

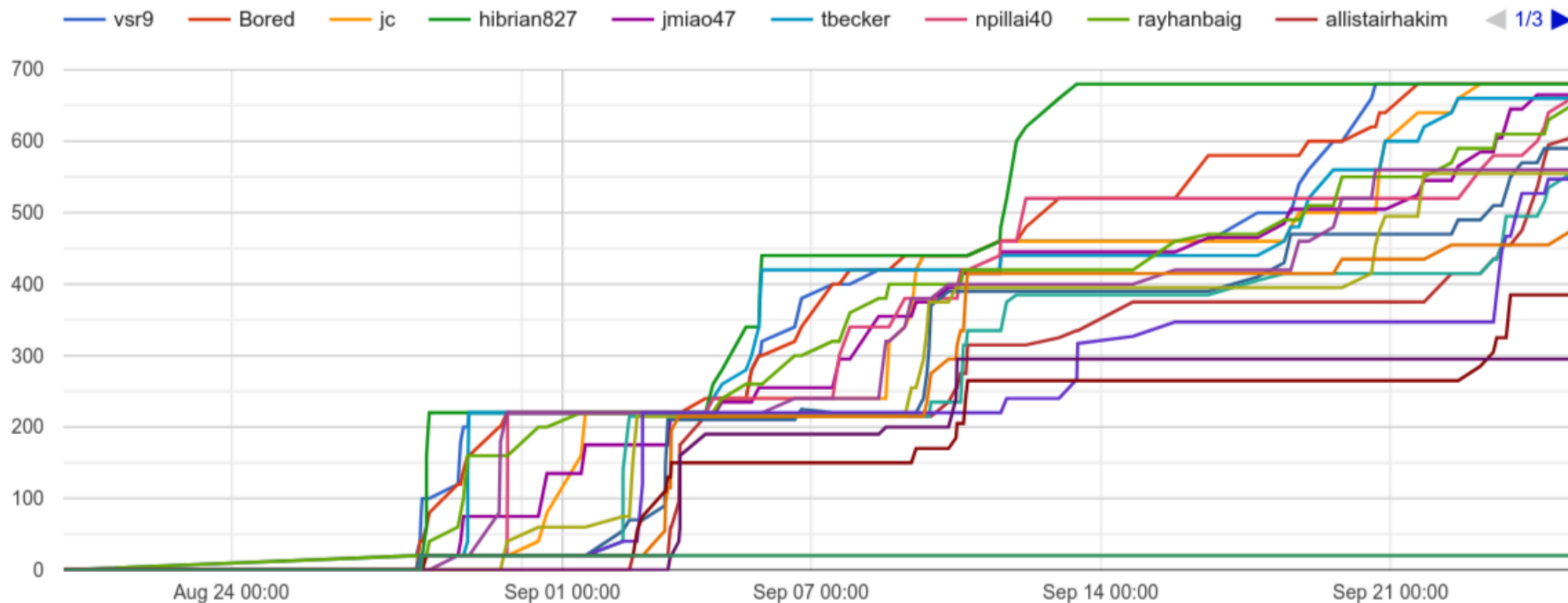
# Lec05: Stack Protections

*Taesoo Kim*

# Administrivia

- Please submit your write-ups on time!
- Please write down your collaborators' names on the write-ups
- Due: Lab04 is out, and its due on Oct 4 at midnight

# Lab03: Stack Overflow



# Discussion: Not Yet Motivated?

## ENDPOINT & ON-PREM PROGRAMS

| Program                                                        | Description                                                                                                         | Award Range         |
|----------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|---------------------|
| <a href="#">Microsoft Hyper-V</a>                              | Critical remote code execution, information disclosure and denial of services vulnerabilities in Hyper-V            | Up to \$250,000 USD |
| <a href="#">Microsoft Windows Insider Preview</a>              | Critical and important vulnerabilities in Windows Insider Preview                                                   | Up to \$100,000 USD |
| <a href="#">Microsoft Applications and On-Premises Servers</a> | Critical and important vulnerabilities in Microsoft Applications and On-Premises Servers                            | Up to \$30,000 USD  |
| <a href="#">Microsoft Edge (Chromium-based)</a>                | Critical, important, and moderate vulnerabilities in Microsoft Edge (Chromium-based) Dev, Beta, and Stable channels | Up to \$30,000 USD  |
| <a href="#">Microsoft 365 Insider</a>                          | Vulnerabilities on Microsoft 365 Insider                                                                            | Up to \$15,000 USD  |

## Application tier 1 rewards

| Category                    | 1) Remote/No User Interaction | 2) User must follow a link that exploits the vulnerable app | 3) User must install malicious app or victim app is configured in a non-default way | 4) Attacker must be on the same network (e.g. MITM) |
|-----------------------------|-------------------------------|-------------------------------------------------------------|-------------------------------------------------------------------------------------|-----------------------------------------------------|
| A) Arbitrary Code Execution | \$300,000                     | \$150,000                                                   | \$15,000                                                                            | \$9,000                                             |
| B) Theft of Sensitive Data* | \$75,000                      | \$37,500                                                    | \$9,000                                                                             | \$6,000                                             |
| C) Other Vulnerabilities    | \$24,000                      | \$9,000                                                     | \$4,500                                                                             | \$2,400                                             |

| Entry Point                                                                | End Point                     | Maximum Reward | <a href="#">View Examples</a> |
|----------------------------------------------------------------------------|-------------------------------|----------------|-------------------------------|
| Network attack with no user interaction                                    | Kernel                        | \$2,000,000    | ▼                             |
|                                                                            | User space                    | \$350,000      | ▼                             |
| Network attack with user interaction                                       | Kernel                        | \$1,000,000    | ▼                             |
| Wireless proximity-based attacks targeting Apple-designed radio components | Application processor         | \$1,000,000    | ▼                             |
| Unauthorized physical device access                                        | Access to sensitive user data | \$500,000      | ▼                             |
| App sandbox escape                                                         | Kernel                        | \$500,000      | ▼                             |
|                                                                            | Access to sensitive user data | \$100,000      | ▼                             |
| Browser attack                                                             | Kernel                        | \$1,000,000    | ▼                             |
|                                                                            | WebContent sandbox escape     | \$300,000      | ▼                             |
|                                                                            | Web Content code execution    | \$10,000       | ▼                             |

# Discussion: Not Yet Motivated?

## Evolving the Android & Chrome VRPs for the AI Era

The primary goal of the Android & Google Devices and Chrome Vulnerability Reward Programs (VRPs) has been to partner with the external security researcher community to analyze areas lacking coverage, helping us discover vulnerabilities that historically have been hard to find. In today's shifting security research landscape, we want to continue to reward researchers for their expertise and creativity in finding the most challenging and impactful vulnerabilities in our products.

Over the past few years, AI and automation have accelerated the pace of vulnerability discovery, and our teams are moving at an unprecedented rate – remediating risks more effectively than ever before. The latest advancements in AI from Google and the broader industry have made it significantly easier to take a test case and explain the root cause, propose a suitable fix, and to find variants of known problems. And to keep pace with vulnerability discovery, we've been continuing to implement structural improvements in our products to make it increasingly difficult to achieve full chain exploits. For example, Android recently introduced Advanced Protection Mode and continues to drive adoption of [memory safe languages](#). Chrome has driven structural improvements to the V8 sandbox, container hardening, and in-browser memory quarantining. By pairing automated tools like [Big Sleep](#), [CodeMender](#), and [OSS-Fuzz service](#) with the human expertise from teams like [Project Zero](#), [Bug Hunters](#), dedicated product security teams, and the [Google Threat Intelligence Group](#), we have a multi-layered approach to addressing security issues across Google products and other non-Google platforms to keep users safe online.

As the security research landscape evolves with AI, we're making changes in our programs to ensure we're rewarding the most challenging and impactful vulnerabilities in our products. This focus provides the most value to our security teams and helps keep users safe today, all while making sure security researchers continue to be rewarded for their efforts.

## Top-Tier Rewards for Hard Problems

We know that certain particularly impactful exploits remain incredibly difficult to achieve and we've greatly appreciated collaborating with the researcher community to discover and unearth them. We want to build on this partnership by continuing to emphasize the highest tiers of rewards across both Android and Chrome.

Android:

- Up to **\$1,500,000** for a zero-click full chain Pixel Titan M2 compromise with persistence.
- Up to **\$750,000** for a zero-click full chain Pixel Titan M2 compromise.

Chrome:

- Up to **\$250,000** for full chain browser process exploits on the latest operating systems and hardware.
- Up to **\$250,128** bonus for a report that successfully exploits an allocation we believe to be protected by MiraclePtr.

# Discussion: jmp-to-where

- What's the bug?
- What's special about this challenge?

```
(x86-32)
0x08048000
| code    .text section
| data    .data section
| bss     .bss section
| ...
| stack   stack
| args    program args
| envs    program envs
| prog    filename of program
| null    final dword of zero
0xBFFFFFFF
```

# Discussion: jmp-to-where

```
(stack)
| argc
| argv[0]
| ...
| argv[argc-1]
| NULL
| env[0]
| ...
| env[n]
| NULL
| ...
| [/your-program-name] <= is this something you can control?
```

→ A user-controlled input can be used in a surprising way!

# Discussion: unusual-main

- What's the bug?
- What's special about this challenge?

# Discussion: unusual-main

- vuln()

```
0x080484d5 <+63>:    mov     ebx,DWORD PTR [ebp-0x4]
==> 0x080484d8 <+66>:    leave
0x080484d9 <+67>:    ret
```

→ Stack pivoting: control **eip** by modifying **ebp**

# Discussion: man-strncpy

- What's the bug?
- What's special about this challenge?

# Discussion: man-strncpy

```
1  char *
2  strncpy(char *dest, const char *src, size_t n) {
3      size_t i;
4      for (i = 0; i < n && src[i] != '\0'; i++)
5          dest[i] = src[i];
6      for ( ; i < n; i++)
7          dest[i] = '\0';
8      return dest;
9  }
```

- NULL on the remaining bytes
- Corrupting the lowest byte of `ebp`
- And, any other problems?

# Discussion: man-strncpy

- What's your lesson?
  - API misuses introduce subtle bugs (but mostly designed for [reasons](#))
  - By the way, check out a better language like [Rust](#)
- How to prevent this?

# Discussion: man-strncpy

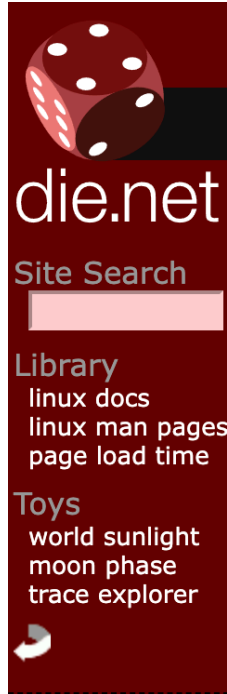
```
1      char buf[BUFSIZ];
2      strncpy(buf, input, ???);
3
4      1. BUFSIZ    (sizeof(buf))?
5      2. BUFSIZ-1 (sizeof(buf)-1)?
6
7  NOTE.
8      snprintf(buf, sizeof(buf), "...")
```

# Discussion: man-strncpy (safe usage)

```
1 | char buf[BUFSIZ];  
2 | strncpy(buf, input, sizeof(buf) - 1);  
3 | buf[sizeof(buf) - 1] = '\\0';
```

# Discussion: alternative strncpy()

```
1 | strncpy(buf, s, sizeof(buf));
```



## strncpy(3) - Linux man page

### Name

**strncpy, strlcat** - size-bounded string copying and concatenation

### Library

Utility functions from BSD systems (libbsd, -lbsd)

### Synopsis

**#include <bsd/string.h>**

*size\_t*

**strncpy**(*char \*dst, const char \*src, size\_t size*);

# Discussion: upside-down

| 0x00000000 |    | 0x00000000 |    |
|------------|----|------------|----|
| ^          |    | arg1       |    |
| buf        |    | ret        |    |
| [ ]        |    | fp         |    |
| [ ]        |    | buf        | ?? |
| fp         |    | [ ]        |    |
| ret        | vv | [ ]        |    |
| arg1       |    |            | vv |
| ..         |    | v          |    |
| 0xFFFFFFFF |    | 0xFFFFFFFF |    |

- Overwriting function arguments saved on the stack

# Discussion: upside-down

- More secure? less? in terms of security?

# How to Prevent Stack Overflow?

- Two approaches:
  - Bug prevention
  - Exploitation mitigation
- Protect “integrity” of ra, funcptr, etc (code pointers)
  - (e.g., exploitation mitigation → NX, canary)
- Prevent the buffer overflow at the first place
  - (e.g., code analysis, better APIs)

# Today's Tutorial

- In-class tutorial
  - Let's understand the implementation of the stack protector.
  - Let's exploit the (insecurely) protected crackme0x00 to get a flag!

# Reminder: crackme0x00

```
$ objdump -M intel -d crackme0x00
```

```
...
```

```
80485c4:  lea    eax, [ebp-0x1c]
```

```
80485c7:  push   eax
```

```
80485c8:  lea    eax, [ebx-0x191e]
```

```
80485ce:  push   eax
```

```
80485cf:  call   8048440 <__isoc99_scanf@plt>
```

```
|<=- 0x1c==>|+--- ebp
```

```
top
```

```
v
```

```
[ ~~~~> ] ][fp][ra]
```

# Reminder: Exploiting crackme0x00

```
top      |<=- 0x1c ==>|+--- ebp
          v
[        [~~~~>  ]    ][fp][ra]
AAAABBBB.....GGGGHHHH
```

# crackme0x00 in C

```
1  int main(int argc, char *argv[])
2  {
3      char buf[16];
4      printf("IOLI Crackme Level 0x00\n");
5      printf("Password:");
6
7      // Q. How to fix this?
8      scanf("%s", buf);
9
10     if (!strcmp(buf, "250382"))
11         printf("Password OK :)\n");
12     else
13         printf("Invalid Password!\n");
14     return 0;
15 }
```

# How to fix crackme0x00's bug?

```
// NOTE. 15 not 16  
scanf("%15s", buf);
```

```
// NOTE. GCC allocates memory for char *buf,  
// which requires a manual free()  
scanf("%ms", &buf);
```

# StackGuard ('1998) vs Phrack ('1996)

## **StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks\***

Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton,<sup>†</sup> Jonathan Walpole,  
Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang

*Department of Computer Science and Engineering  
Oregon Graduate Institute of Science & Technology*

immunix-request@cse.ogi.edu, <http://cse.ogi.edu/DISC/projects/immunix>

### **Abstract**

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attacks gained notoriety in 1988 as part of the Morris Worm incident on the Internet. While it is fairly simple to fix individual buffer overflow vulnerabilities, buffer

### **1 Introduction**

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attack gained notoriety in 1988 as part of the Morris Worm incident on the Internet [23]. Despite the fact that fixing individual buffer overflow vulnerabilities is fairly simple, buffer overflow attacks continue to this day, as reported in the SANS Network Security Digest:

# Core Idea of Stack Protector

- Use a “canary” value as an indicator of the integrity of fp/ra

```

top
|<=--- buf -----=>|+--- ebp
v
[[ [ [canary][fp][ra][ ....] .. ]]
X0X0X0 XXXX
(corrupted?)
```

# Where does Stack Canary come from?

- `gs`: thread local storage
  - Check value in pwndbg: `gsbase`
- Disarming stack canary by overwriting it

```
11fd:      55
11fe:      89 e5
1200:      56
1201:      53
1202:      83 ec 18
1205:      e8 f6 fe ff ff
120a:      81 c3 f6 2d 00 00
1210:      8b 45 0c
1213:      89 45 e0
1216:      65 a1 14 00 00 00
121c:      89 45 f4
```

```
push    ebp
mov     ebp, esp
push    esi
push    ebx
sub     esp, 0x18
call    1100 <__x86.get_pc_thunk.bx>
add     ebx, 0x2df6
mov     eax, DWORD PTR [ebp+0xc]
mov     DWORD PTR [ebp-0x20], eax
mov     eax, gs:0x14
mov     DWORD PTR [ebp-0xc], eax
```

# Subtle Design Choices for the Stack Canary

- Where to put? (e.g., right above ra? fp? local vars?)
- Which value should I use? (e.g., secrete? random? per exec? per func?)
- How to check its integrity? (e.g., xor? cmp?)
- What to do after you find corrupted? (e.g., crash? report?)

# DEMO: GCC's Stack Protector

- Check `Makefile`
- Check compilation options
- Run `diff.sh`

# In-class Tutorial

- Step 1: Understanding GCC's Stack Protector
- Step 2: Let's exploit 0xdeadbeef canary!

```
$ ssh lab04@52.201.10.159  
Password: xxxxxxxx
```

```
$ cd tut04-ssp  
$ cat README
```

# References

- [Bypassing StackShield](#)