

Lecture Notes for 04/04/06:

UNTRUSTED CODE

Fatima Zarinni.

Last class we started to talk about the different System Solutions for Stack Overflow. We are going to continue the subject.

Stages of Stack Overflow:

The following are the steps for performing a buffer overflow attack:

- 1) Find a bug in the code.
- 2) Send overflowing input.
- 3) Overwrite return address to point to buffer.
- 4) Jump to *RA
- 5) Execute code.
- 6) Make System Calls.

Solutions for Each of the Above Steps:

- 1) Static Analysis.
- 2) CCured.
- 3) Address Space Randomization(ASR) and PointGuard
- 4) Stack Guards
- 5) ISR (Instruction Set Randomization)
- 6) System Call Randomization.

We talked about Steps 1 through 4, today we are going to talk about steps 5 and 6.

Instruction Set Randomization:

Instruction Set Randomization is used in order to prevent an attacker from running his injected machine code on the victim machine.

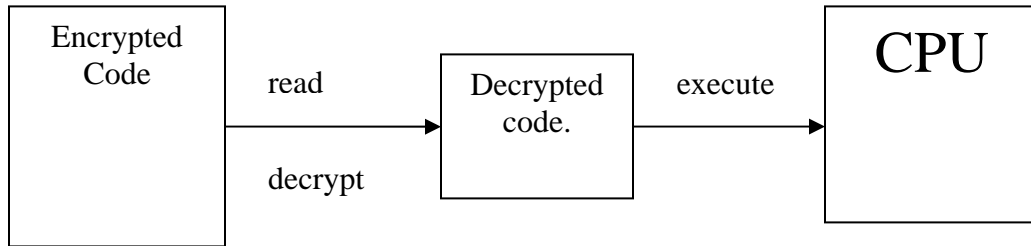
Assume that an attacker finds a way to inject some machine code on the victim machine. Now, the attacker wants to inject machine code that would be executable on the victim CPU. So, in order for the attack to work the attacker needs to know the following:

Attacker must know:

- OS of the victim machine.
- Instruction Set of the victim machine.

So, our goal is to not allow the attacker to know the instruction set of our machine. We can do the following:

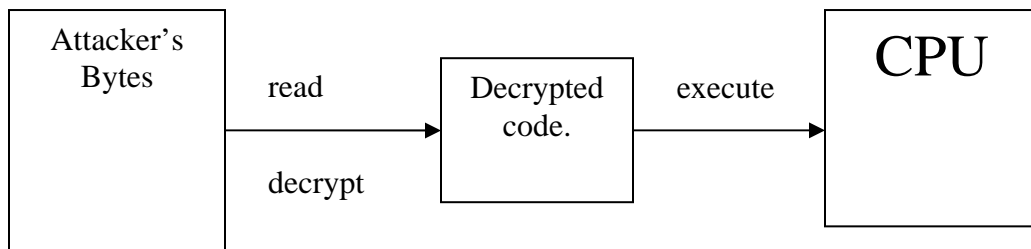
- 1) When executing code, we encrypt it and load the encrypted version into memory.
- 2) Right before executing an instruction, we decrypt it and give it to the CPU.



In the above case encryption is only for code and not for data. The idea is similar to PointGuard.

So, now what happens when attacker finds vulnerability in the code and tries a code injection buffer overflow attack?

Assume that the attacker successfully injected his code in the memory and made the return address point to his injected code. So, when the program gets the return address, the attacker succeeds in making the PC point to his bytes. So, now the following scenario would happen:



CPU decrypts the Attacker's bytes before executing. So, if attacker did not know the Key the decryption process would give random bytes and then these random bytes would be executed by the CPU. Hence, if the attacker did not know the Key, it is less likely that he would get what he wants. Now, if we feed anything to the CPU, then the program would just crash.

Thus, the above technique is called Instruction Set Randomization.

Let us look at some comparisons:

- ASR:
 - Can be used on a binary file very automatically.
 - You do not need to change source code.
 - It is very easy to use.
 - Weaker guarantee, but less work

- Static Analysis:
 - Strongest Guarantees.
 - Lots of hard work.

- CCured:
 - Works when there are buffer overflows.
 - Moderately successful.

- StackGuard:
 - Stack Guard is quite successful.

- Point Guard:
 - PointGuard is not successful.

- ISR:
 - Too much overhead
 - It only works for code injection buffer overflow attacks.
 - Not Successful.

Now, let us look at System Call Randomization.

System Call Randomization:

Permanent damage can be caused by system calls done by the attacker. Hence, we need to prevent attacker from performing system calls.

So, how does a system call work?

Let us look at the following assembly code:

```
push eax    (number of bytes.)
push eax    (address of buffer.)
push ebx    (file descriptor.)
push 127    (read system call. 127 represents a system call type, maybe a read syscall.)
int 0x80
```

The last line asks the OS to perform the system call.

How can we prevent an attacker from performing system calls?

- Randomize system call numbers.
 - So, each time when a program is run, all system call types would get a different number.
 - In our example above, 127 is a system call number.
- Randomize system call argument orders.
- Change system call instructions.
 - Update system call table.
 - Change C library.
 - Change Kernel.
- Rebuild the entire system to perform random system calls.

At what stage of the buffer overflow attack would System Call Randomization be used as defense?

After the attacker succeeds in overflowing the buffer and changing the return address in order to perform a return-to-libc attack and when he uses code already present, to do system calls.

For System Call Randomization, defense has high overhead and low security.

[New Topic Starting from Next Page]

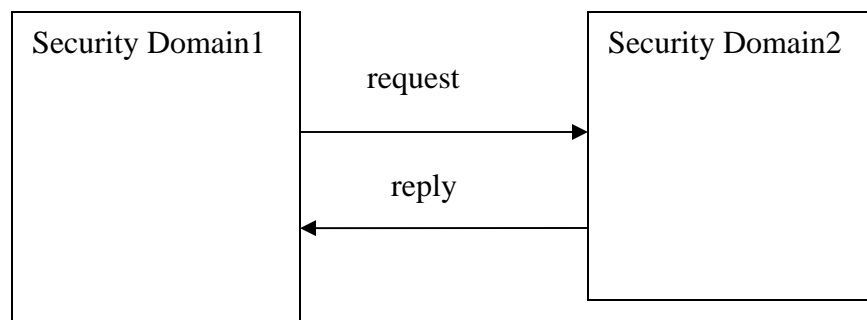
Software Fault Isolation:

Theme: The direction of Computer Security research is towards finer grain sharing.

In the beginning of the semester we saw the example of a closed room with an operator. There was no sharing on that system. Then we saw the VM architecture in which VMs did not share memory, but they could send messages back and forth. Now, one process can make request to another process on Unix.

The problem is that everything is very slow.

Sharing and IPC:



Unix Processes.
(Using Pipes)

(Pipes sort of work like method invocations. Security Domain 1 requests something from Security Domain 2, and Security Domain 2 replies back to Security Domain 1.)

- Messages can go back and forth via pipes.
- Requests go as string of bytes.
- Replies come back as string of bytes.

A lot of things have to happen for making messages go back and forth between processes:

- There must be a context switch to stop 1 process from running and save the state of that process. Then load the state of the second process, and start running the second process.
- So, Context Switches are expensive.

So, now let us look at some efficiency issues that arise due to calls between different domains.

Function call	0.1 micro seconds
Pipe IPC (Send 1 byte over and get 1 byte back)	200 microseconds. (2000 times slower than function calls.)
SFI IPC	1 micro second.

Since Pipe IPC is very slow, therefore we cannot make a lot of these calls. If we want to transfer a lot of data, Pipe first copies data from domain 1 to the OS, and then from the OS it copies to domain 2.

One important operation in SFI is that it reduces the number of copies to just 1 copy.

Our Goal:

Keep two security domains that do not trust each other in a single address space and do calls much faster.

How do we do faster inter domain communication?

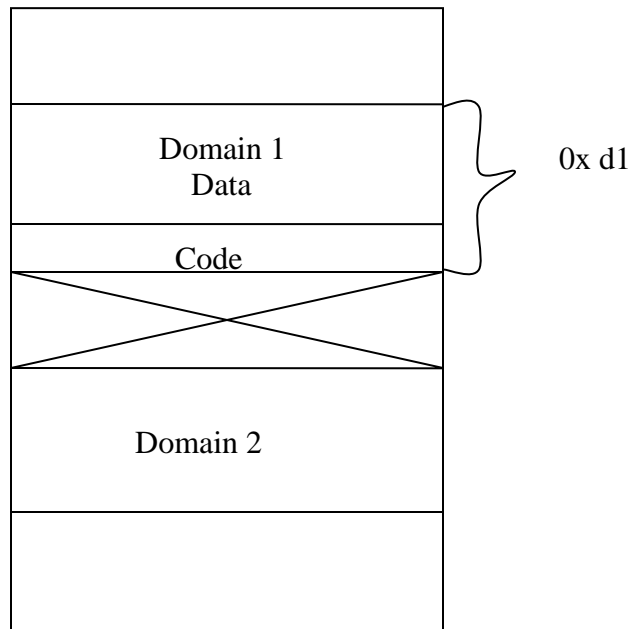
We must get rid of Context Switches.

So, the following is what we want:

- Domains must exist in the same address space and process thread.
 - Inside a 1 Unix process we can have two domains that do not trust each other and live in the same address space and run correctly.
 - If we could do that then I don't need a pipe or context switch between the two domains.

- Is memory protection an issue?
 - System call overhead is so much that we do not want to involve the OS.

So, again, we want to make two domains live in the same address space without allowing them to step on each other's toes. How can we do that?



Address Space

So, we want to make sure that whenever process is running code in domain 1, we cannot step on stuff that is in Domain 2, and vice versa.

Idea:

To be able to do the above we must precede every memory access by the process with a bounds check.

So, let us look at the following sample code:

Domain 1:

```
add eax ebx
mul ecx eax
mov ecx [eax] ← Moves ecx to memory location at eax. We must check that eax points
                into domain 1.
```

So, we need to fix the above third line of the code.

Rewriting the code with the fix, which ensures that eax points in to a location of domain1:

Domain 1:

```
add eax ebx
mul ecx eax
sethib eax 0xd1 // Sets the high bytes of eax to 0xd1
mv ecx [eax]
```

So, this fixes the problem of domain 1 trying to access domain 2's memory locations. Is there any way we could get around this fix?

We can have a jump statement right before the sethib statement, in order to jump to the "mv ecx [eax]" instruction. To prevent this we must have Control Flow integrity.

Control Flow Integrity:

Sub Problem to Solve to be Able to Achieve our Goal:

- All basic blocks must execute from the beginning.

What is a basic block?

A Block in a program is a sequence of consecutive instructions that do not have any jump statements, except that the last instruction can sometimes be a jump. This jump statement gives us the decision of where to go next.

Example:

Let us say that we have the following C code:

```
x = 7;
y = x + 5;
if (x > 11) {
    printf ("Hello");
}
else { exit(0);
}
```

The corresponding assembly code is:

Basic Block 1:

```
mov 7 eax
add eax 5 ebx
comp ebx, 11
jge L1
```

Basic Block 2:

```
:
:
:
call printf
jmp L2
```

Basic Block 3:

```
L1:
    call Exit
```

// End of Example!

So, we want the code for Domain 1 and Domain 2 to execute basic blocks from the beginning, because we want to make sure that all accesses to memory locations are preceded by checks.

So, for example, we want to make sure that
“sethib eax 0xd1” is executed always before
“move ecx [eax].”

So, Control flow integrity would do the following:

- It would introduce a new instruction “label Tag” at the top of each block.
- “label” does not do anything.
- “TAG” is a 4 byte number
- We must make sure that each jmp statement is going to the label instruction of a building block.

So, for example control flow integrity would add the following instructions to a block of the assembly program:

```
Label TAG
:
:
cmp [eax + 4], TAG // Check if jump would actually jump to the beginning of a
jne ABORT          // basic block. If not then abort.
:
:
jmp eax
```

Now, there is another issue. Code comes from un-trusted domain, so how do we know that checks exist?

- We can re-write the code dynamically to include checks.
- We can have compiler to insert the labels and checks.