useful by going through source code technique and then tell you that there are certain bugs.

So there exist certain static analysis tool that tell by going through the code. We going to talk about it.

**Model Checking:-**

It means model is supposed to capture/model the behavior the system in which we want to find bugs.

Model would be program, checking means finding bugs/errors.

How do you find error?

- Bug is incorrect behavior. It is doing what it is not supposed to do. Notion of bug is notion of correct behavior.

Finding bug means deviations from expected behavior. How do you do that? By **property specification/logical assertion**.

Eg: Let us see the code/program

```plaintext
switch (cmd){
    case “user”: 
```
Example of what might ftp server do. As there is user command, a list (ls commands), get a file and put file which ftp is supposed to do. The default case allows you to execute a shell command. Before it executes that it checks whether it is valid command or not. Reason it allows is, if you have ftp access you have full user account and once you have authenticated yourself, you can execute any shell command.

**Problem is:** - there are situations, this logic may not be correct. In particular, a bug was found which relates how ftp relinquishes its root privileges.

For example, Unix server start as root access, then depending on user account, it is reasserts the root privileges.

For ftp to work, it might need to gain root privileges so it reasserts root privileges, it binds to it and relinquish the privileges and back with user logged in privileges.

Let’s say, **Setuid(100)**, user logs in with uid 100, then u expect ftp server to execute this command. Subsequently it may do, **seteuid(0)**, where e is for effective user id.

It then supposed to **bind** and then **seteuid(100)**. This is what you see if server is operating without bug, In the case of bug, you don’t see **bind**

**Setuid(100)**
Seteuid(0)

System(..) // instead of bind

Seteuid(100)

Only bind should be there. Property might say, **whenever seteuid is done only command that can run after that is bind system call.** So we can analyze the program and we can say potential bug is there.

Idea of model checking in case of security checking is: -

- Identify property/rules for writing correct program or correspond to violation. Identify the behavior and then check depending on the property.

**Details of model checking**

You have a model. In general model can be finite state machine. For purpose of our discussion, models are finite state machine.

Very limited and simple. But all computers we use are finite state machine. Everything is finite, tapes and all. Only problem when you view them as finite state machine are that number of states are astronomical.

Eg: 100 bytes of memory, 800 bits. so finite state machine will be having $2^{800}$ states. Very large number.

**Small computers have humongous states. Biggest challenge in model checking. State exploration.**

When u try build model, you do abstraction and reduction so you model less states. Not too much concern.

What can properties be?

- Properties== finite state machine. Allows us to think the model checking problem to be automata checking problem.
  Two properties:
  - Safety ➔ bad things never happen. Eg: after seteuid bad system calls are never made.
Live ness properties: - good things eventually happen. Not in bounded amount of time. We cannot model them, it is talking about infinite time. Eventually happens, not in stipulated time. Eg: - Buchi automata. Extension of finite state machine. We do not worry about this, while finding bugs like whether Server will serve the request in 10ms time or not should not be the concern.

Safety properties can be modeled using finite state machine.

Quick example of safety property. How will we model this particular property?

So this finite state machine captures the **bad state**.

Model automata M is there, P property automata is there. So let’s see below the intersection. Does P intersection M have a path that reaches the bad state of P? That’s what it is trying to answer.

Understand this with another example:-

Construct automata that represents the model automata
Now, if we try doing automata intersection, which is nothing but running two automata simultaneously in sync we see the following model.

Here we see intersection automata reaches P3 state, so we can see if bad state is present or not.
This indicates the situation, bug is property in the program. We should remember, whenever we try to find bug is, this depends on what we understand of the bug definition.

Core idea behind the model checking-

**Take program and do analysis, abstract it and construct model like this.** Someone give you catalog of property to check. That’s how **Property automata comes about and intersection is made and you put together.**

Class concluded.