Emulation based analysis using binary instrumentation

Application on CTF

SPEAKERS

Myunghun Cha

- From Republic of Korea
- POSTECH senior student majoring CSE
- Team Leader of PLUS
- CODEGATE 2009 Hacking Contest 3rd place
- DEFCON 2009 CTF 3rd place
- DEFCON 2011 CTF 8th place
- Many hacking contest experience

Jinsuk Park

- POSTECH sophomore majoring ME
- Team member of PLUS

PLUS

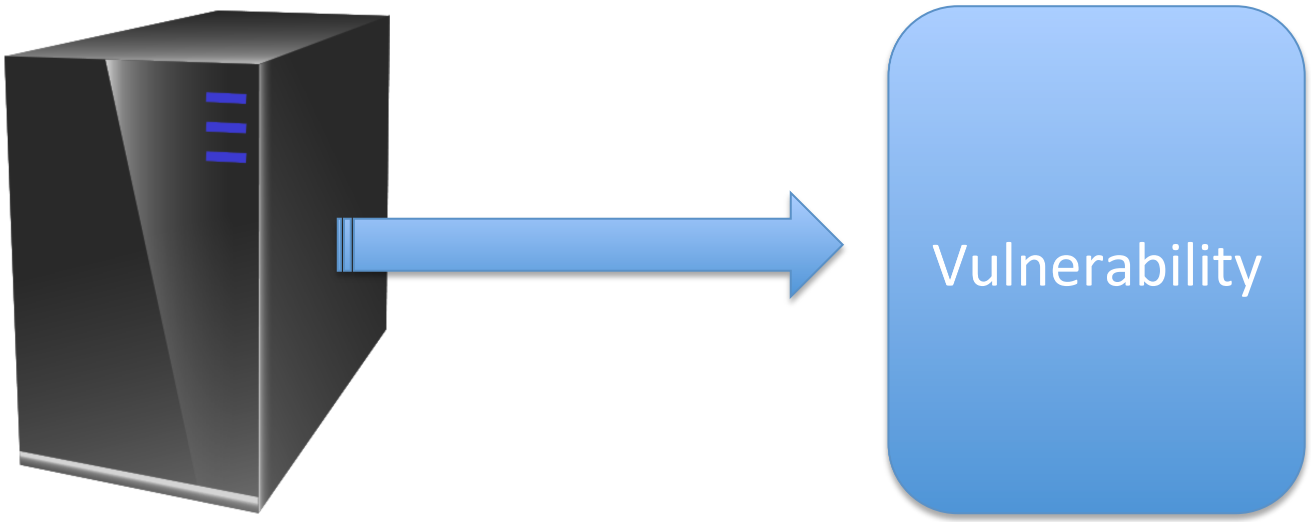
- POSTECH Laboratory for UNIX Security
- Found in 1992
- Researching on various security issues
- Participating in lots of hacking contests
- Participated in DEFCON CTF three times
  - 2009 (3rd)
  - 2010 (3rd)
  - 2011 (8th)
  - 2012

Motivation

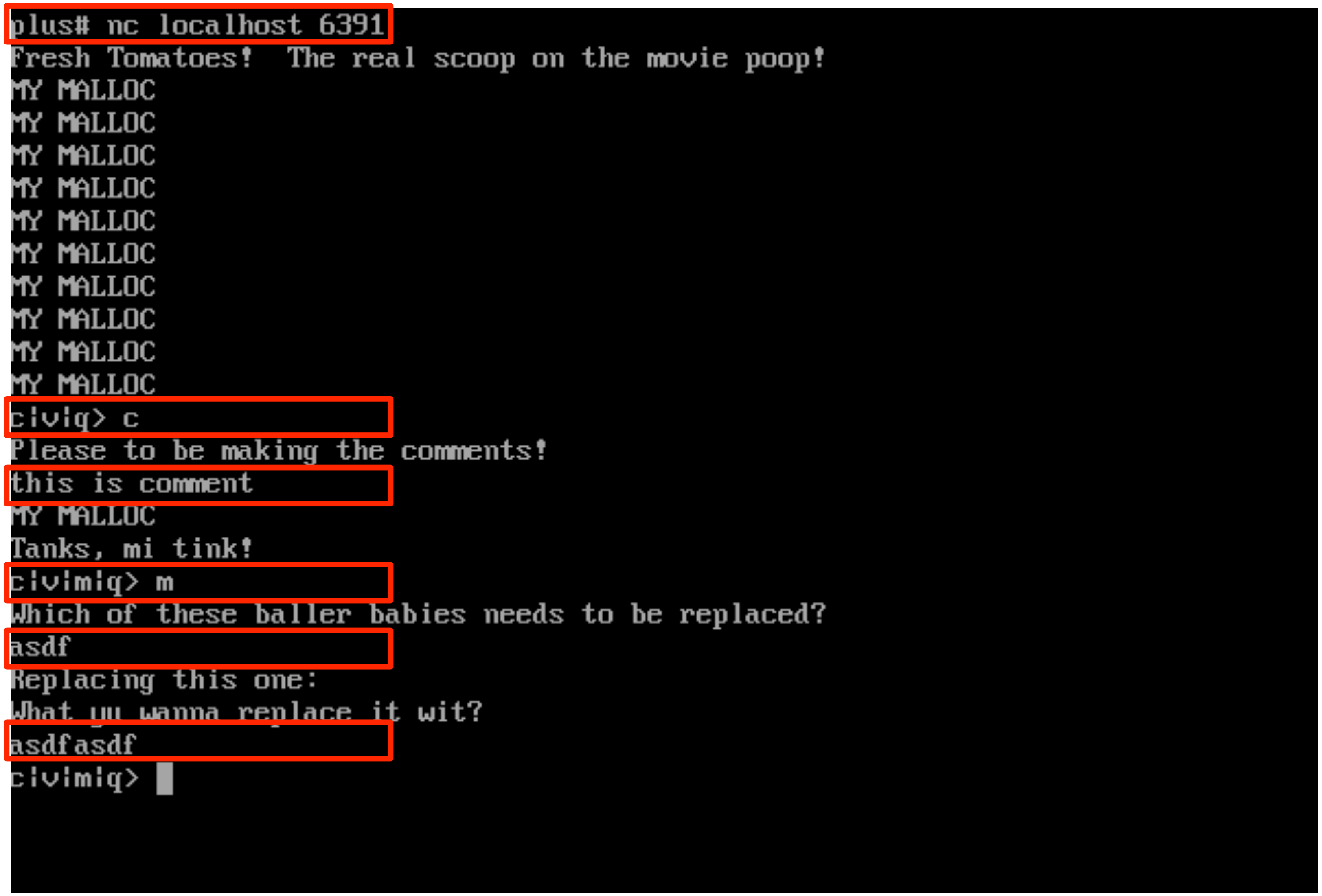
DEFCON CTF

CTF Basic Rule

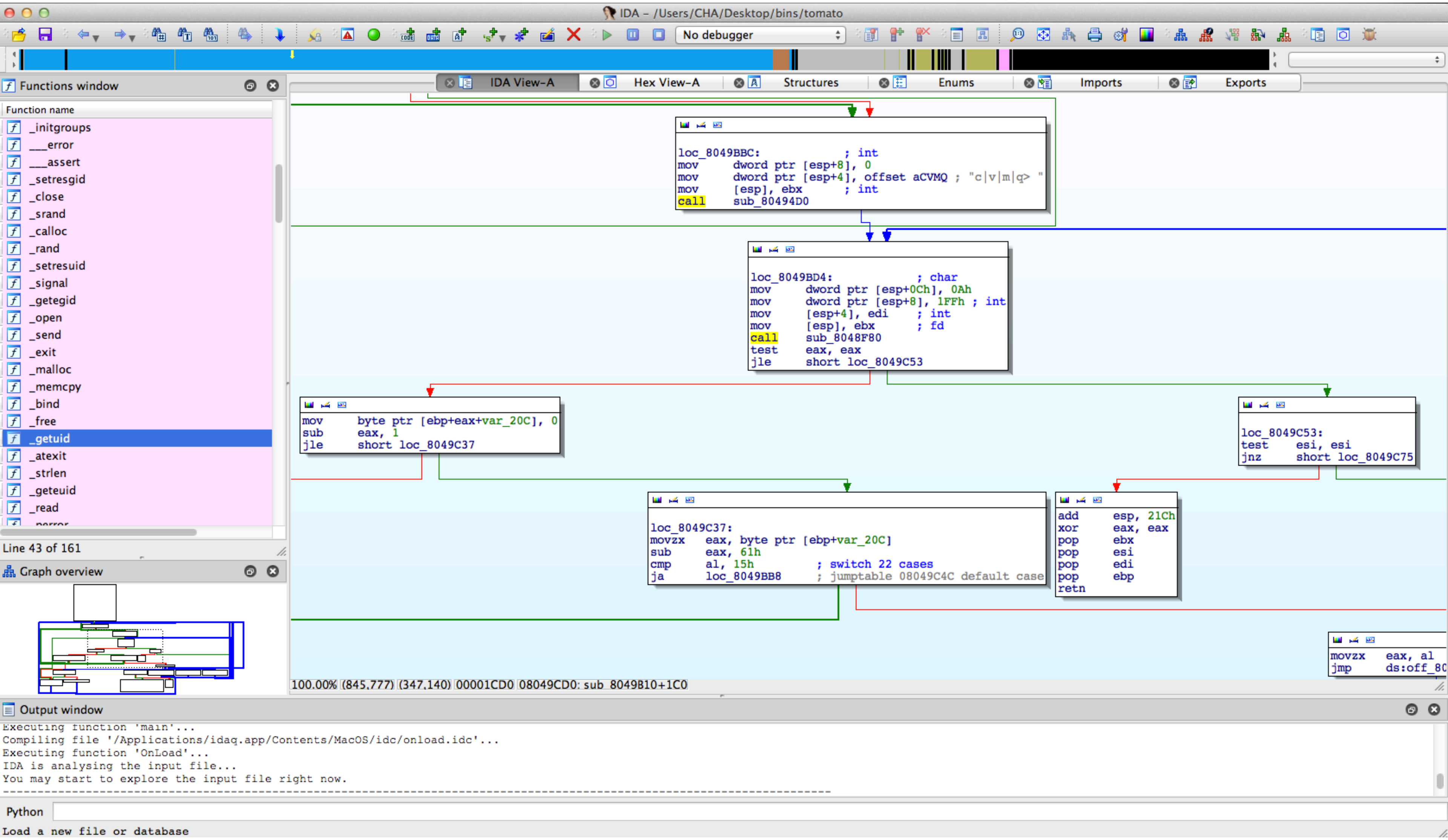
- CTF : Capture The Flag
- Each team is given vulnerable server
- Vulnerable daemons are running on the server



CTF Daemon



```
plus# nc localhost 6391
Fresh Tomatoes!  The real scoop on the movie poop!
MY MALLOC
MY MALLOC
MY MALLOC
MY MALLOC
MY MALLOC
MY MALLOC
MY MALLOC
MY MALLOC
MY MALLOC
MY MALLOC
c|v|q> c
Please to be making the comments!
this is comment
MY MALLOC
Tanks, mi tink!
c|v|m|q> m
Which of these baller babies needs to be replaced?
asdf
Replacing this one:
What yu wanna replace it wit?
asdfasdf
c|v|m|q> █
```



```
Function name
_initgroups
___error
___assert
_setresgid
_close
_srand
_calloc
_rand
_setresuid
_signal
_getegid
_open
_send
_exit
_malloc
_memcpy
_bind
_free
_getuid
_atexit
_strlen
_geteuid
_read
Line 43 of 161
```

```
loc_8049BBC:                          ; int
mov     dword ptr [esp+8], 0
mov     dword ptr [esp+4], offset aCVMQ ; "c|v|m|q> "
mov     [esp], ebx          ; int
call    sub_80494D0
```

```
loc_8049BD4:                          ; char
mov     dword ptr [esp+0Ch], 0Ah
mov     dword ptr [esp+8], 1FFh ; int
mov     [esp+4], edi        ; int
mov     [esp], ebx          ; fd
call    sub_8048F80
test    eax, eax
jle     short loc_8049C53
```

```
mov     byte ptr [ebp+eax+var_20C], 0
sub     eax, 1
jle     short loc_8049C37
```

```
loc_8049C53:
test    esi, esi
jnz     short loc_8049C75
```

```
loc_8049C37:
movzx   eax, byte ptr [ebp+var_20C]
sub     eax, 61h
cmp     al, 15h             ; switch 22 cases
ja      loc_8049BB8         ; jumptable 08049C4C default case
```

```
add     esp, 21Ch
xor     eax, eax
pop     ebx
pop     esi
pop     edi
pop     ebp
retn
```

```
movzx   eax, al
jmp     ds:off_80
```

```
100.00% (845,777) (347,140) 00001CD0 08049CD0: sub_8049B10+1C0
```

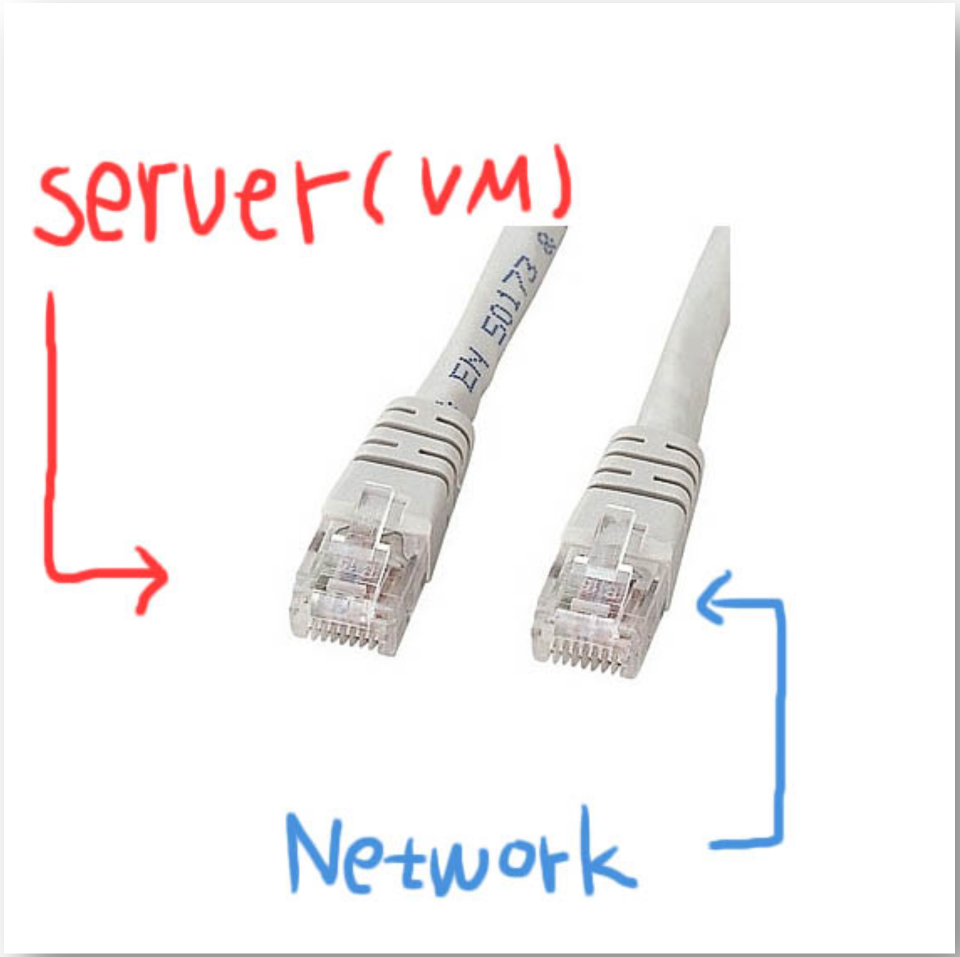
```
Executing function 'main'...
Compiling file '/Applications/idaq.app/Contents/MacOS/idc/onload.idc'...
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
-----------------------------------------------------------------------------------------------
```

```
Load a new file or database
```

Scoring

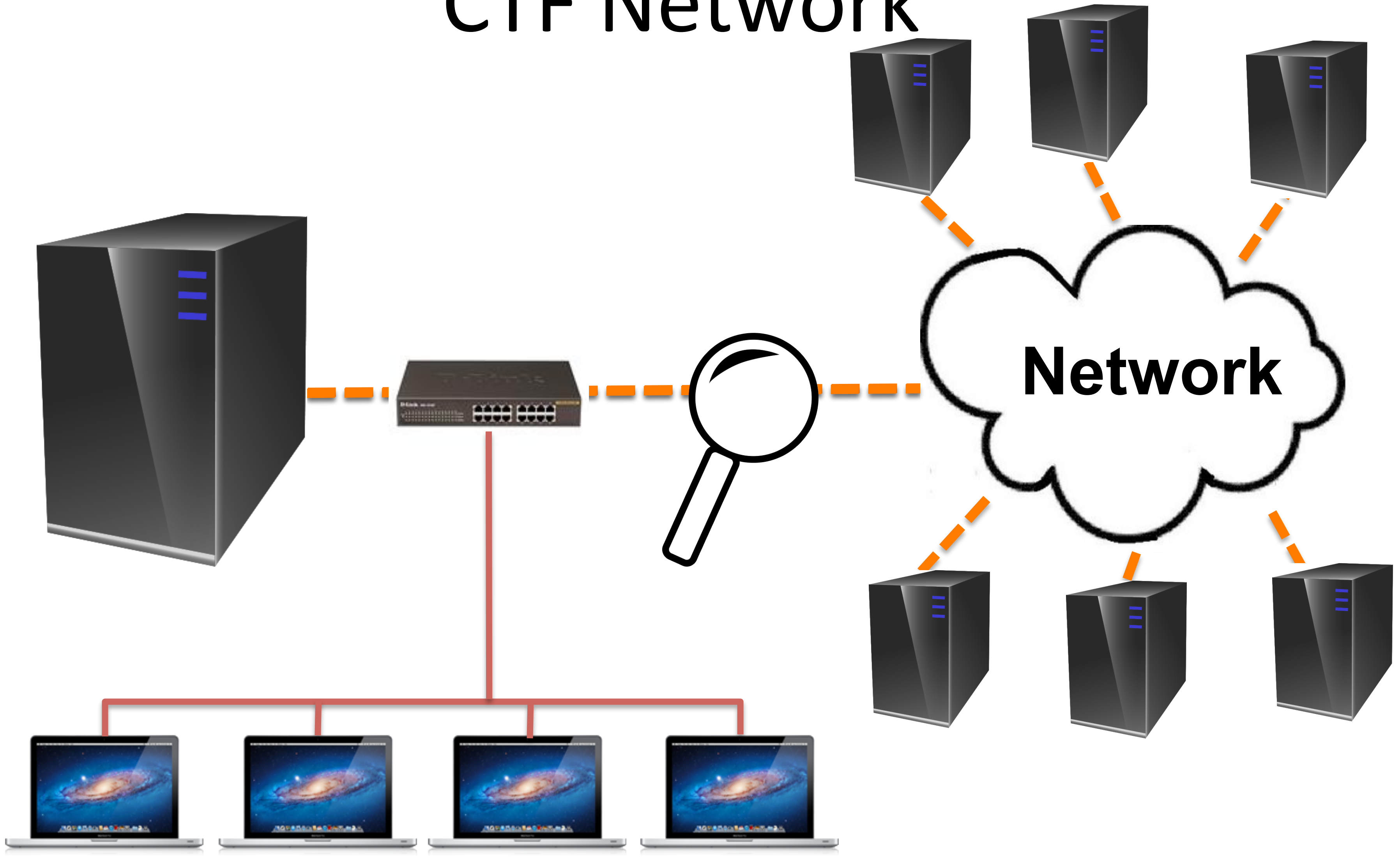
- There's a key file for each daemon which is changed periodically
- You should read or write the key file to get a score
- It simulates information stealling and corruption in real world

CTF Network



Given two lan cables

CTF Network



CTF Summary

- We can attack over the wire
- We can sniff, suspect, or drop packet
- We can attack analyzing binary

  or using other teams' exploit

What do I want to do?

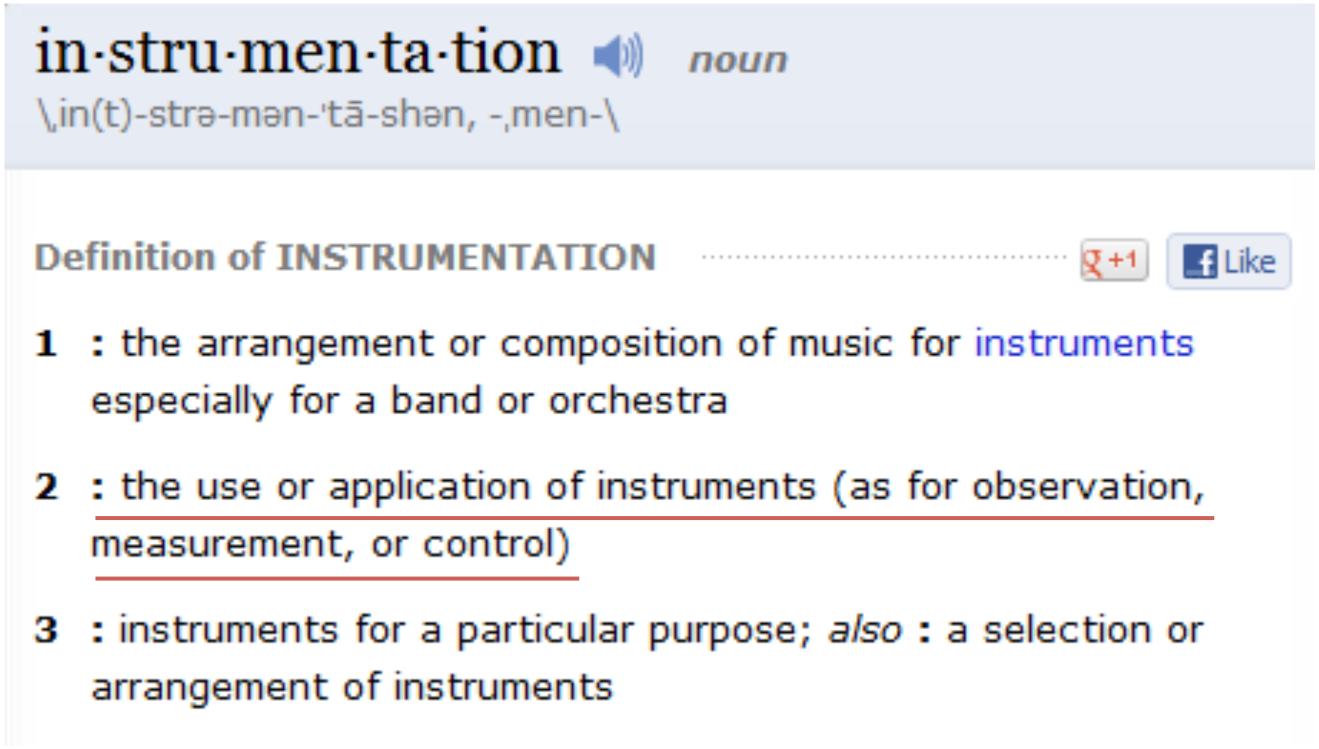
- I want to detect attacks
- I want to analyze vulnerability easily using other teams' attack
- Then… how?

EMULATION BASED ANALYSIS

Emulation Based Analysis

- We can detect bug following specific patterns
  - Stack boundary check
  - memcpy without string length check
  - EIP address check
  - Format string from user input
- Verification user input is much more easier than finding hidden bug
- Dynamic analysis is easier than static analysis

Instrumentation?



in·stru·men·ta·tion *noun*

\ˌin(t)-strə-mən-ˈtā-shən, -ˌmen-\

Definition of INSTRUMENTATION

1 : the arrangement or composition of music for instruments especially for a band or orchestra

2 : the use or application of instruments (as for observation, measurement, or control)

3 : instruments for a particular purpose; *also* : a selection or arrangement of instruments

Dynamic Binary Instrumentation

- Ability to monitor or measure the level of a program's performance, to diagnose errors and to write trace information

Dynamic Binary Instrumentation

- A technique to analyze and **modify** the behavior of a binary program by injecting arbitrary code at arbitrary places **while it is executing**

Usage

- Simulation / Emulation
- Performance Analysis
- Program optimization
- Bug detection
- Correctness Checking
- Call graphs
- Memory Analysis

For hackers?

- Fuzzing
- Covert Debugging
- Exploitable Vulnerability Detection
- Automated Reverse Engineering
- Bypass Anti-Debuggers
- Automated exploitation
- Automated unpacking

DBI frameworks

- Pin
- **Valgrind**
- DynamoRio
- Etc.

Why valgrind?

- Valgrind runs on BSD but PIN does not
  (which is DEFCON CTF Environment)

Valgrind : Introduction

- Valgrind Core
  - DBI framework
  - Simulated CPU
- Valgrind tool
  - Written in C using Valgrind framework
  - Used as Plug-ins for Valgrind
- Valgrind Core + tool plug-in = Valgrind tool

Valgrind : Tools

- Memcheck: check memory management of the binary executable
- Cachegrind: cache profiling
- Helgrind: Data races conditions detection
- Massif: Heap profiler
- User written tool
- usage: valgrind --tool=<toolname> [options] prog-and-args

Valgrind : How it works

1. Disassembly

Machine code(x86)
↓
Intermediate Language(IR)

2. Instrumentation

IR
↓
Instrumented IR

3. Assembly

Instrumented IR
↓
Machine code(x86)

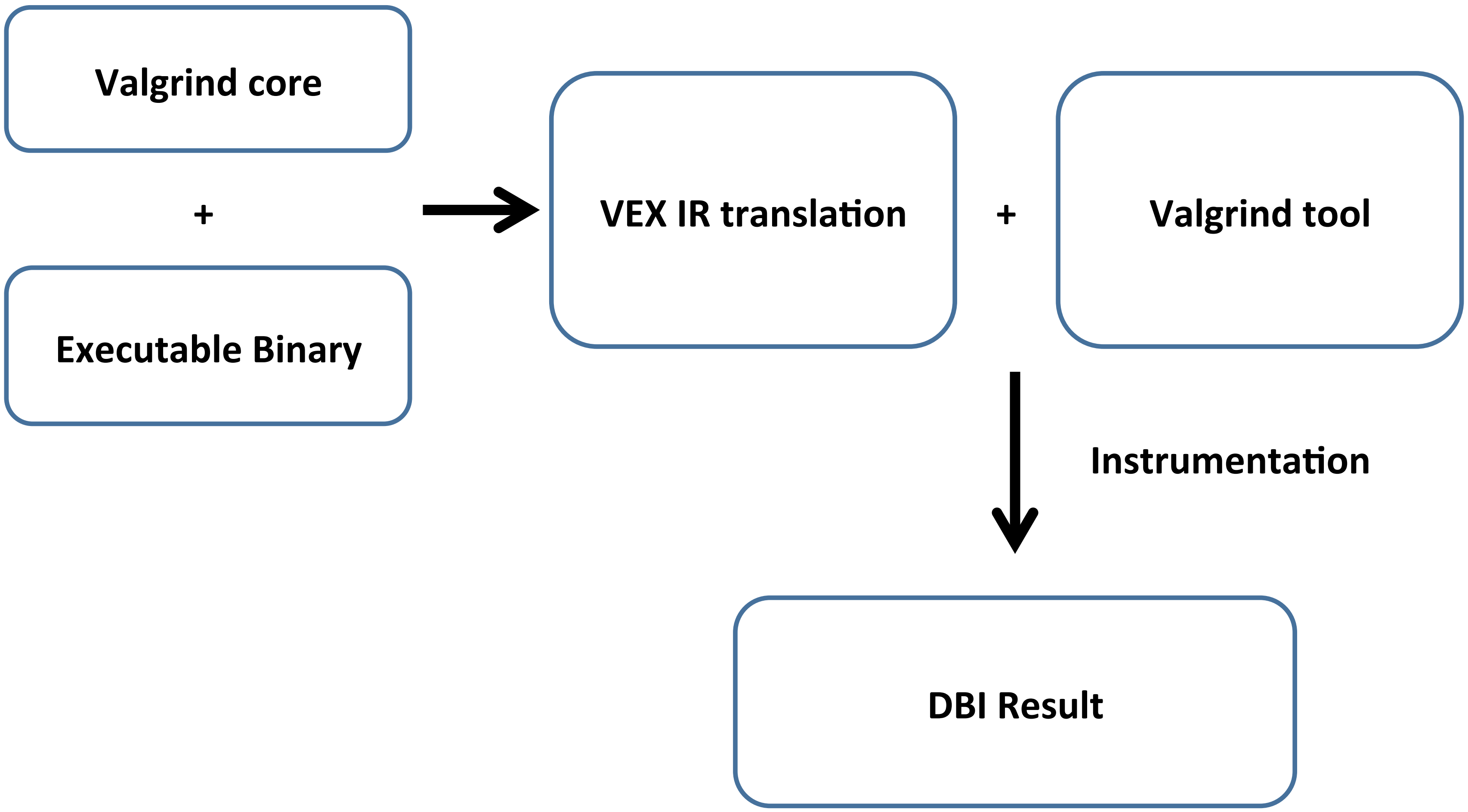
VEX IR(Intermediate Representation)

- Valgrind's binary translation mechanism
- VEX IR: machine independent intermediate representation
- Translates a block of binary codes to simplified VEX representation

VEX IR : Example

- addl %eax, %ebx :
  - t3 = GET:I32(0) # get %eax, a 32-bit integer
  - t2 = GET:I32(12) # get %ebx, a 32-bit integer
  - t1 = Add32(t3,t2) # addl
  - PUT(0) = t1 # put %eax

Valgrind : Overview



Attack Detection Using Valgrind DBI Framework

CTFGRIND

What does it do?

- match registered execution patterns
- checks sensitive memory area overwriting
- marks execution flow using IDA Plug-in

Pattern 1: RET overwriting

- We can get the guest machine’s register values
- We should protect our RET and stored EBP

1. Monitor every memory operation (Store)
2. Compare target address with $EBP
3. Output callstack

Pattern 2: GOT overwriting

- We can do in the same manner, because the address of GOT is static in a binary

Pattern 3: Strcpy

- What if a bug comes from using library function such as strcpy

1. We can compare the RET before the library function call and after the call
2. There could be many vulnerable library functions such as memcpy, strcpy, and scanf

Possible usage #1

- Attach directly to running daemon
- Prevent attack before exploitation
- Stop the process when a danger is detected
- <u>Possible slow down</u>

Possible usage #2

- Runs on a separated shadow machine
- When it detects attack, register the packet pattern to firewall to prevent further attack
- Can’t defend the first attack

IDA Plugin

- CTFGRIND logs the call stack when the attack detected
- IDA Plugin reads the file and marks the execution path
- Helpful to analyze other teams' exploit

DEMO

REFERENCE

- Emulation-
  based Security Testing for Formal Verification (Bl
  ack Hat Europe 2009) – Bruno Luiz
- Optimizing binary code produced by Valgrind – Luis Veiga
- Valgrind – Mario Sanchez, Cecilia Gonzalez
- Hacking using Binary instrumentation – Gal Diskin
- Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation -
  Nicholas Nethercote, Juliam Seward
- Valgrind Technical Manual

감사합니다

hoon0612 @postech.ac.kr
jinmel@postech.ac.kr