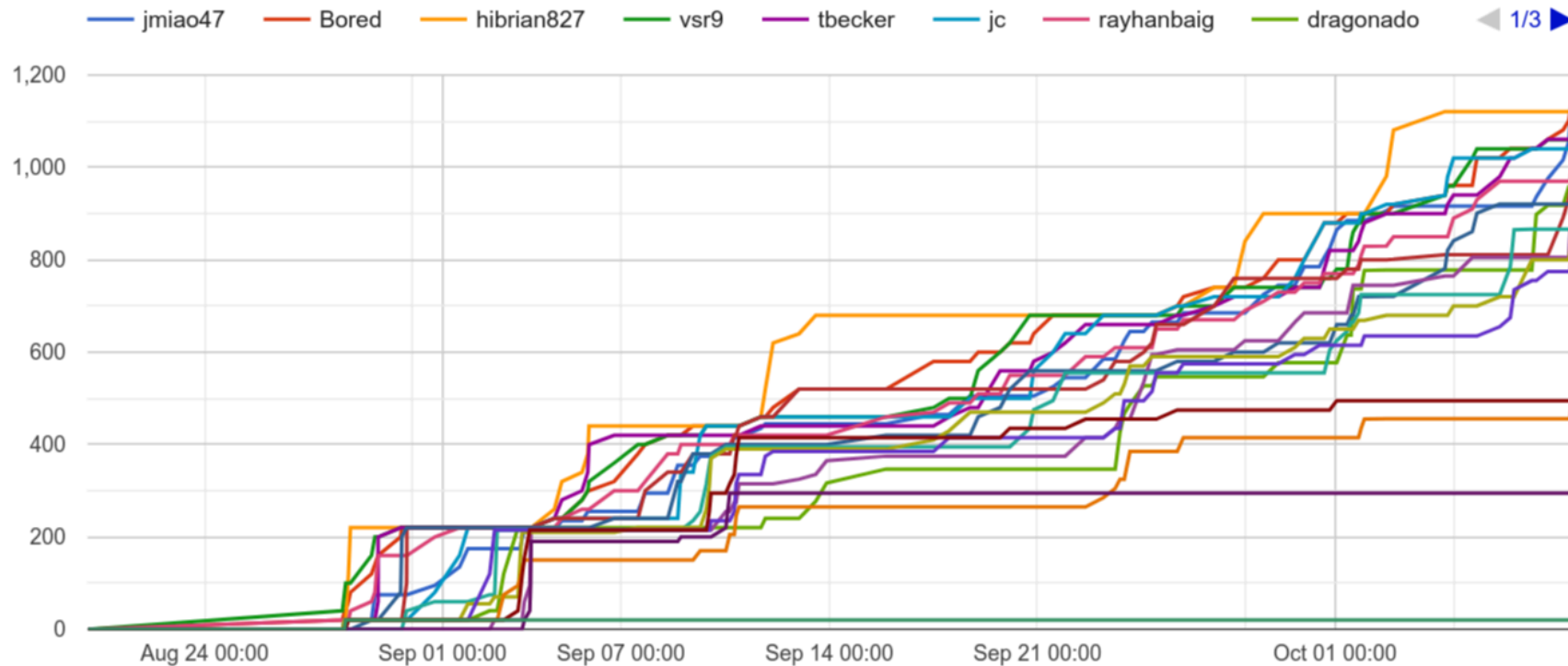


Lec07: Return-oriented Programming

Taesoo Kim

Scoreboard

2



Administrivia

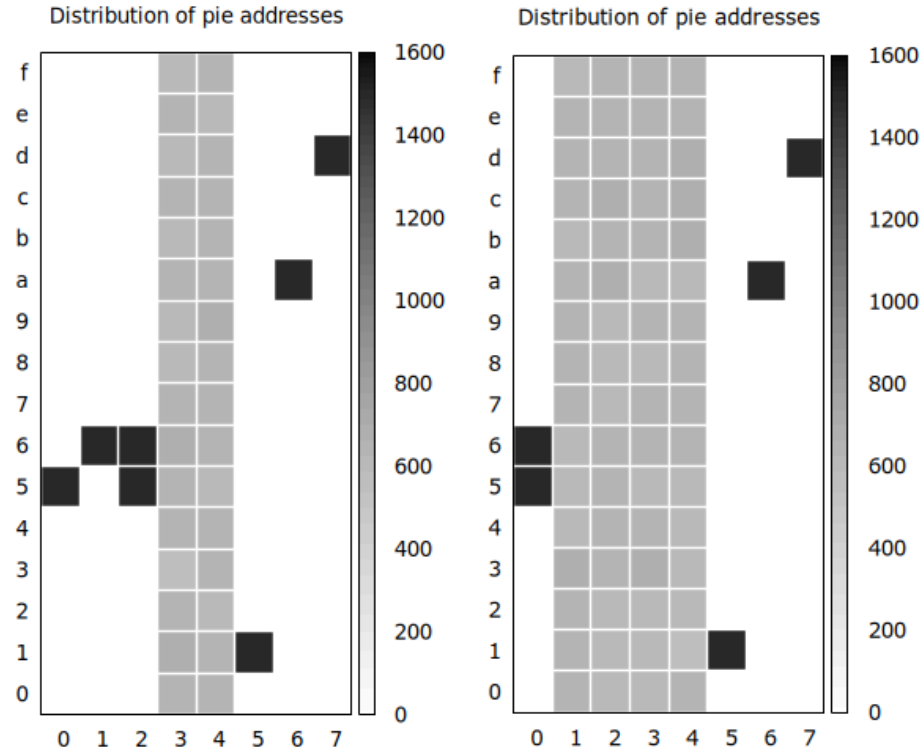
- Congrats! Just completed 50% of labs!
- Due: Lab06 is out and its due on Oct 22 (two weeks)!
- Lab10: [NSA Codebreaker Challenge](#) → Due: Dec 9
- In-class CTF (Dec 4): Please find your team mates (3-4 people)!

Discussion: moving-target

- What's `check-aslr.sh` and `pie.c`?
- How many times should we try to exploit?

Lack of entropy is a fundamental limitation (e.g., 32-bit x86).

Discussion: moving-target



→ How many bits of entropy?

Discussion: fmtstr-*?

- fmtstr-read/write/digging are relatively easy

But, How to Prevent fmtstr-*?

But, How to Prevent fmtstr-*?

1. Relaxing POSIX compliance (e.g., Windows)
 - Discarding `%n` - Q1. Why?
 - Limiting width (e.g., `%.512x` in XP, `%.622496x` in 2000) - Q2. Why?
2. **Dynamic**: enabling `FORTIFY` in gcc (e.g., Ubuntu)
3. **Static**: code annotation (e.g., Linux)

FORTIFY (-D_FORTIFY_SOURCE=2)

- Ensuring that all positional arguments are used
 - e.g., `%2$d` is not ok without `%1$d` - Q3. Why?
- Ensuring that `fmtstr` is in the read-only region (when `%n`)
 - e.g., `%n` should not be in a writable region - Q4. Why?

```
$ ./fortify-yes %2$d
*** invalid %N$ use detected ***
```

```
$ ./fortify-yes %n
*** %n in writable segment detected ***
```

Discussion: mini-sudo (CVE-2012-0059)

10

- What is `-D9` for?

Discussion: mini-sudo (CVE-2012-0199)

11

```
1 void sudo_debug(int level, const char *fmt, ...) {
2     va_list ap;
3     char *fmt2;
4
5     if (level > debug_level) return;
6
7     /* Bucket fmt with program name and a newline to make it
8        a single write */
9     easprintf(&fmt2, "%s: %s\n", getprogname(), fmt);
10    va_start(ap, fmt);
11    vfprintf(stderr, fmt2, ap);
12    va_end(ap);
13    efree(fmt2);
14 }
```

CVE-2013-1848: Linux ext3

```
1 void ext3_msg(struct super_block *sb, const char *prefix,  
2             const char *fmt, ...)  
3 {  
4     struct va_format vaf;  
5     va_list args;  
6  
7     va_start(args, fmt);  
8  
9     vaf.fmt = fmt;  
10    vaf.va = &args;  
11  
12    printk("%sEXT3-fs (%s): %pV\n", prefix, sb->s_id, &vaf);  
13  
14    va_end(args);  
15 }
```

CVE-2013-1848: Linux ext3

- Intended/correct usages:

```
void ext3_msg(struct super_block *sb, const char *prefix,  
             const char *fmt, ...);
```

```
ext3_msg(sb, KERN_ERR, "Invalid uid value %d", option);
```

CVE-2013-1848: Linux ext3

- What's wrong? and why so fascinating?

```
1 // @get_sb_block()
2 ext3_msg(sb, "error: invalid sb specification: %s", *data);
3
4 // @ext3_blkdev_get()
5 ext3_msg(sb, "error: failed to open journal device %s: %ld",
6          __bdevname(dev, b), PTR_ERR(bdev));
```

CVE-2013-1848: Linux ext3

- Annotating fmtstr for automatic checking at compilation time

```
1  extern __printf(3, 4)
2  void ext3_msg(struct super_block *sb, const char *prefix,
3              const char *fmt, ...);
4
5  // Compilation ERROR!
6  ext3_msg(sb, "error: invalid sb specification: %s", *data);
7  ext3_msg(sb, "error: failed to open journal device %s: %ld",
8            __bdevname(dev, b), PTR_ERR(bdev));
```

→ The compiler checks if the arguments are correctly matched w/ fmt specifiers

Summary of Lab05

- **Fundamental limitations** of DEP/ASLR (e.g., lack of entropy, info leaks)
 - brainfxxk: leaking code pointer
 - logic error: incorrectly using fd
 - profile: uninitialized use + leaking code pointers
- Powerful **format string vulnerability**
 - fmtstr-heap: data pointers not in the input buffer
 - mini-sudo: overwriting local variables

Take-outs from DEP/ASLR?

- Are DEP/ASLR bullet-proof defenses? No!
 - Do you think DEP/ASLR make attackers' life more difficult? Yes!
- Although we can't place shellcode into stack/heap, we can **still** hijack the control flow of a program in many interesting ways

Modern Exploit against DEP/ASLR

1. Leak code pointers

- i.e., decoding the memory layout of a library or program

2. Construct Return-Oriented Programming (**ROP!**)

- i.e., arbitrary code execution

→ Need 1+ more bugs that allow leaks and control-flow hijack!

Today's Tutorial

- In-class tutorial:
 - Ret-to-libc
 - Code pointer leakage / gadget finding
 - First ROP in x86!

Reminder: crackme0x00 (again!)

```
1 void start() {
2     printf("IOLI Crackme Level 0x00\n");
3     printf("Password:");
4
5     char buf[32];
6     memset(buf, 0, sizeof(buf));
7     read(0, buf, 256);
8
9     if (!strcmp(buf, "250382"))
10         printf("Password OK :)\n");
11     else
12         printf("Invalid Password!\n");
13 }
```

Reminder: crackme0x00

```
$ checksec ./target
[*] '/home/lab/tut-rop/target'
Arch:      i386-32-little
RELRO:     Partial RELRO      <- (later)
Stack:     No canary found    <- Hey!
NX:        NX enabled        <- No shellcode in data sections
PIE:       No PIE (0x8048000) <- The code section is not randomized
```

Reminder: crackme0x00

```
1  int main(int argc, char *argv[]) {
2      setvbuf(stdout, NULL, _IONBF, 0);
3      setvbuf(stdin, NULL, _IONBF, 0);
4
5      void *self = dlopen(NULL, RTLD_NOW);
6      printf("stack    : %p\n", &argc);
7      printf("system(): %p\n", dlsym(self, "system"));
8      printf("printf(): %p\n", dlsym(self, "printf"));
9
10     start();
11
12     return 0;
13 }
```

→ Assumed you first leaked these code pointers! (In fact, not needed)!

Ret-to-libc: printf

```
printf("Password OK:");
```

```
[buf  ]
```

```
[.....]
```

```
[ra   ] => printf
```

```
[dummy]
```

```
[arg1 ] => "Password OK :)"
```

Ret-to-libc: system

```
system("/bin/sh");
```

```
[buf  ]
```

```
[.....]
```

```
[ra   ] => system
```

```
[dummy]
```

```
[arg1 ] => "/bin/sh"
```

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

> ret (after stack smashing)

```
      [buf      ]  
      [..... ]  
esp -> [ra      ] => (1) printf  
      [dummy   ]  
      [arg1    ] => (1) "Password OK :)"  
      [        ]
```

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

> push ebp (in printf())

```
    [buf      ]  
    [.....  ]  
    [         ]  
esp -> [dummy  ]  
    [arg1     ] => (1) "Password OK :)"  
    [         ]
```

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

> ret (in printf())

```
      [buf      ]  
      [..... ]  
      [         ]  
esp -> [dummy   ]  
      [arg1     ] => (1) "Password OK :)"  
      [         ]
```

→ It takes `dummy` as an instruction pointer! so let's `chain` it.

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

> ret (in printf())

```
      [buf      ]  
      [..... ]  
      [         ]  
esp -> [ret      ] -> (2) system  
      [arg1     ] => (1) "Password OK :)"  
      [         ]
```

→ Where to put system()'s argument?

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

> ret (in printf())

```
      [buf      ]  
      [..... ]  
      [         ]  
esp -> [ret      ] -> (2) system  
      [old-arg1 ] => (1) "Password OK :)"  
      [arg1     ] -> (2) "bin/sh"
```

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

```
[buf      ]  
[..... ]  
[(1)ret   ] => (1) printf  
[(2)ret   ] -> (2) system  
[(1)arg1  ] => (1) "Password OK :)"  
[(2)arg1  ] -> (2) "bin/sh"
```

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

```
[buf      ]  
[..... ]  
[(1)ret   ] => (1) printf  
[(2)ret   ] -> (2) system  
[(1)arg1  ] => (1) "Password OK :)"  
[(2)arg1  ] -> (2) "bin/sh"
```

→ What happens when `system()` returns?

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

> push ebp (in system())

```
    [buf      ]  
    [..... ]  
    [(1)ret   ] => (1) printf  
    [(2)ret   ] -> (2) system  
esp -> [(1)arg1] => (1) "Password OK :)"  
      [(2)arg1] -> (2) "bin/sh"
```

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

```
> ret (in system())
```

```
    [buf      ]  
    [..... ]  
    [(1)ret   ] => (1) printf  
    [(2)ret   ] -> (2) system  
esp -> [(1)arg1] => (1) "Password OK :)"  
      [(2)arg1] -> (2) "bin/sh"
```

→ Segmentation fault at the address of "Password OK :)!"

Chaining Two Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

```
> ret (in system())
```

```
    [buf      ]  
    [..... ]  
    [(1)ret   ] => (1) printf  
    [(2)ret   ] -> (2) system  
esp -> [(1)arg1] => (1) "Password OK :)"  
      [(2)arg1] -> (2) "bin/sh"
```

→ How would you chain more than two calls?

Chaining N Function Calls

```
printf("Password OK:");  
system("/bin/sh");
```

```
> ret (in printf())
```

```
      [buf      ]  
      [..... ]  
      [ret      ] => (1) printf  
esp -> [dummy   ]  
      [arg1     ] => (1) "Password OK :)"  
      [         ]
```

→ **Trick**: Clean up the stack first instead of calling `system()`, making it repeatable!

Chaining N Function Calls

```
printf("Password OK:");
system("/bin/sh");
```

> ret (in printf())

	[buf	]	
	[.....	]	
	[ret	] => (1) printf	
	[dummy	]	
^	[arg1	] => (1) "Password OK :)"	^
+-----+ (first call)			
esp ->	[ret	] => (2) system	
	[dummy	]	
	[arg1	] => (2) "/bin/sh"	

→ How to make the stack as if it's first call?

Chaining N Function Calls

```
printf("Password OK:");
system("/bin/sh");
```

> ret (in printf())

	[buf	]	
	[.....	]	
	[ret	]	=> (1) printf
esp ->	[gadget	]	-----> pop/ret!
	[arg1	]	=> (1) "Password OK :)" ^

	[ret	]	=> (2) system
	[dummy	]	
	[arg1	]	=> (2) "/bin/sh"

→ A code snippet that contains pop/ret (e.g., `pop ebp; ret`)

Chaining N Function Calls

```
printf("Password OK:");
system("/bin/sh");
```

> pop ebp (in a pop/ret gadget)

	[buf	]	
	[.....	]	
	[ret	]	=> (1) printf
	[gadget	]	-----> pop/ret!
esp ->	[arg1	]	=> (1) "Password OK :)" ^

	[ret	]	=> (2) system
	[dummy	]	
	[arg1	]	=> (2) "/bin/sh"

→ pop will consume one more slot in the stack.

Chaining N Function Calls

```
printf("Password OK:");
system("/bin/sh");
```

> ret (in a pop/ret gadget)

	[buf	]	
	[.....	]	
	[ret	]	=> (1) printf
	[gadget	]	-----> pop/ret!
	[arg1	]	=> (1) "Password OK :)" ^

esp ->	[ret	]	=> (2) system
	[dummy	]	
	[arg1	]	=> (2) "/bin/sh"

→ system() will be invoked as if it's a first call

Chaining N Function Calls

```
printf("Password OK:");
system("/bin/sh");
exit(0);
```

```
[buf      ]
[.....   ]
[ret      ] => (1) printf
[gadget   ] -----> pop/ret!
[arg1     ] => (1) "Password OK :)" ^
```

```
-----
[ret      ] => (2) system
[gadget   ] -----> pop/ret!
[arg1     ] => (2) "/bin/sh"
```

```
-----
[ret      ] => (3) exit
[gadget   ] -----> pop/ret!
[arg1     ] => (3) 0
```

Calling Functions with 1+ Argument?

```
open("/proc/flag", 0_RDONLY)
```

```
...
```

```
[buf      ]
[.....   ]
[ret      ] => (1) open
[gadget   ] -----> pop/pop/ret!
[arg1     ] => (1) "/proc/flag"
[arg2     ] => (1) 0_RDONLY
```

```
-----
```

```
[ret      ]
[gadget   ]
[arg1     ]
```

```
...
```

→ `pop/pop/ret` will clean up two slots instead!

Today's Tutorial: Chaining Three Calls

```
open("/proc/flag", O_RDONLY)  
read(3, tmp, 1024)  
write(1, tmp, 1024)
```

In-class Tutorial

- Step1: Ret-to-libc
- Step2: Understanding module base
- Step3: First ROP

```
$ ssh lab06@54.88.195.85  
Password: <password>
```

```
$ cd tut07-rop  
$ cat README
```

References

- ROP
- The advanced return-into-lib(c) exploits