An Efficient Black-box Technique for Defeating Web Application Attacks

Example: SquirrelMail Command injection

In the previous class we discussed about various validation attacks like injection attacks. In the injection attack the attacker injects the untrusted data in such a way it violates the intended function. So for detecting that untrusted data there are several approaches. One of them was:

We have to transform the program that is running here and transform it in an automotive way and the transformed program will identify which byte came from where. The tainted portion is shown in red color and untainted is shown in black color. The example is shown in the below program.

```
Incoming Request

$send_to_list = $_GET['sendto']
$command = "gpg -r $send_to_list 2>&1"
Popen($command)

Program

Sendto="nobody:rm -rf"
$command="gpg -r nobody:rm -rf 2>&1"
Popen($command) Attack: Removes files

Outgoing Request
```

This approach has some drawbacks like run time overheads. There is also this thing that do we need to go into to the program and figure out what is tainted and untainted. No, as it is increasing overhead so it is possible to treat it as black box and check the input and output and from that figure out that there any information flow-taking place. So by
looking at the input and output we can figure out the information flow taking place or not.

Here suppose we have a situation that we are doing simple string operation like string concatenation. This app is simple and only coping the input to the output. Most of the application involves this simple procedure.

**Approach overview:**

The information is coming from data base web server.

It is:
- Efficient, language-neutral and non intrusive.
- Consists of:
  - Taint inference: Black box technique to infer taint by observing inputs and outputs of protected apps
  - Syntax- and taint aware policies for detecting unintended use of tainted data.

So, now the input and output has to be compared. One way is to do is to look at the output sting and look at the sub-strings in the input string. For general look for sub-sequences:

**For example:**
SquirrelMail, when given the “to” field value

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

This app is adding ‘@’ and –r flags symbol in the output. This is how this code operates.

If we check this input appears as the sub-string in the output the answer will be no. It is not a sub-string.

If we ask the question is there a common sub-sequence, between the input and output?

For example: In abcde, bce is not a sub-string but it is a subsequence. If we find each of the characters in the original string as well as find them in order then we can say that the string is a subsequence of the input string.

**There are two things here in figure 1:**
1. Exertion of control
2. Control is intended or unintended: it’s a policy question

**Overview of Syntax+Taint-aware policies**

(Figure 1) We can come up with policies in various ways. Can we think of a policy in such a ways, which can distinguish between an intended use and unintended use? Can we say something about that this data is tainted or untainted?

So the answer is that yes, we can distinguish which is an attack and which is not. Like in figure 1, we can see that there is a semicolon ';' in the upper second right box which separates two commands. So ‘;’ is an indication of an attack. $command=“gng-r
nobody;rm -rf” 2>&1”.

But what if it is an escape semicolon or the semicolon within quotes. Those cannot be located. So we have to figure out whether this semicolon is ok or not. So we have to come up with the policy, which can detect these attacks. So can we make a general policy and do not depend on app to app. These types of questions are still in research.

**Injection attacks and syntax aware policies:**

![Diagram](image)

We segment or parse these requests at certain level and look at the structure as shown above. A valid command looks like this gpg –r (shown in green), the syntax looks like sekar@abc.com

- Tainted data looks like a single word. When we have an attack then the tainted data starts and ends with various tainted words. This is the property that we observe in every kind of attack.
- Here we can have multiple words from a single piece of input. If suppose an entity having a single word, ends up making multiple words then that has the potential to change the structure of the command that we are sending.
- Here in the above figure if I put different value in place of ‘sekar’, then the value will be different but the structure will be same. But on the right hand side if I change the value then my structure will change.
The tainted ‘;’ is present here that matches one of the policy. If ‘;’ is inside ‘’ then it is interpreted as a part of sting then there would be no problem. As long as it is present as a single word the presence of ‘;’ does not matter. What matters is if ‘;’ makes its own word then that creates a problem. It is the change in the structure of the query, which changes the meaning of the request.

When we take untrusted input then it should go in one field and not to span in the whole structure. This kind of policy works in cross-site scripting.

Can taint tracking is helpful to detect stack smashing attack and how it is related to taint? The length exceeding the bound is bad irrespective of where the data is coming from. So the taint information is not helping us to detect stack smashing. If we look at the return address and if we check, whether it is tainted or untainted. If it is tainted then we know that there is an attack. In taint tracking we do not need the information how the data was used but only need information whether something has changed or not. This is the black box technique of detecting the attack.

So the stack smashing can be detected by a simple policy that says that no two pointers can be tainted. If we take address of a function that information is local to a program. So the address of every function must be untainted. The only reason that the address is tainted is because there is an attack. So a tainted pointer always tells that there was an attack.

Suppose we have an array, which is a variable, which gets tainted due to an attack. So what happens in stack smashing is that there is a ret address, the tainted data crosses the boundary.
Model Checking: A type of Static Analysis

Model Checking is composed of model and checking. Model is supposed to capture the behavior in the system in which we want to find the bugs. Checking means finding errors. So how do we find errors? What is the definition of a bug? Definition of bug is incorrect behavior. The notion of the bug is related to notion of correct behavior.
Example Program:

```java
switch(cmd) {
    case "user":       
        ..... 
    case "ls":       
        ..... 
    case "get":         
        ..... 
        default :   
            if(authenticated validCmd(cmd)) 
                system(cmd);
    }
```

It has the default in which it executes the shell command. The reason why it allows us to do is that it has if we have FTP access then we have full root access. Once we have authenticated our self as normal user then we can execute any shell command. The problem though there are situation that the login may not be correct. A bug is found which relates to FTP. So FTP server works as, in Unix the server starts as the root process, when the user logs in then they change the user id to the user who just logged in. The way the FTP server works is that there are situations were FTP server wants to regain the root privileges for actual transfer of file, the program needs to open files. What FTP server does is that it reasserts it’s the root privileges and then uses the this root privilege to bind a call Then it releases the root privilege and now we are back into the normal user. There is possibility that something may get wrong. The way it suppose to happen is:

Example:
step 1: seteuid(100) //set effective user id
        //the user logs in and has the user id:- Seteuid :- 100
step 2: seteuid(0)
step 3: bind(…)
step 4: seteuid(100)

**If there is bug then we will see:**
step 1: seteuid(100)
step 2: seteuid(0)
step 3: system()    // will execute the shell
step 4: seteuid(100)  
This command execute as root privilege.

To prevent this the property will say that when seteuid is done, after that only system call that can be executed is bind.

step 1: seteuid(100)
step 2: seteuid(0)
step 3  bind()
step 4: seteuid (100)

What is a model?
In general a model can be a FSM (Finite State Machine) and other models as well. But the purpose of our discussion we will consider model as finite state machine. FSM are very limited and very simple but every computer these days uses them. The only problem in viewing every thing FSM is that with the number of states. For example, a machine with 100 bytes of memory has a FSM which has $2^{800}$ bits (states). It is a very large number.

**Model = Finite State Machine**

When we develop the models then we do abstraction and reduction.

**Properties = Finite State Machine**
We can make the property to be finite state machine.
Model is automata and property is a automata. The intersection gives a model, which is common of both the models

There are two kinds of properties:
1. **Safety**: some bad things never happen. Common security property. We say that after seteuid(0) no bad system call would be made.
2. **Liveness**: means that good things eventually happen. If we think of it as the property as model by FSM, we cannot model liveness property as it is talking about infinite paths. They require what are called Buchi automata.

We are concentrated in safety properties only.
Example of modeling a safety property:

**Automata P**

```
other
P1
isetueid(0)
P2
Bind(…)
Initial state
isetueid(x)
where x != 0
other system calls
P3
BAD
BAD state
```
Automata P

This finite state machine captures the bad state.

Automata M

We are asking question that does P intersection M have a path that reaches the bad state of P.

Intersection automata

We take both the automata like M1, P1 there is only one transition and that transition is also possible on P1 and so. We will see here that intersection automata reaches state <M3,P3> and check that whether bad state is present or not.
Idea behind the model checking-

Do analysis of the program and construct FSM model like discussed above. Also we will have the properties to be checked. That is the way we make the property automata and intersection is made and we put together.

Class concluded.