

04/04/06 Lecture Notes: Untrusted code

Beili Wang

Stages of Static Overflow:

1. Find bug in code
2. Send overflowing input
3. Overwrite return address with point to buffer
4. Jump to *RA
5. Execute code
6. Make system call

Solution:

1. Static Analysis
2. CCured
3. Address Space Randomization (ASR), PointGuard
4. StackGuard
5. Instruction Set Randomization
6. System call Randomization

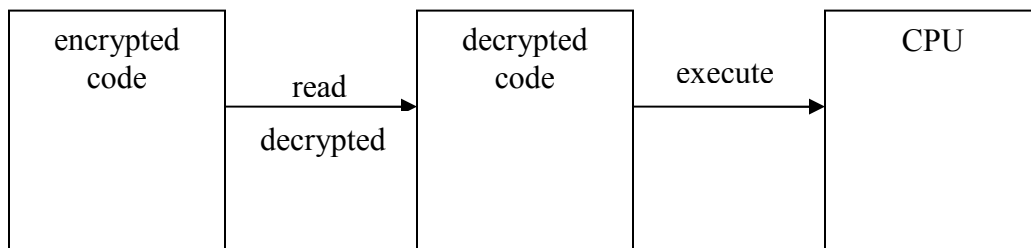
Last lecture talked about stage 1 to 4 and corresponding solution 1 to 4. This lecture will talk about the rest.

Instruction Set Randomization:

Attacker injects code and runs the code on the victim's machine, so attacker need to know about the victim:

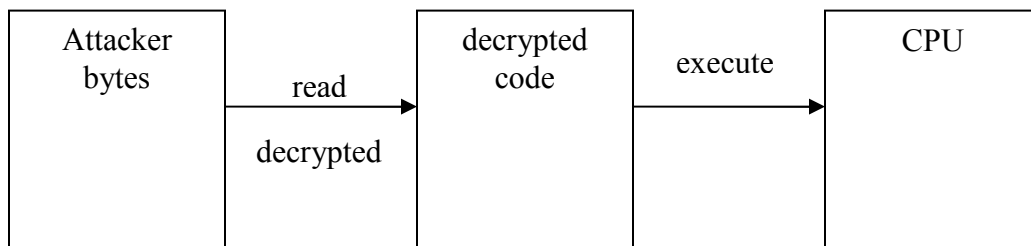
Attacker must know:

- OS
- Instruction set



The idea is similar as the PointGuard, here code is encrypted and data is not.

Let's look at attacker code:



CPU decrypts the bytes. Attacker does not know the key so random bytes are generated. CPU tries to execute random bytes. It is very unlikely attacker gets what he wants. What happens when CPU executes random bytes? It depends on architectures. Some will cause crash, some will produce invalid results.

Static Analysis has strong guarantee but requires a lot of work.
CCured works when there is buffer overflow. It is moderately successful.

Address Space Randomization has randomization on binary and has weak guarantee,
require little work. PointGuard is not successful

StackGuard is successful.

Instruction Set Randomization offers low protection. It assumes that there is injection.
Instruction Set Randomization has overhead. It is not successful

Instruction Set Randomization can be thought as of disk encrypted or encrypted on load
time. Some buses that connect memory to CPU are encrypted. Idea is similar as the cable
that connects between computer and monitor is encrypted. Movie studio prevent attack
by encrypted the cable.

System Call Randomization: Stop attacker from system calls.

```
push ecx    //number of bytes
push eax    //address of buffer
push ebx    //file descriptor
push 127    //read system call
int 0x80
```

Things can be randomized:

- Randomize system call numbers. In this example, randomize system call number 127.
- Randomize system argument orders
- Change system instructions
 - o Update system call table
 - o Change C library
 - o Change kernel
 - o Change application
- Rewrite entire system with random system call

Possible attack:

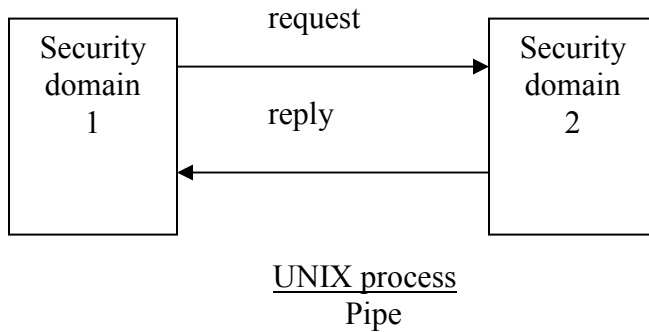
- return-to-libc attack
- reuse the code already here to do system call
- libc calls instead of system call

Above is the system techniques deal with buffer overflow. Next is a new topic: Software Fault Isolation.

Computer system security research on:

System has finer and finer cooperation and sharing (more and more sharing at time).
Problem is it's all very slow.

Sharing and IPC



Security domain 1 writes to file

Security domain 2 reads from file and processes request, writes to same file

Function call 0.1 us

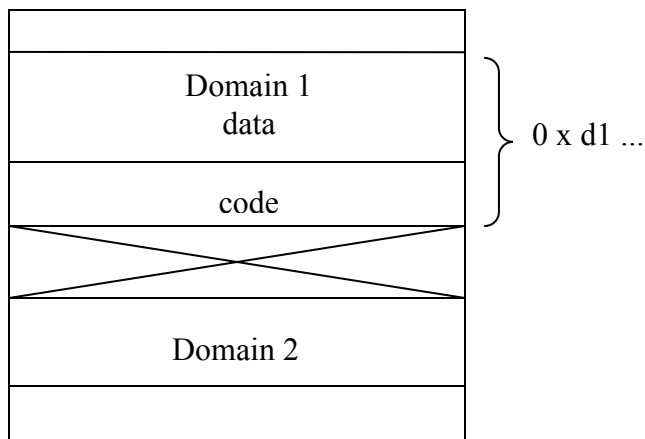
Pipe IPC 200 us (200 x slower)

SFI IPC 1 us

Goal: how do we do function calls really fast, when two domains live in the same address space and do not trust each other.

- Domains must exist in the same address space and process thread.
- Memory protection? (How to do that?)
 - o System call overhead
 - o Context switches overhead

Address Space



Precede every memory access with a bounds check.

Domain 1

add eax ebx

mul ecx eax

```
mov ecx [eax] // move ecx into memory location, pointed by eax, need to check eax
               points into domain 1
```

Rewrite the code: check that eax point into domain 1

Domain 1

```
add eax ebx
mul ecx eax
sethib eax 0xd1 // set the high byte eax to d1
mov ecx [eax]
```

What if after “mul ecx eax” jump to “mov ecx [eax]” directly? Control Flow Integrity

Control Flow Integrity

Subproblem: all basic blocks must execute form the beginning.

Reminds basic block (c code)

```
x = 7;
y = x + 5;
If (x > 11) {
    printf (“Hello”);
}
else {
    exit(0);
}
```

Definition: Decision what to go next is a basic block.

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
```

Always want execute basic blocks from the beginning. Make sure all moves have protected checks.

For example, make sure “sethib” always before “mov”.

New Instruction: Label TAG

Every basic block begins with “label” instruction.

```
Label TAG
:
:
cmp [eax + 4], TAG //before jump instruction check jump will jump to the TAG
jne ABORT //if not match. then ABORT
:
:
jmp eax
```

Simple check:

```
XOR eax TAG
XOR eax TAG
```

How do we have these checks: 3 options:

1. check when compile
2. check when load
3. rewrite code