Practical Return Oriented Programming

An Introduction to ROP Techniques

by Jeremy Heng (@nn_amon)

Introduction

- A gentle introduction to Return Oriented Programming (ROP)
- Goal: Understand a subset called Ret2PLT

Motivations

- Class Assignment: Classic Stack Overflow with No Protections
- Introduce a method to defeat protected binaries
- Focus on 32 bit Linux binaries

Pre-Requisites

- Thorough understanding of classic buffer overflow techniques
- C
- Python, GDB, Bash

Warning

- This will be a content-heavy, fast paced lesson
- Feel free to pause or follow along with the lesson plan at:

  https://github.com/nnamon/PracticalRet2Libc

Basic Exploitation Refresher

Recall Assignment 1

- Classic Stack Overflow
- No ASLR
- No NX/DEP
- No Stack Canaries
- Jump to Shellcode

```c
#define BUFSIZE 64

int idx  = 0;
int idx1 = 0;
int idx2 = 0;
size_t byte_read1 = 0;
size_t byte_read2 = 0;
char buf1[BUFSIZE + 1];
char buf2[BUFSIZE + 1];

void bof(FILE * fd1, FILE * fd2)
{
    int i, j;
    char buf[BUFSIZE];
    i = j = 1;

    byte_read1 = fread(buf1, 1, BUFSIZE, fd1);
    byte_read2 = fread(buf2, 1, BUFSIZE, fd2);
    buf1[BUFSIZE] = '\0';
    buf2[BUFSIZE] = '\0';

    if(byte_read1 != byte_read2)
    {
        printf("Reading different number of bytes from ./exploit1 and ./exploit2!\n");
        return;
    }

    for(idx = 0; idx < byte_read1 + byte_read2; idx++)
    {
        idx1 = (idx % 2) ? BUFSIZE : idx / 2;
        idx2 = (idx % 2) ? (idx - 1) / 2: BUFSIZE;

        buf[idx] = buf1[idx1] + buf2[idx2];
    }

    printf("i = 0x%x, j = 0x%x\n", i, j);
}
```

Let’s delve into the inner workings of why and how ‘Jump to Shellcode’ works...

But with a simpler program.

The Vulnerable Program

```c
#include <unistd.h>
#include <stdio.h>

void vuln() {
    char buffer[16];
    read(0, buffer, 100);
    puts(buffer);
}

int main() {
    vuln();
}
```

Disable All Compile Time Protections/ASLR

```
gcc -m32 -o vuln1-nocanary-execstack -fno-stack-protector -zexecstack vuln1.c
echo 0 | sudo tee /sys/proc/kernel/randomize_va_space
```

```
gcc -m32 -o vuln1-nocanary-execstack -fno-stack-protector -zexecstack vuln1.c
```

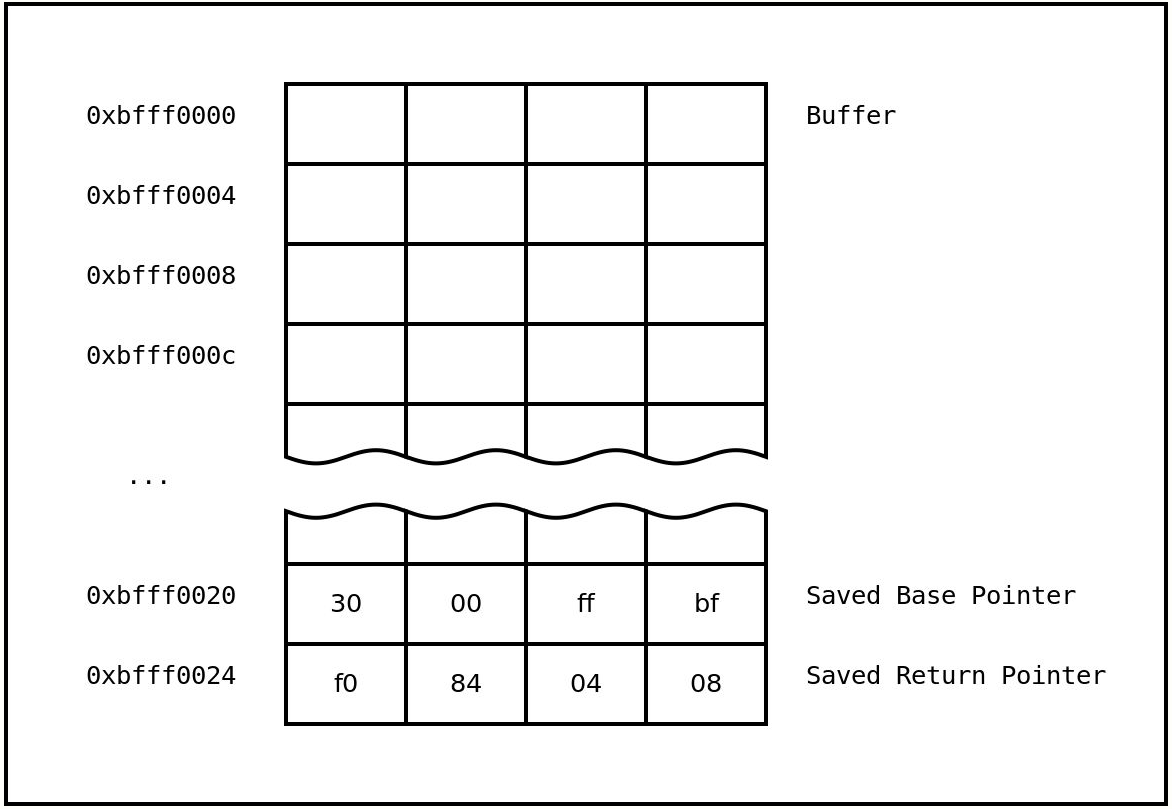
- -m32 - Compile as 32 bit
- -no-stack-protector - Disable stack canaries
- -zexecstack - Mark the stack and heap as executable

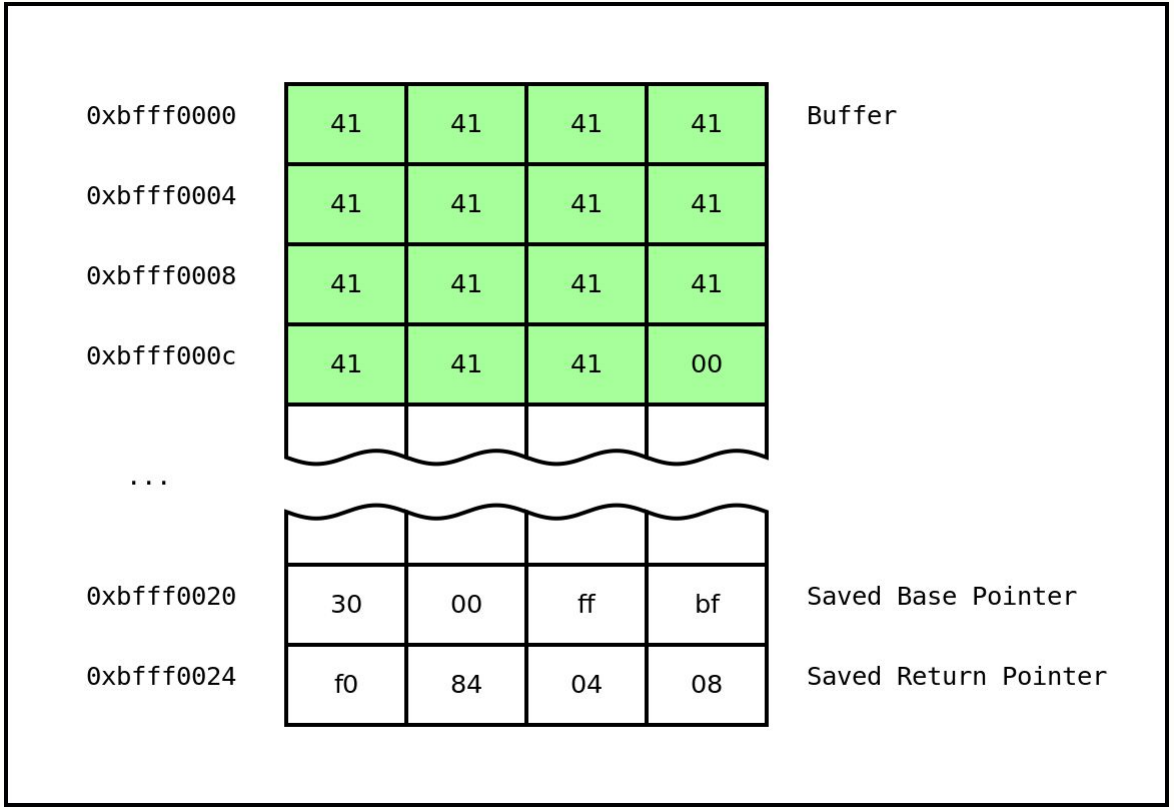
```
echo 0 | sudo tee /sys/proc/kernel/randomize_va_space
```

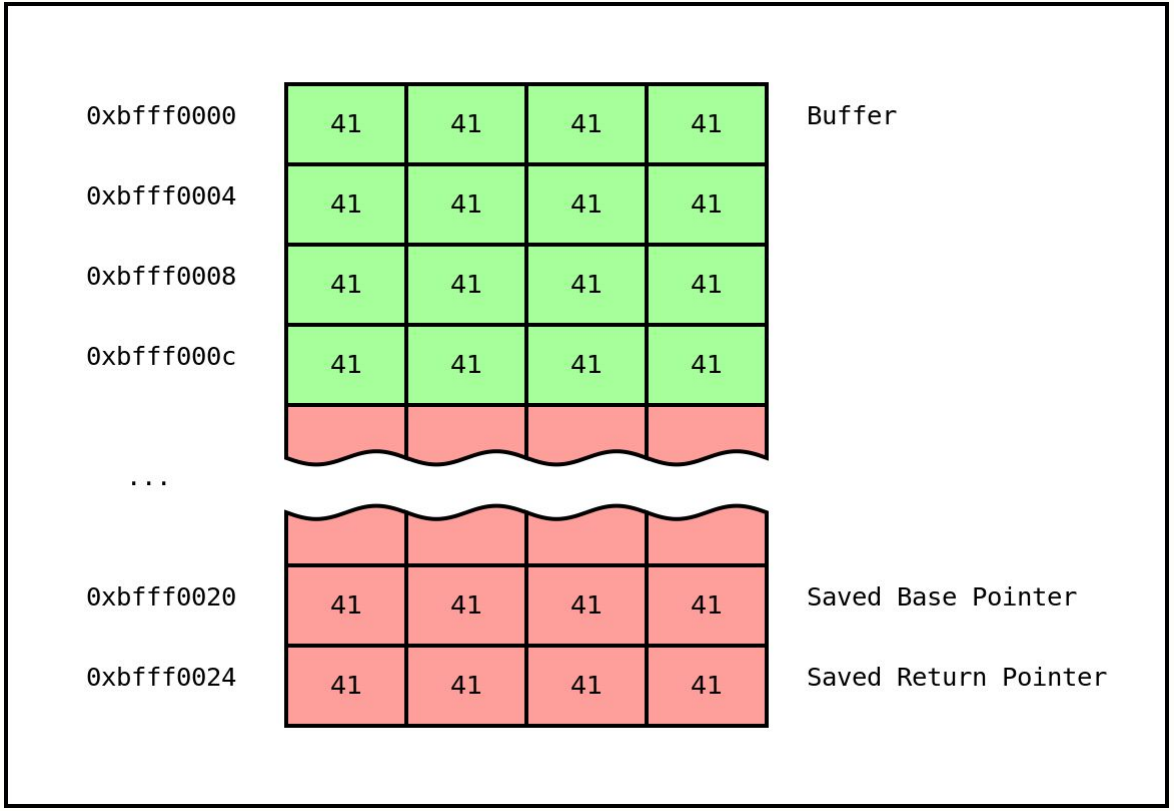
- Disable ASLR by setting the kernel parameter to 0

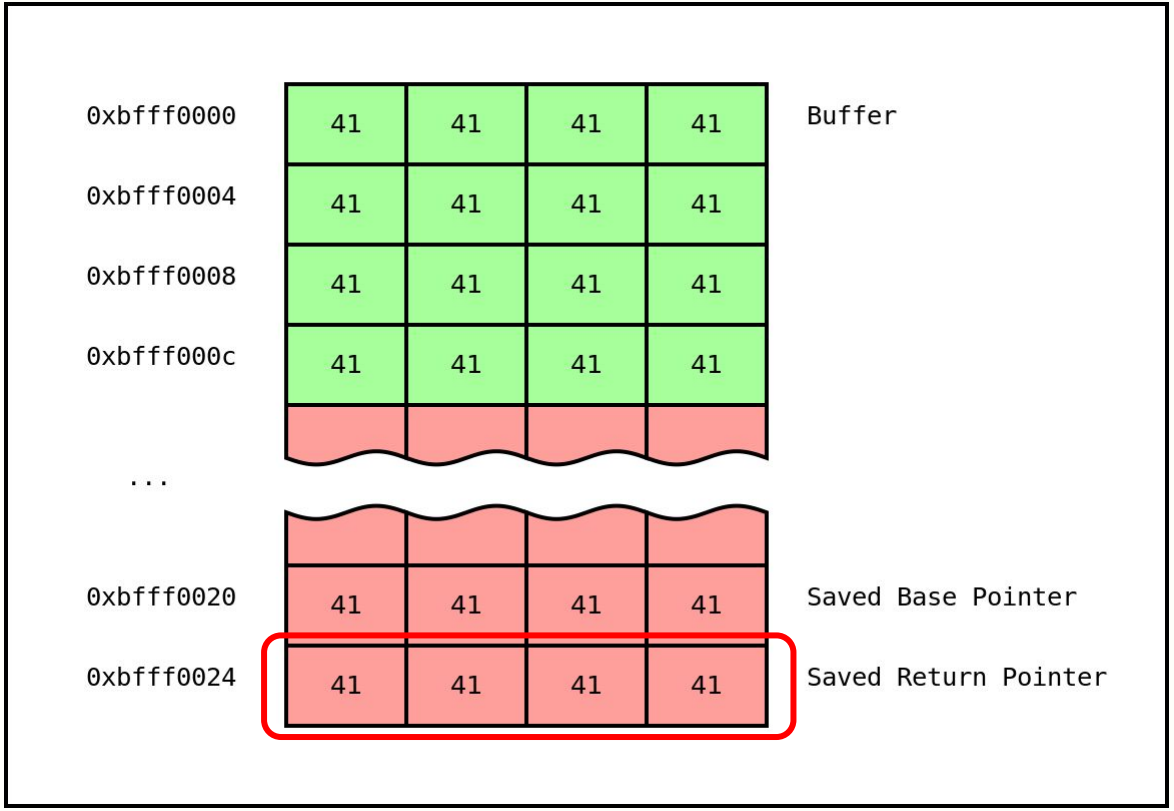
Let’s View the Memory Mappings

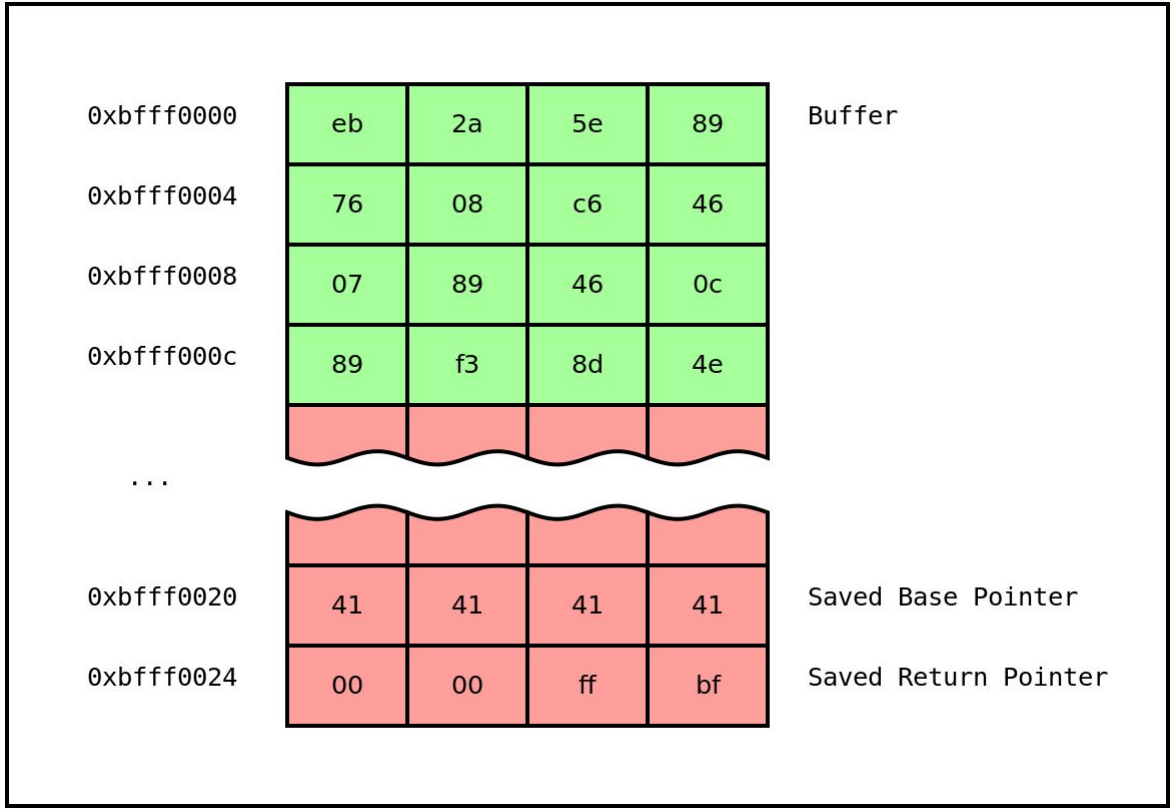
Classic Exploitation Illustration

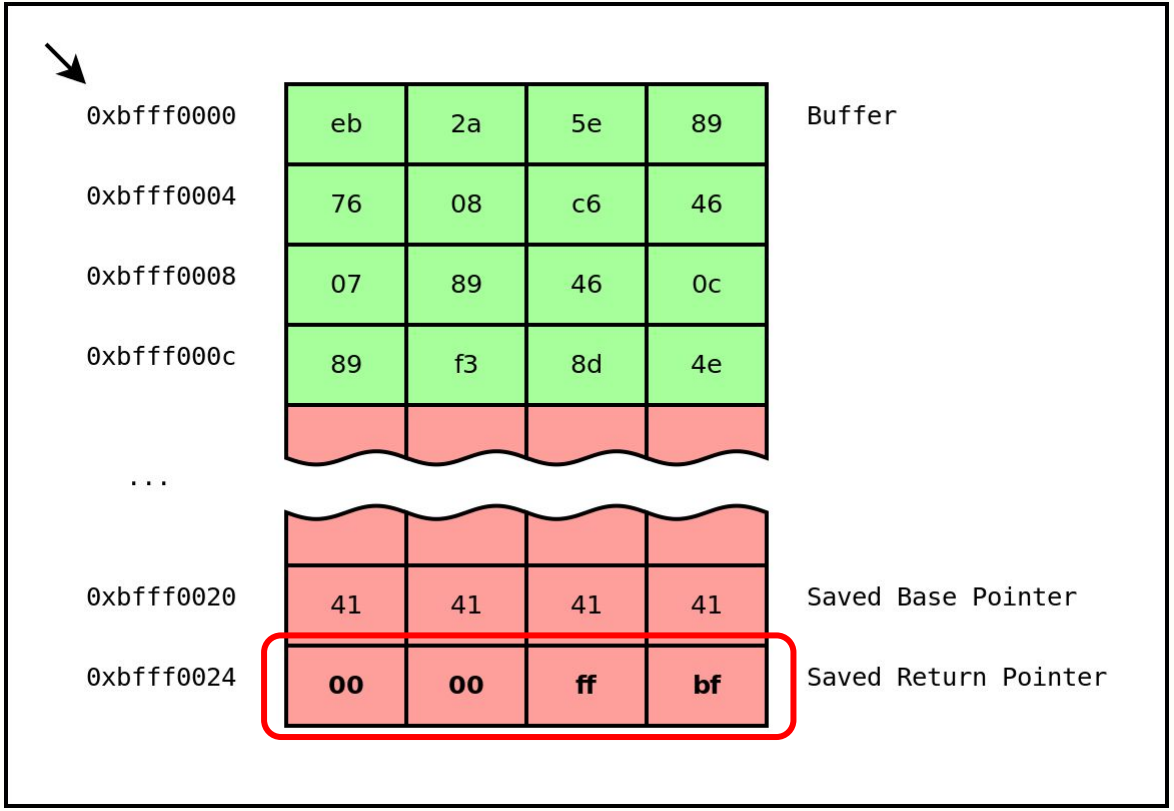

0xbfff0000
0xbfff0004
0xbfff0008
0xbfff000c
...
0xbfff0020
0xbfff0024
30
00
ff
bf
f0
84
04
08
Buffer
Saved Base Pointer
Saved Return Pointer

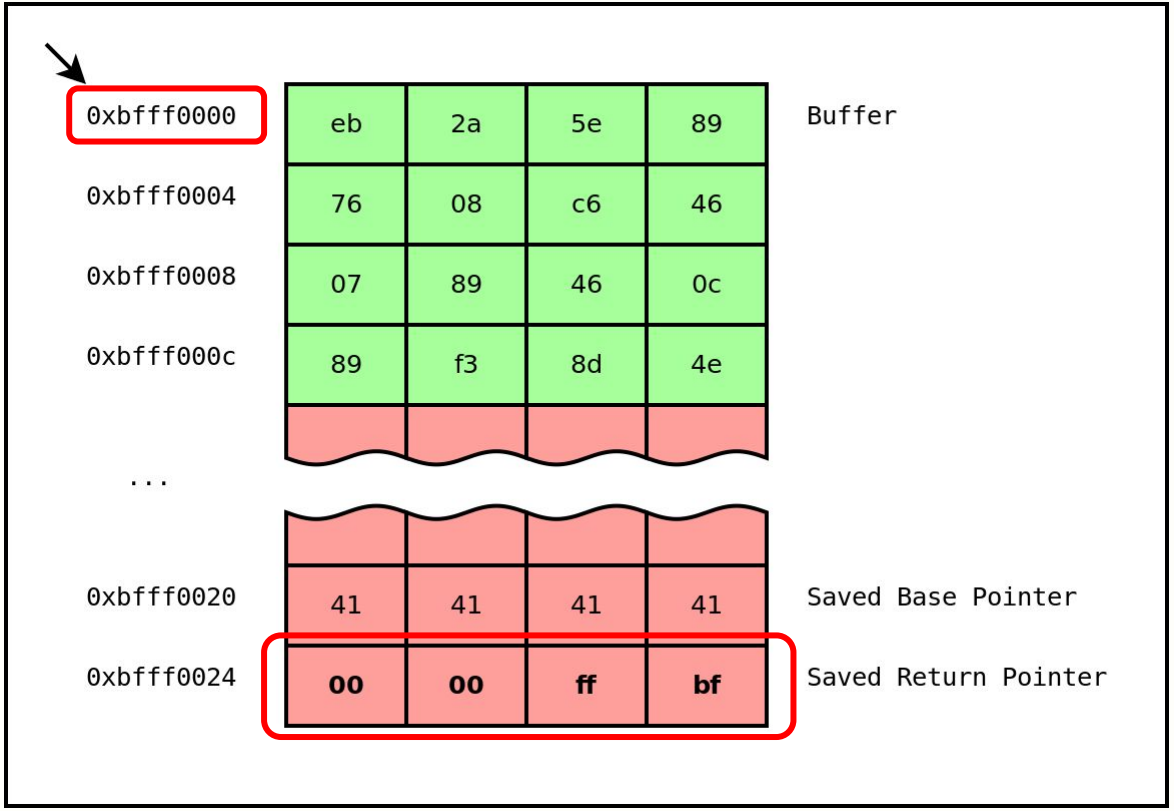

0xbfff0000
41 41 41 41
Buffer
0xbfff0004
41 41 41 41
0xbfff0008
41 41 41 41
0xbfff000c
41 41 41 00
...
0xbfff0020
30 00 ff bf
Saved Base Pointer
0xbfff0024
f0 84 04 08
Saved Return Pointer


0xbfff0000
41 41 41 41
Buffer
0xbfff0004
41 41 41 41
0xbfff0008
41 41 41 41
0xbfff000c
41 41 41 41
...
0xbfff0020
41 41 41 41
Saved Base Pointer
0xbfff0024
41 41 41 41
Saved Return Pointer


0xbfff0000
41 41 41 41
Buffer
0xbfff0004
41 41 41 41
0xbfff0008
41 41 41 41
0xbfff000c
41 41 41 41
...
0xbfff0020
41 41 41 41
Saved Base Pointer
0xbfff0024
41 41 41 41
Saved Return Pointer


0xbfff0000
eb
2a
5e
89
Buffer
0xbfff0004
76
08
c6
46
0xbfff0008
07
89
46
0c
0xbfff000c
89
f3
8d
4e
...
0xbfff0020
41
41
41
41
Saved Base Pointer
0xbfff0024
00
00
ff
bf
Saved Return Pointer


0xbfff0000
eb
2a
5e
89
Buffer
0xbfff0004
76
08
c6
46
0xbfff0008
07
89
46
0c
0xbfff000c
89
f3
8d
4e
...
0xbfff0020
41
41
41
41
Saved Base Pointer
0xbfff0024
00
00
ff
bf
Saved Return Pointer


0xbfff0000
eb
2a
5e
89
Buffer
0xbfff0004
76
08
c6
46
0xbfff0008
07
89
46
0c
0xbfff000c
89
f3
8d
4e
...
0xbfff0020
41
41
41
41
Saved Base Pointer
0xbfff0024
00
00
ff
bf
Saved Return Pointer

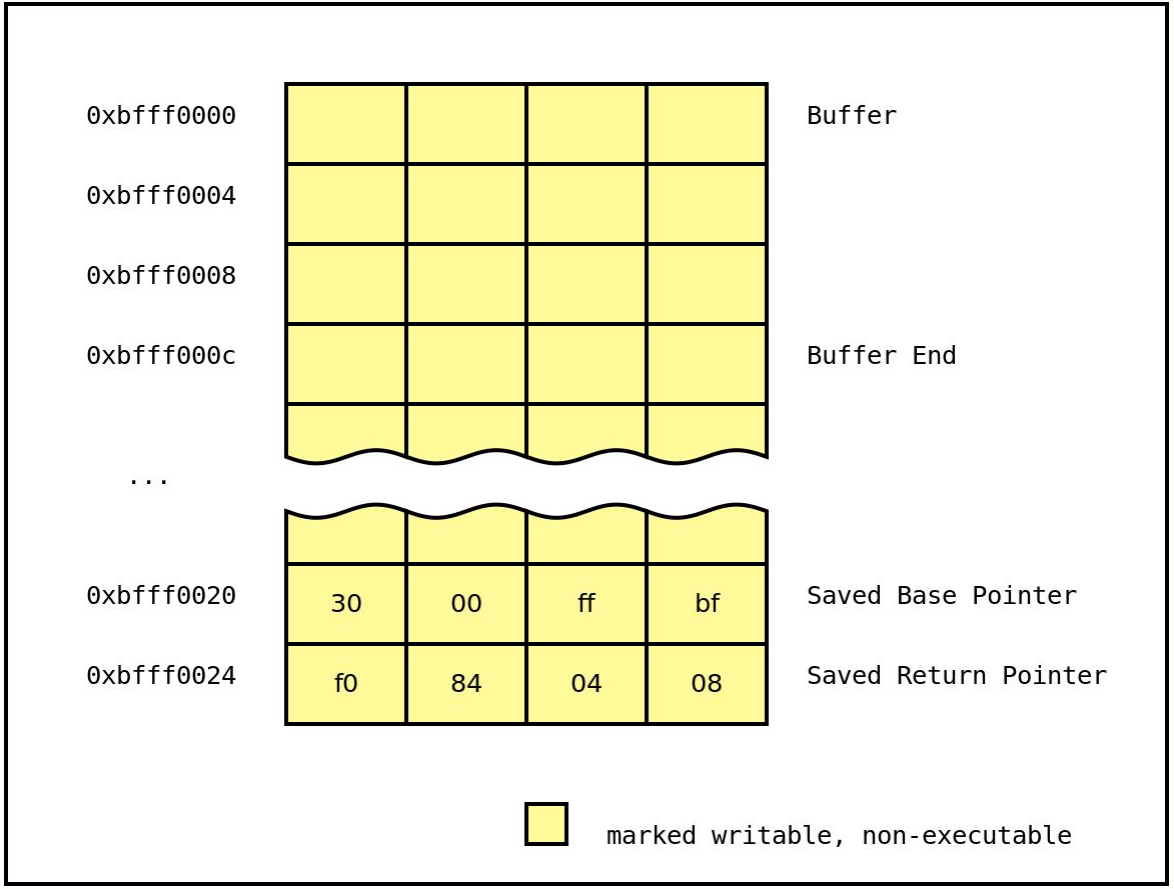
amon@pwned: ~ $

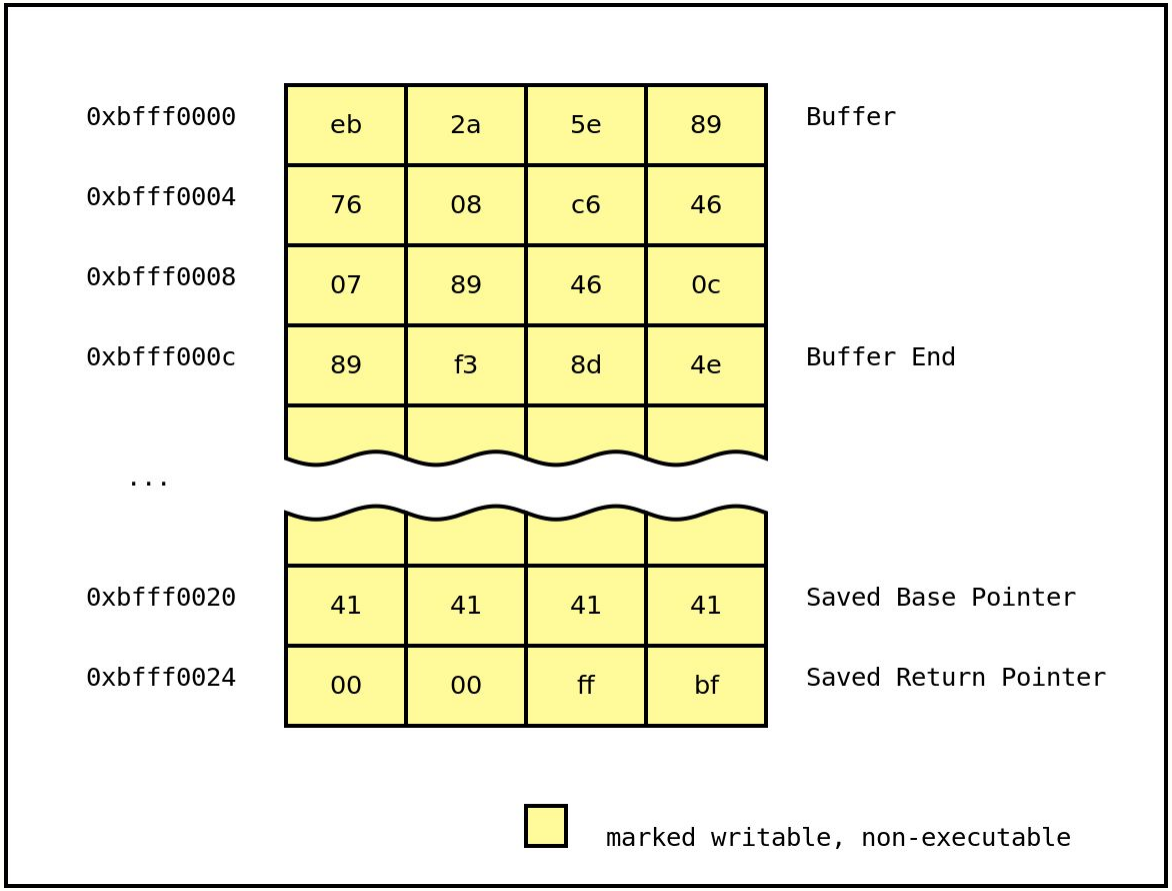
ASLR, NX, and Stack Canaries

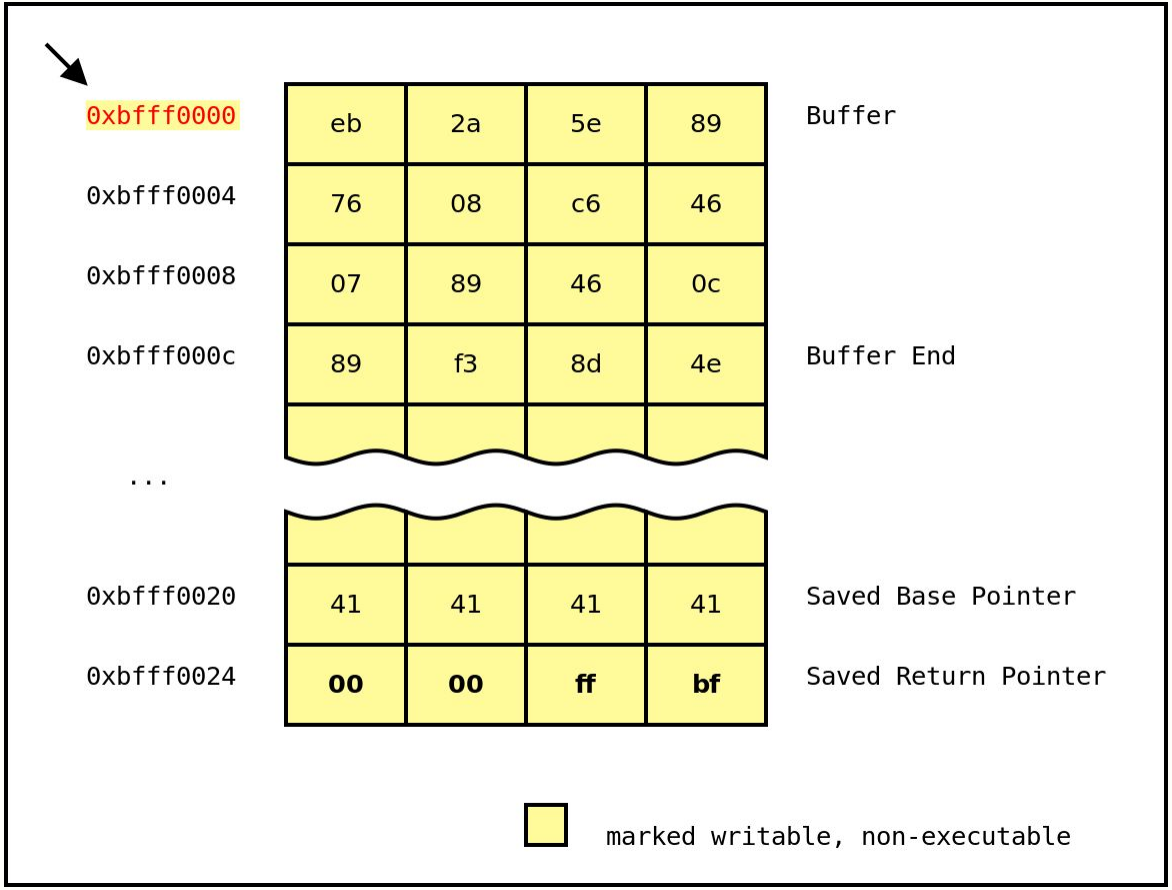
No eXecute (NX)

- a.k.a. Data Execution Prevention (DEP)
- Marks writable regions of memory non-executable
- The stack and heap are typically marked non-executable

Applying the Classic Attack Payload


0xbfff0000
0xbfff0004
0xbfff0008
0xbfff000c
...
0xbfff0020
0xbfff0024
30
00
ff
bf
f0
84
04
08
Buffer
Buffer End
Saved Base Pointer
Saved Return Pointer
marked writable, non-executable

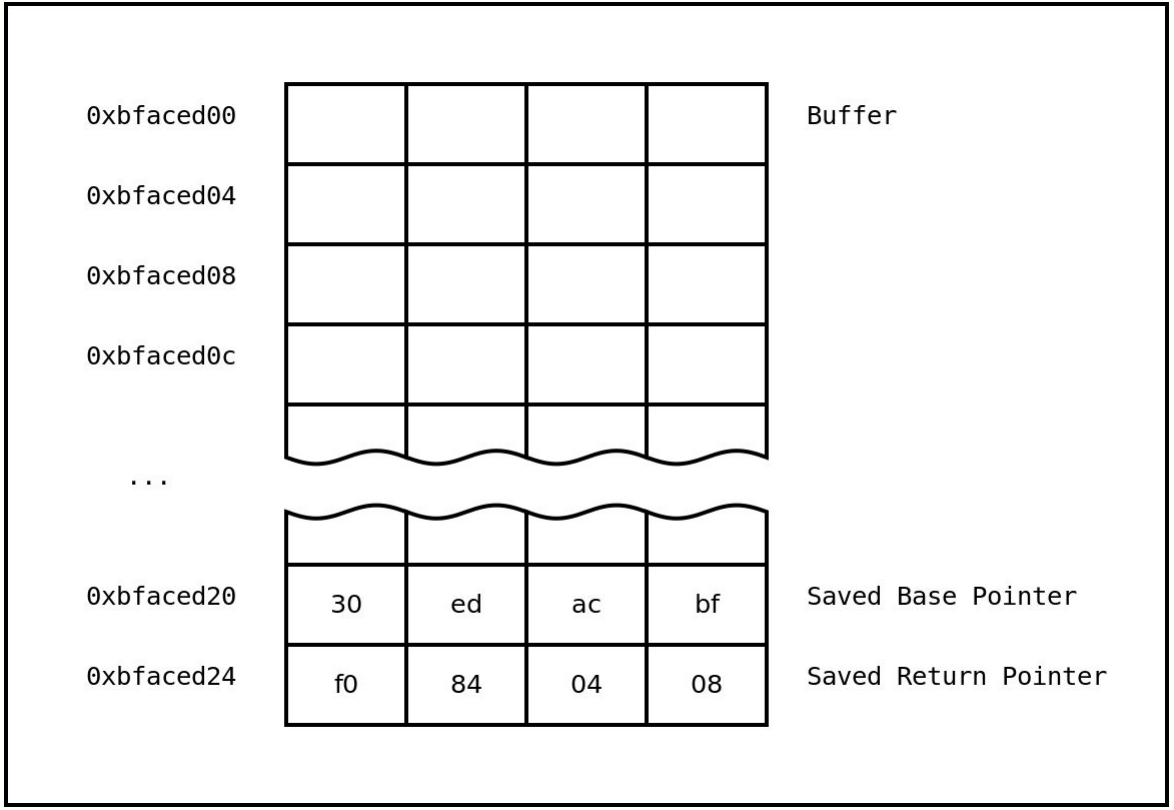

0xbfff0000
eb
2a
5e
89
Buffer
0xbfff0004
76
08
c6
46
0xbfff0008
07
89
46
0c
0xbfff000c
89
f3
8d
4e
Buffer End
...
0xbfff0020
41
41
41
41
Saved Base Pointer
0xbfff0024
00
00
ff
bf
Saved Return Pointer
marked writable, non-executable


0xbfff0000
eb
2a
5e
89
Buffer
0xbfff0004
76
08
c6
46
0xbfff0008
07
89
46
0c
0xbfff000c
89
f3
8d
4e
Buffer End
...
0xbfff0020
41
41
41
41
Saved Base Pointer
0xbfff0024
00
00
ff
bf
Saved Return Pointer
marked writable, non-executable

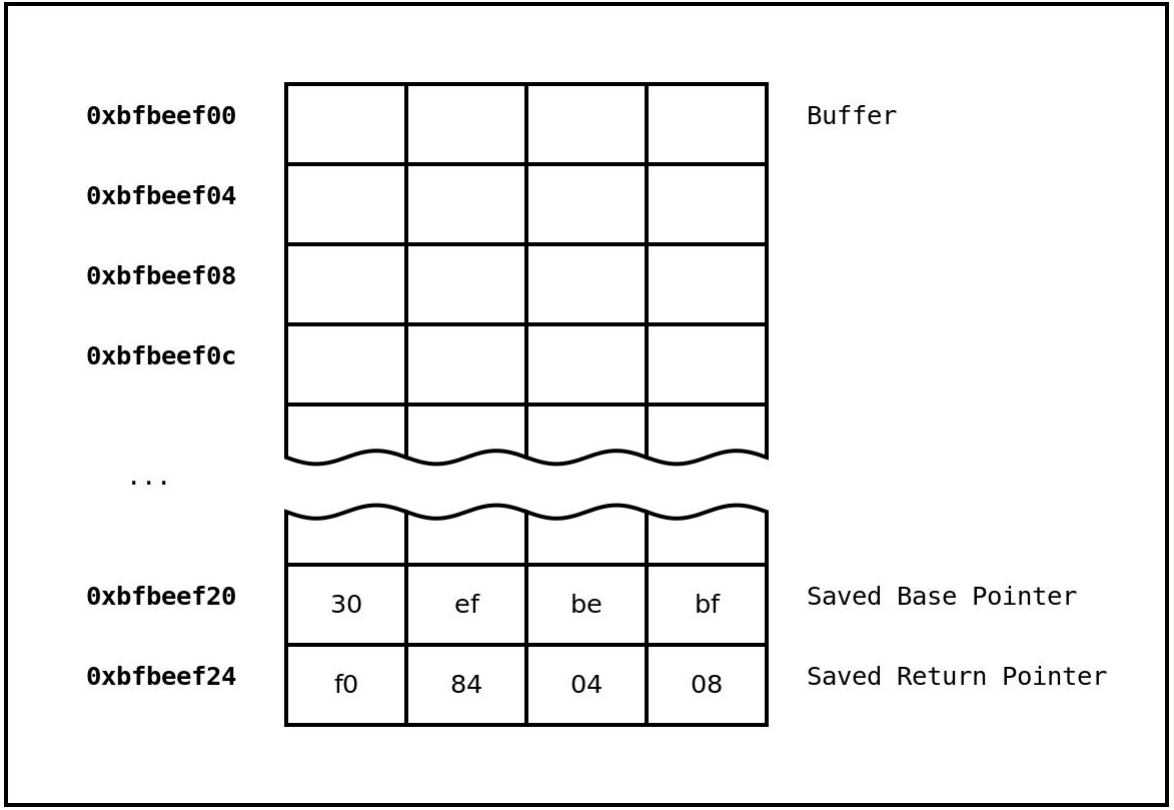
Address Space Layout Randomisation (ASLR)

- Randomises the addresses where shared libs., stack, and heap are mapped at.
- Frustrates an attacker since they cannot predict reliably.

Two Different Runs

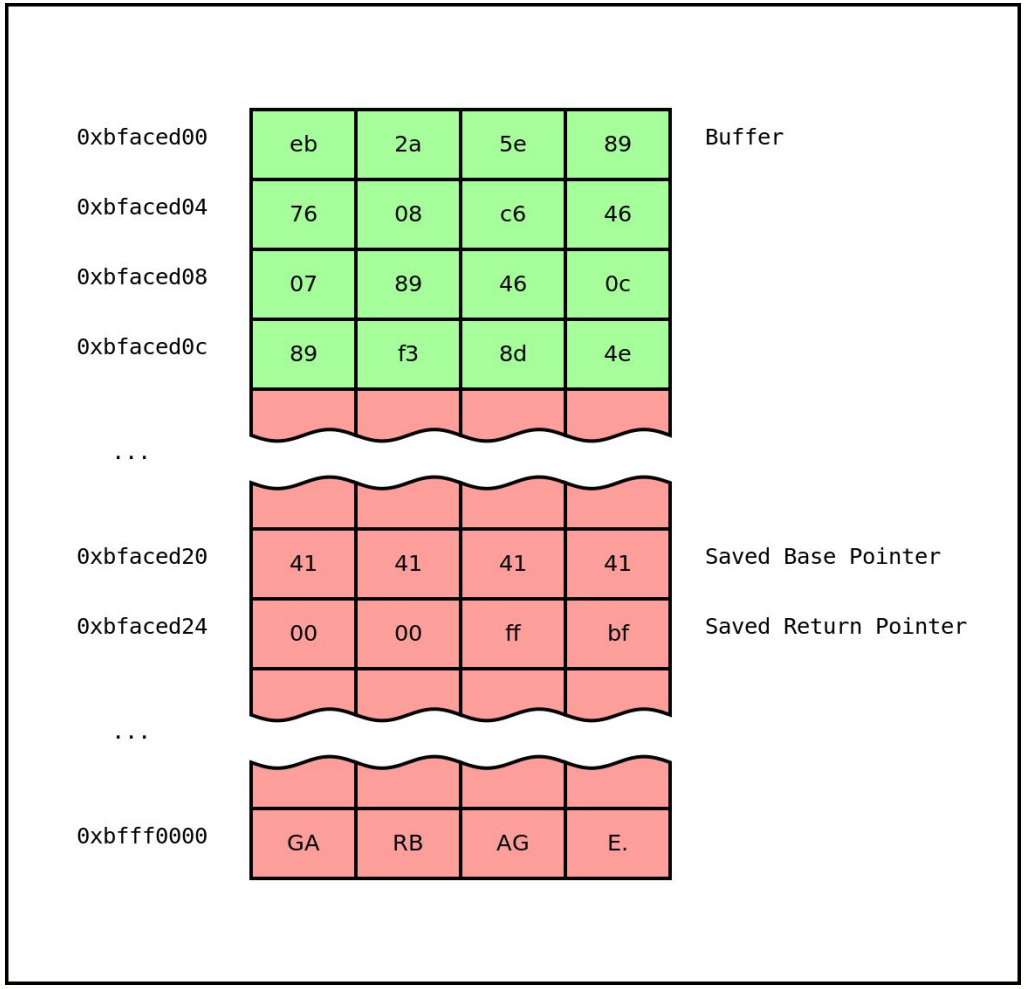

0xbfaced00
0xbfaced04
0xbfaced08
0xbfaced0c
...
0xbfaced20
0xbfaced24
30
ed
ac
bf
f0
84
04
08
Buffer
Saved Base Pointer
Saved Return Pointer

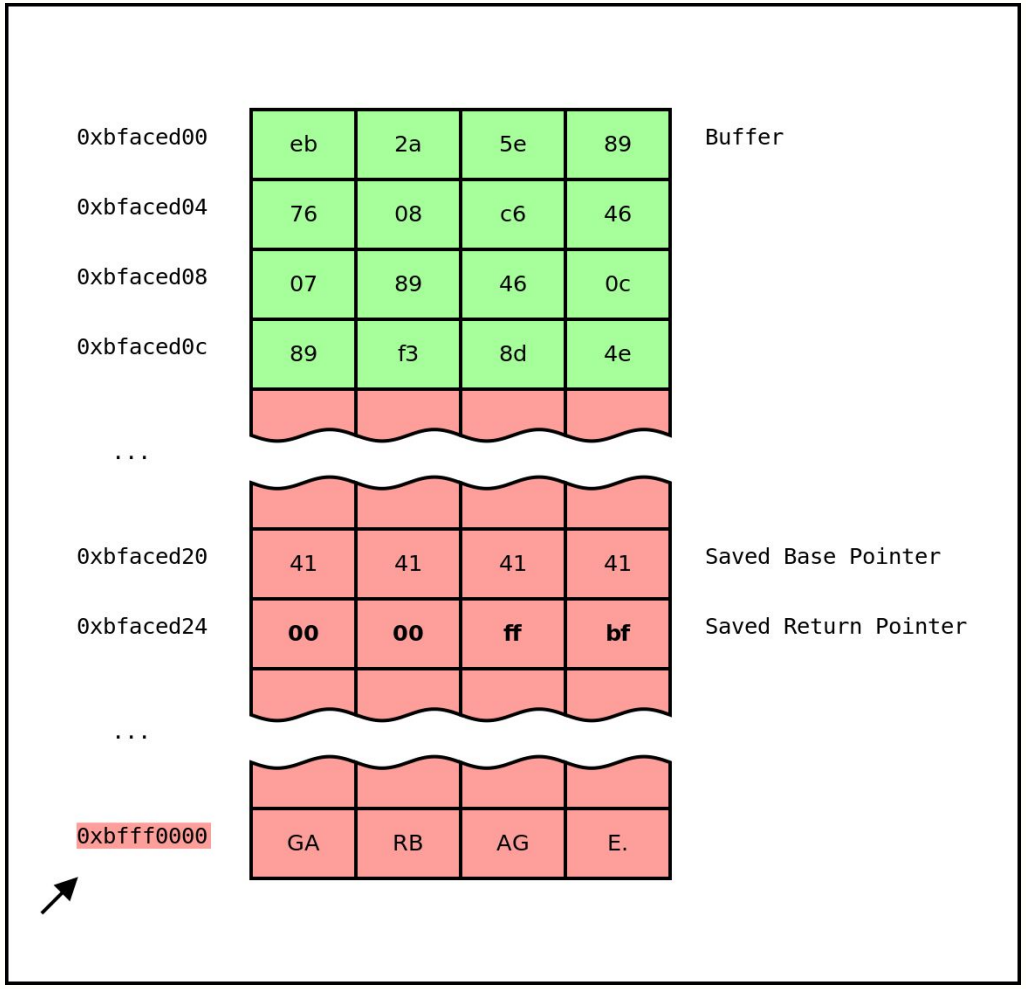
Run 1


0xbfbeef00
0xbfbeef04
0xbfbeef08
0xbfbeef0c
...
0xbfbeef20
0xbfbeef24
30
ef
be
bf
f0
84
04
08
Buffer
Saved Base Pointer
Saved Return Pointer

Run 2

Applying the Classic Attack Payload

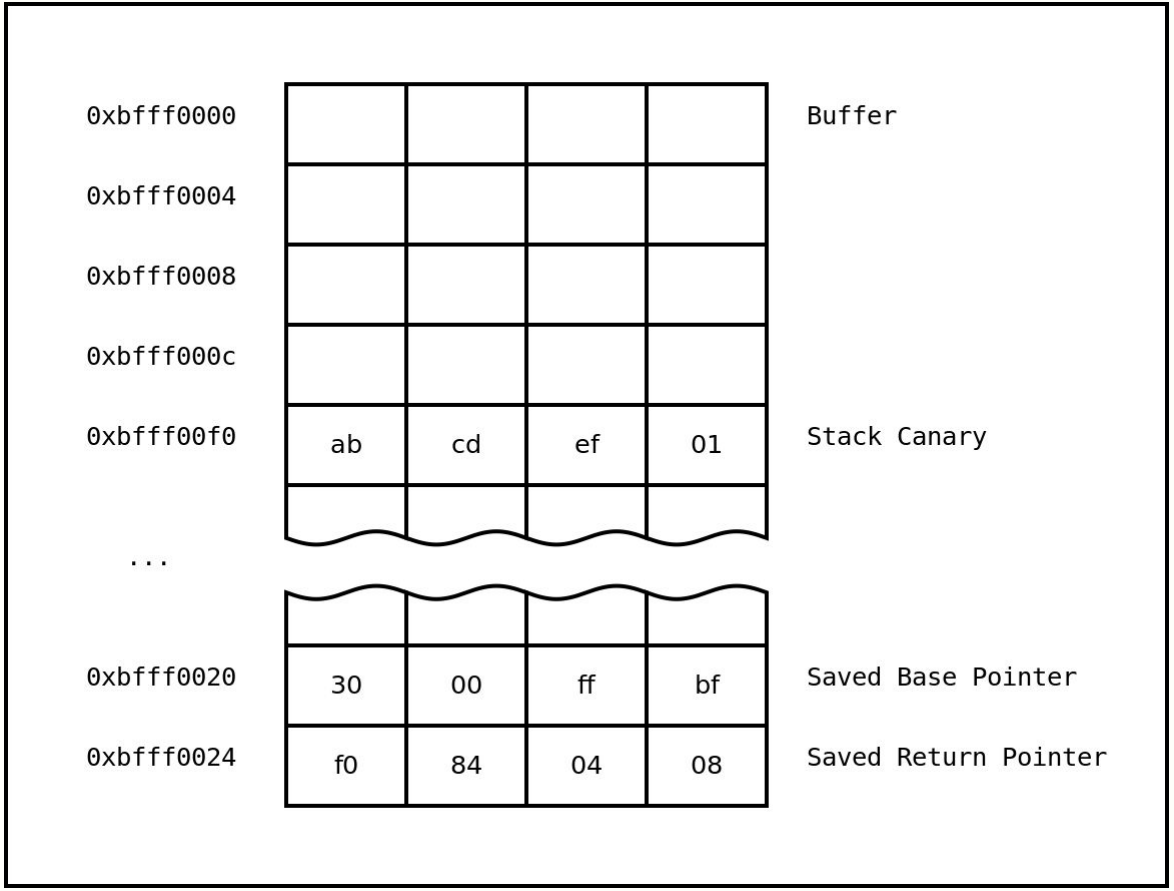

0xbfaced00
eb
2a
5e
89
Buffer
0xbfaced04
76
08
c6
46
0xbfaced08
07
89
46
0c
0xbfaced0c
89
f3
8d
4e
...
0xbfaced20
41
41
41
41
Saved Base Pointer
0xbfaced24
00
00
ff
bf
Saved Return Pointer
...
0xbfff0000
GA
RB
AG
E.

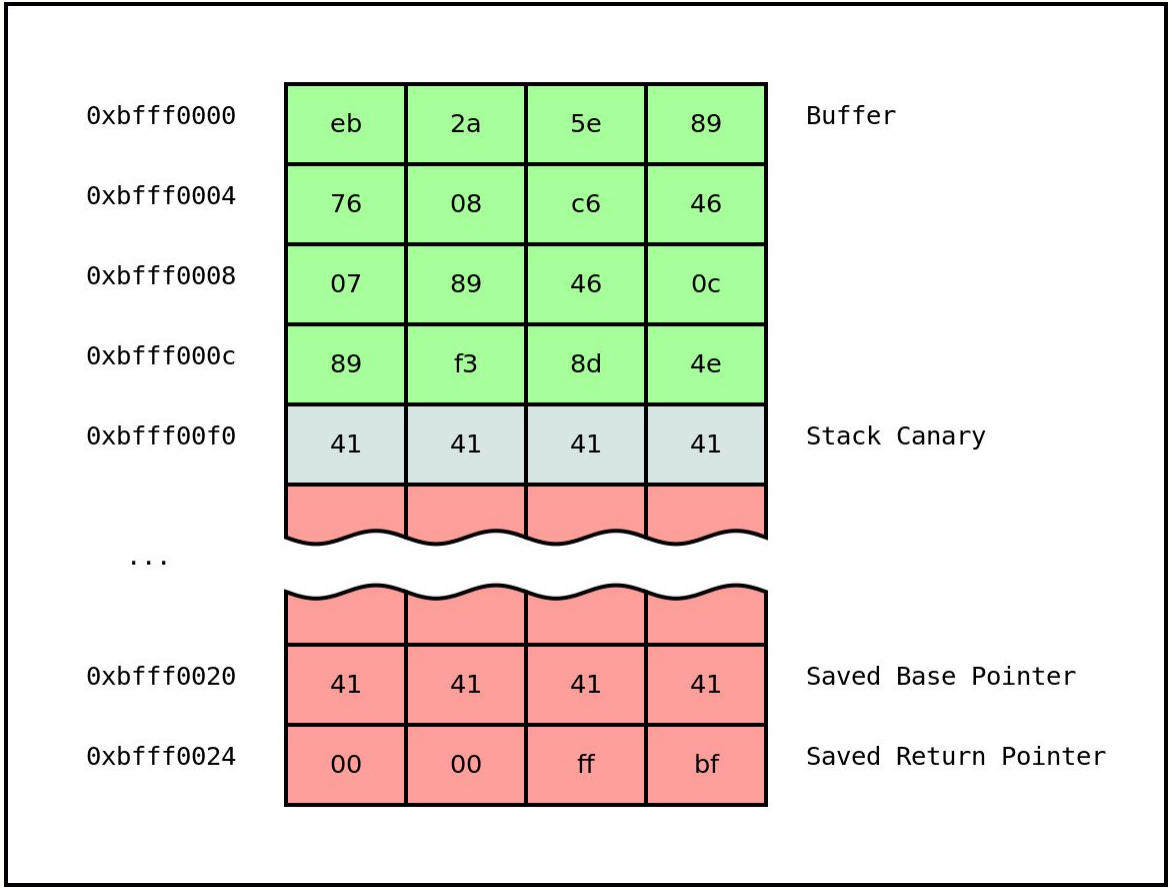

0xbfaced00
eb
2a
5e
89
Buffer
0xbfaced04
76
08
c6
46
0xbfaced08
07
89
46
0c
0xbfaced0c
89
f3
8d
4e
...
0xbfaced20
41
41
41
41
Saved Base Pointer
0xbfaced24
00
00
ff
bf
Saved Return Pointer
...
0xbfff0000
GA
RB
AG
E.

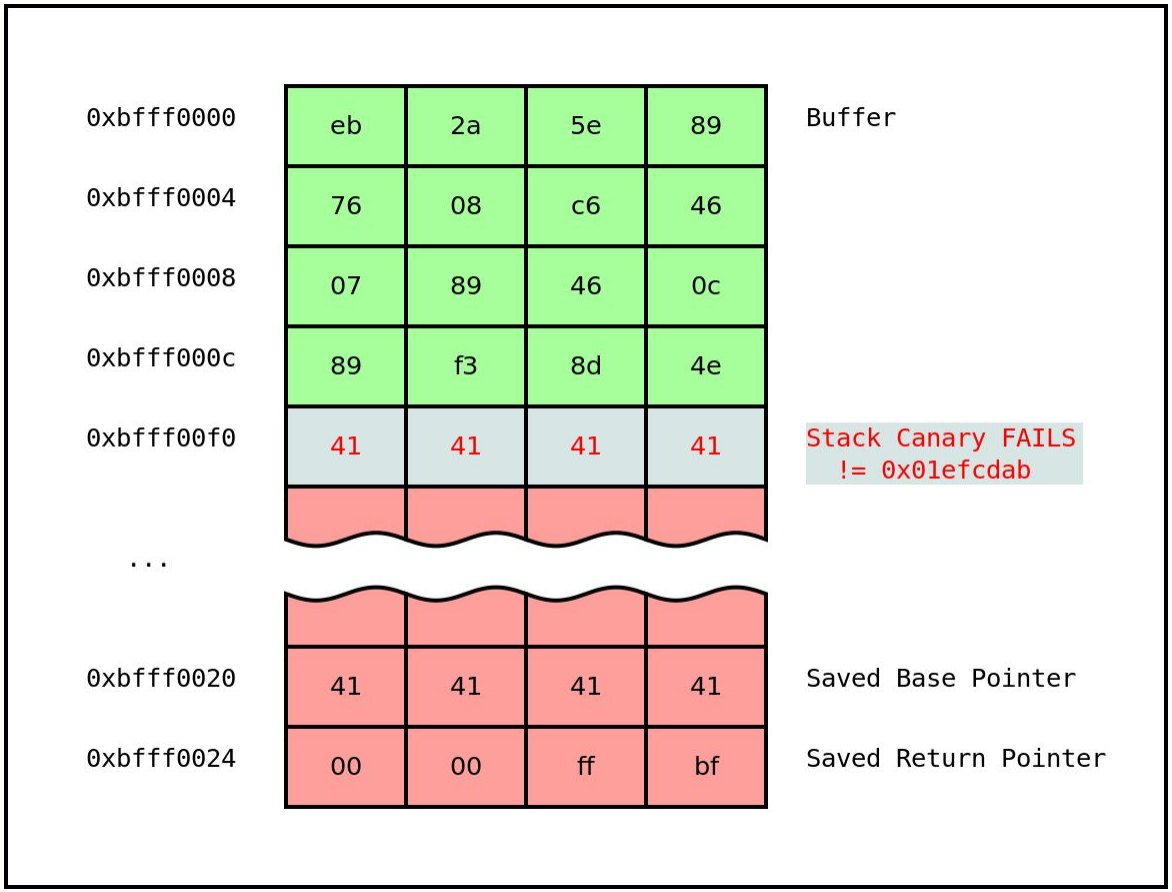
Stack Canaries

- Places a randomised guard value
  - After the stack frame's local variables
  - Before the saved return address
- The guard is checked before the function returns
  - Program exits with error message if it differs
  - Program continues execution as per normal if it matches

Applying the Classic Attack Payload


0xbfff0000
0xbfff0004
0xbfff0008
0xbfff000c
0xbfff00f0
ab
cd
ef
01
Stack Canary
...
0xbfff0020
30
00
ff
bf
Saved Base Pointer
0xbfff0024
f0
84
04
08
Saved Return Pointer
Buffer


0xbfff0000
eb
2a
5e
89
Buffer
0xbfff0004
76
08
c6
46
0xbfff0008
07
89
46
0c
0xbfff000c
89
f3
8d
4e
0xbfff00f0
41
41
41
41
Stack Canary
...
0xbfff0020
41
41
41
41
Saved Base Pointer
0xbfff0024
00
00
ff
bf
Saved Return Pointer


0xbfff0000
eb
2a
5e
89
Buffer
0xbfff0004
76
08
c6
46
0xbfff0008
07
89
46
0c
0xbfff000c
89
f3
8d
4e
0xbfff00f0
41
41
41
41
Stack Canary FAILS
!= 0x01efcdab
...
0xbfff0020
41
41
41
41
Saved Base Pointer
0xbfff0024
00
00
ff
bf
Saved Return Pointer

Return Oriented Programming

Return Oriented Programming (ROP)

- Complex technique that allows us to bypass NX and ASLR
- Does not work against stack canaries
  - Requires leveraging separate memory leak vulnerability to bypass

Return Oriented Programming (ROP)

- Re-uses executable code portions within the binary or other shared libraries
- These code portions are referred to as ‘Gadgets’
- We will be covering a subset called Ret2PLT in this session

Code Reuse Example

New Vulnerable Program

- Vulnerable to a stack overflow. Can you spot where?
- We assume the attacker doesn't know the password

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

void give_shell() {
    system("/bin/sh");
}

void vuln() {
    char password[16];
    puts("What is the password: ");
    scanf("%s", password);
    if (strcmp(password, "31337h4x") == 0) {
        puts("Correct password!");
        give_shell();
    }
    else {
        puts("Incorrect password!");
    }
}

int main() {
    vuln();
}
```

New Vulnerable Program

- Vulnerable to a stack overflow. Can you spot where?
- We assume the attacker doesn't know the password

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

void give_shell() {
    system("/bin/sh");
}

void vuln() {
    char password[16];
    puts("What is the password: ");
    scanf("%s", password);
    if (strcmp(password, "31337h4x") == 0) {
        puts("Correct password!");
        give_shell();
    }
    else {
        puts("Incorrect password!");
    }
}

int main() {
    vuln();
}
```

From this point onwards, ASLR and NX is enabled.

(You can get a copy of the binary in the repository)

Let’s Verify This

Objective: Achieving EIP Control

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

void give_shell() {
    system("/bin/sh");
}

void vuln() {
    char password[16];
    puts("What is the password: ");
    scanf("%s", password);
    if (strcmp(password, "31337h4x") == 0) {
        puts("Correct password!");
        give_shell();
    }
    else {
        puts("Incorrect password!");
    }
}

int main() {
    vuln();
}
```

Vulnerable Line

The Return Address is at offset 28

Let’s Demonstrate This

Objective: Jump Where?

1. Shellcode in Buffer?
2. Somewhere Else?

1. ~~Shellcode in Buffer?~~
2. Somewhere Else?

Remember the ‘give_shell()’ function?

Let’s get its address.

Objective: Putting It All Together

amon@pwned: ~ $

Function Reuse Example

Most programs don’t simply call system(“/bin/sh”) for you.

Let’s modify the program to be a little more realistic

Newer Vulnerable Program

- Still vulnerable to a stack overflow
- Doesn’t give a shell anymore

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

char * not_allowed = "/bin/sh";

void give_date() {
    system("/bin/date");
}

void vuln() {
    char password[16];
    puts("What is the password: ");
    scanf("%s", password);
    if (strcmp(password, "31337h4x") == 0) {
        puts("Correct password!");
        give_date();
    }
    else {
        puts("Incorrect password!");
    }
}

int main() {
    vuln();
}
```

Newer Vulnerable Program

- Still vulnerable to a stack overflow
- Doesn't give a shell anymore

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

char * not_allowed = "/bin/sh";

void give_date() {
    system("/bin/date");
}

void vuln() {
    char password[16];
    puts("What is the password: ");
    scanf("%s", password);
    if (strcmp(password, "31337h4x") == 0) {
        puts("Correct password!");
        give_date();
    }
    else {
        puts("Incorrect password!");
    }
}

int main() {
    vuln();
}
```

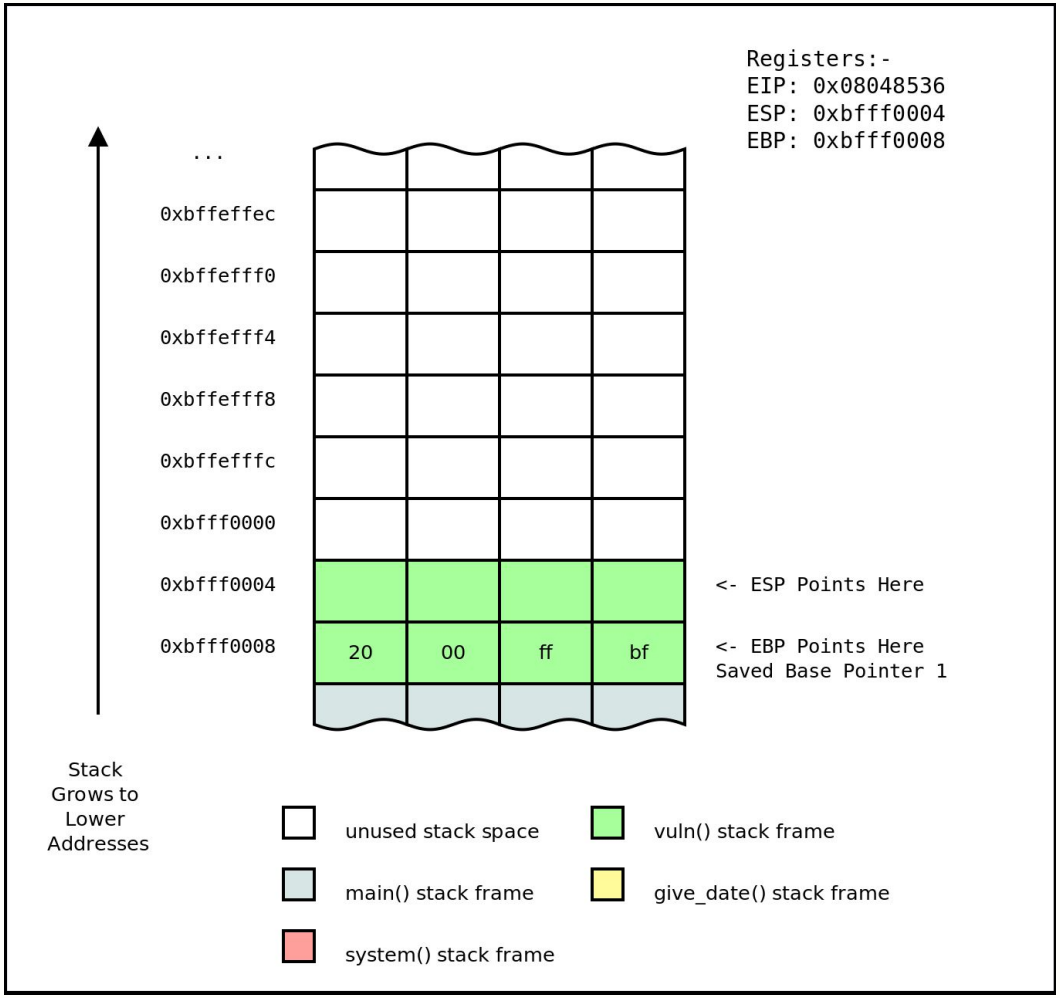
We can’t simply jump to give_date() and get a shell.

To proceed, we need to introduce 32 bit Linux calling conventions.

Calling Conventions

Disassembly Listing

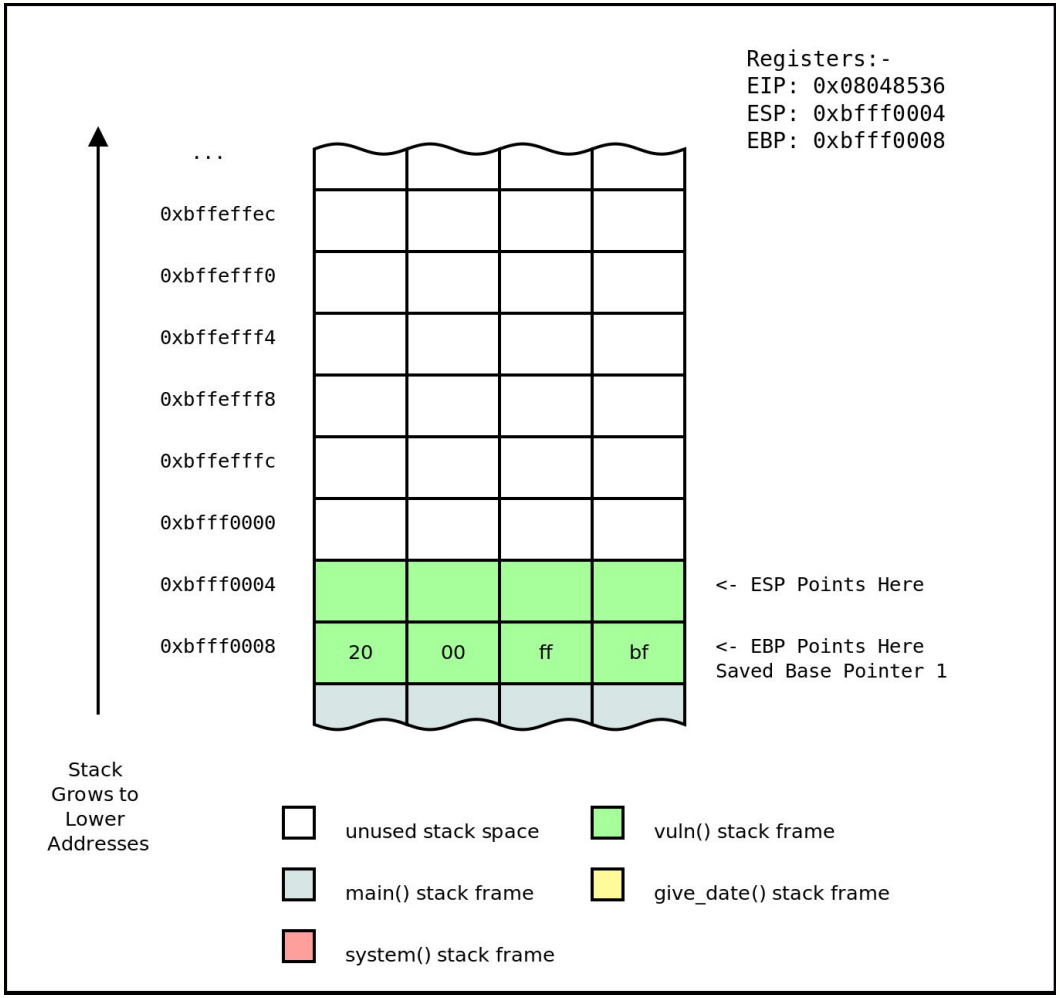
```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...


080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



Disassembly Listing

Function Prologue

```
080484c[illegible]ve_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```

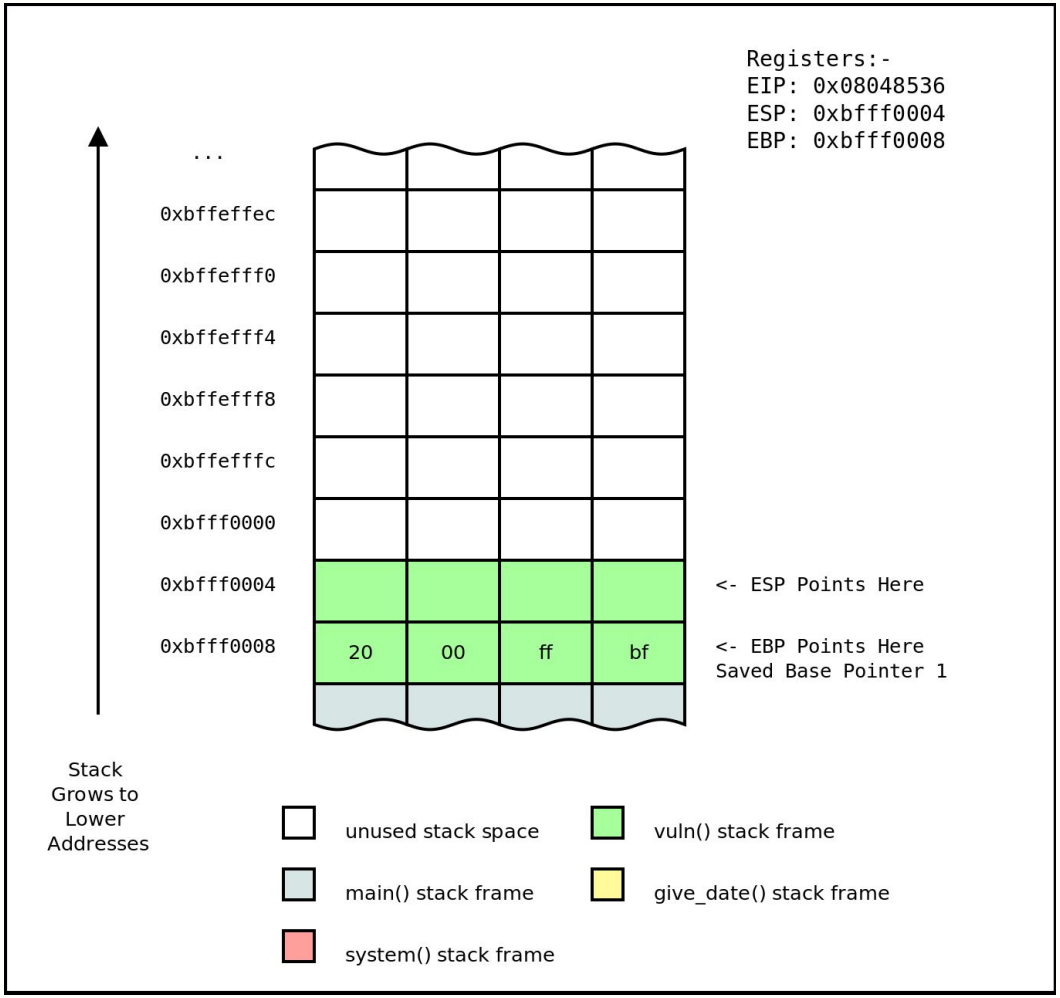


Disassembly Listing

```
080484e4 <vuln>:
...
                                    e>
                                    9>
```

Function Body

```
 80484c  sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```

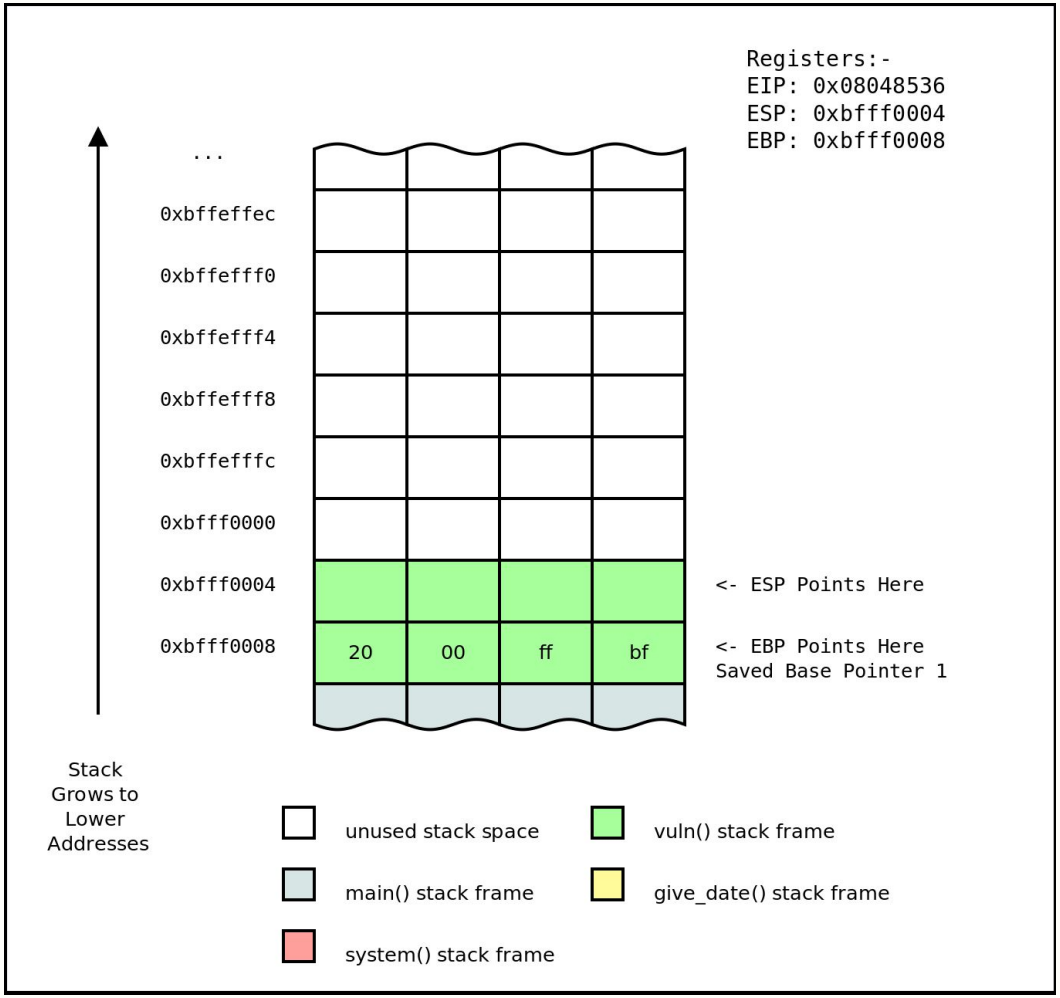


Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
```

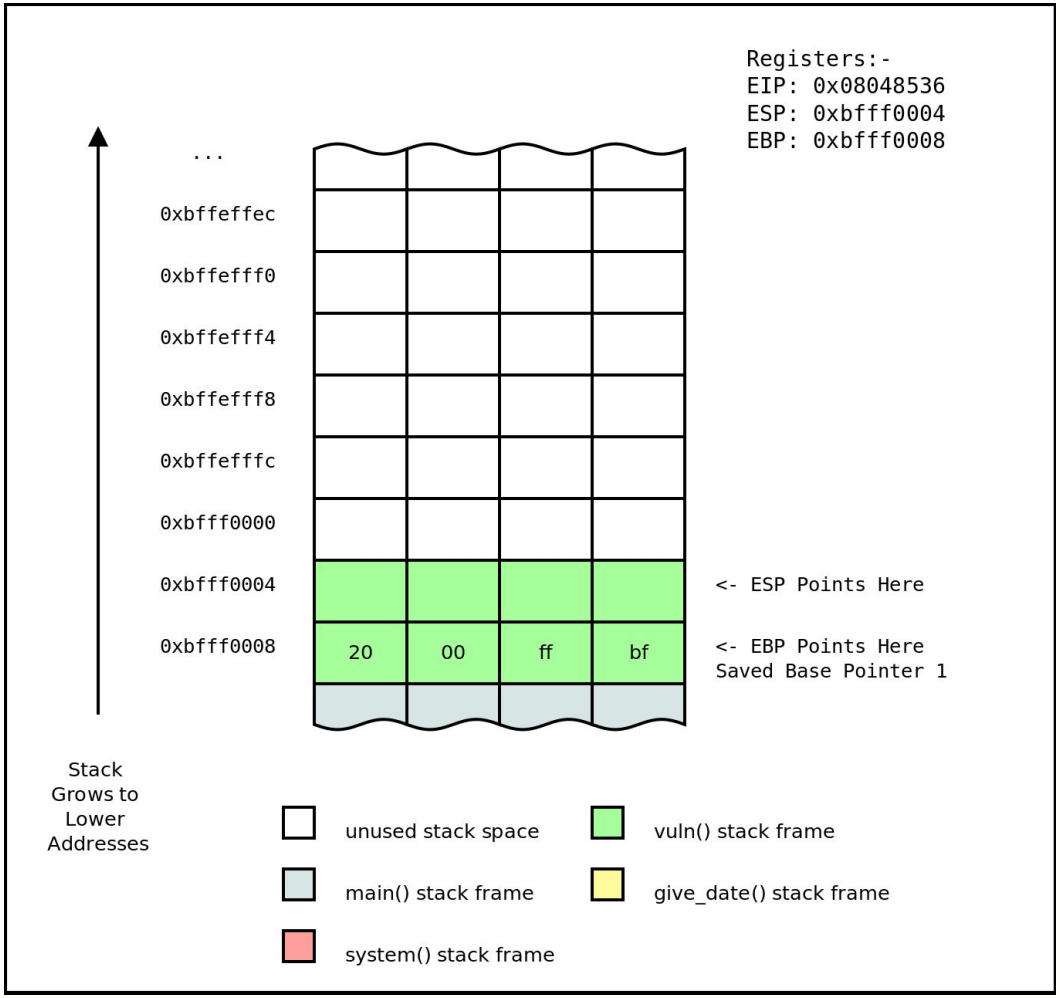
Function Epilogue

```
 80484d[illegible] call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



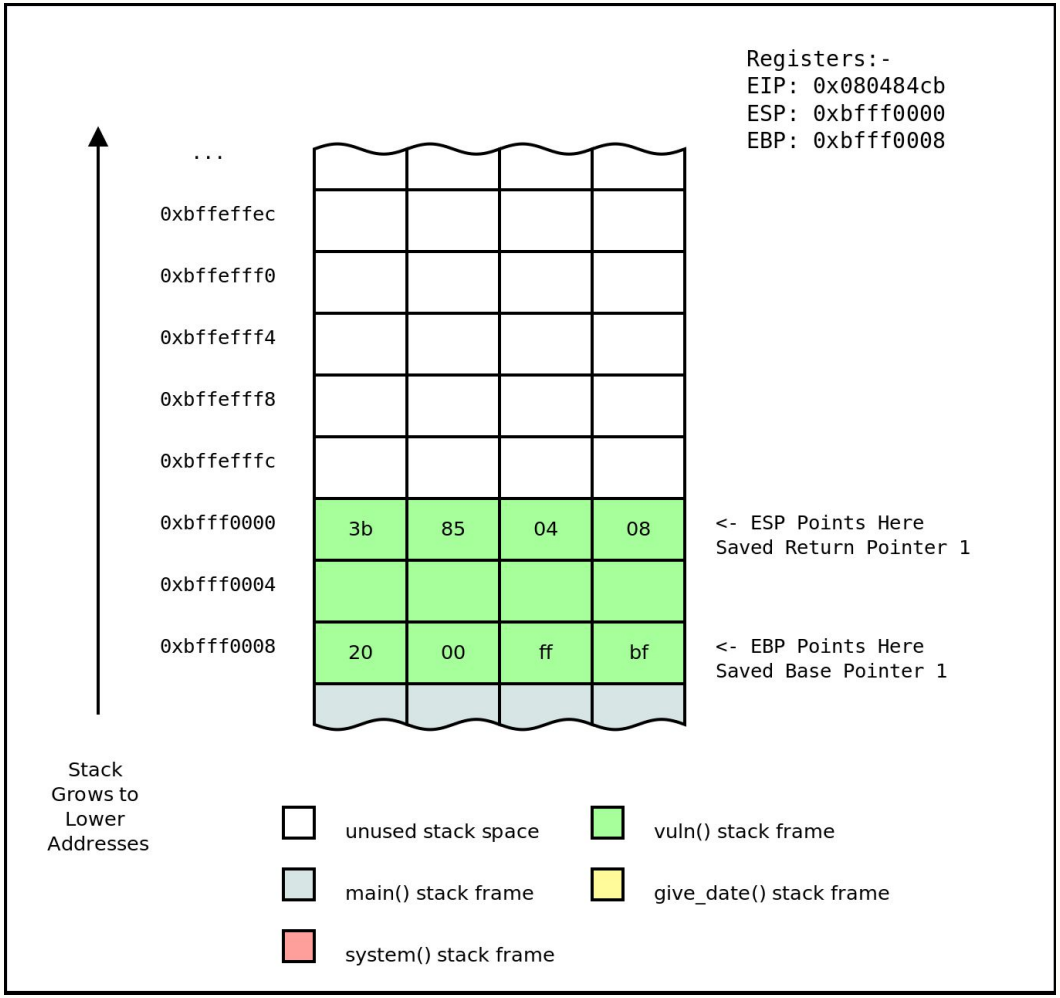
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...

080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



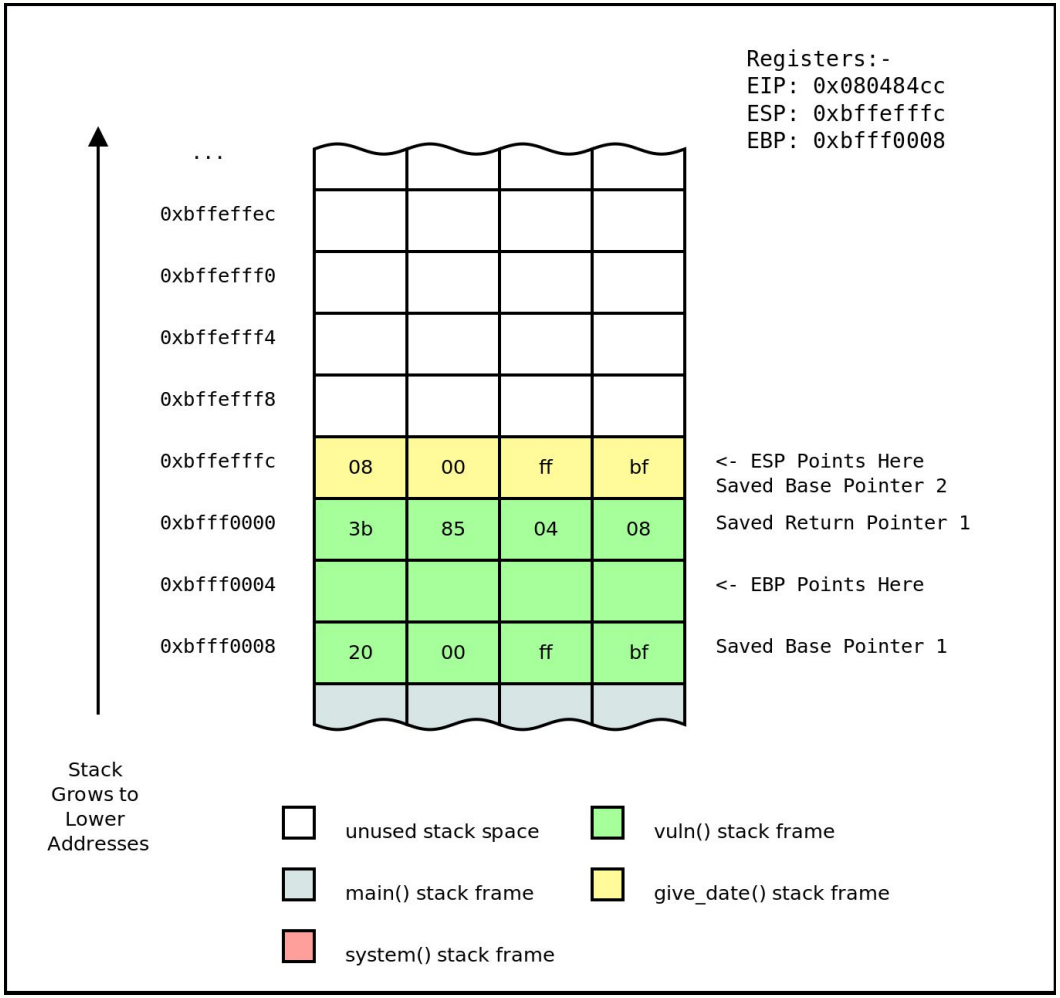
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...

080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



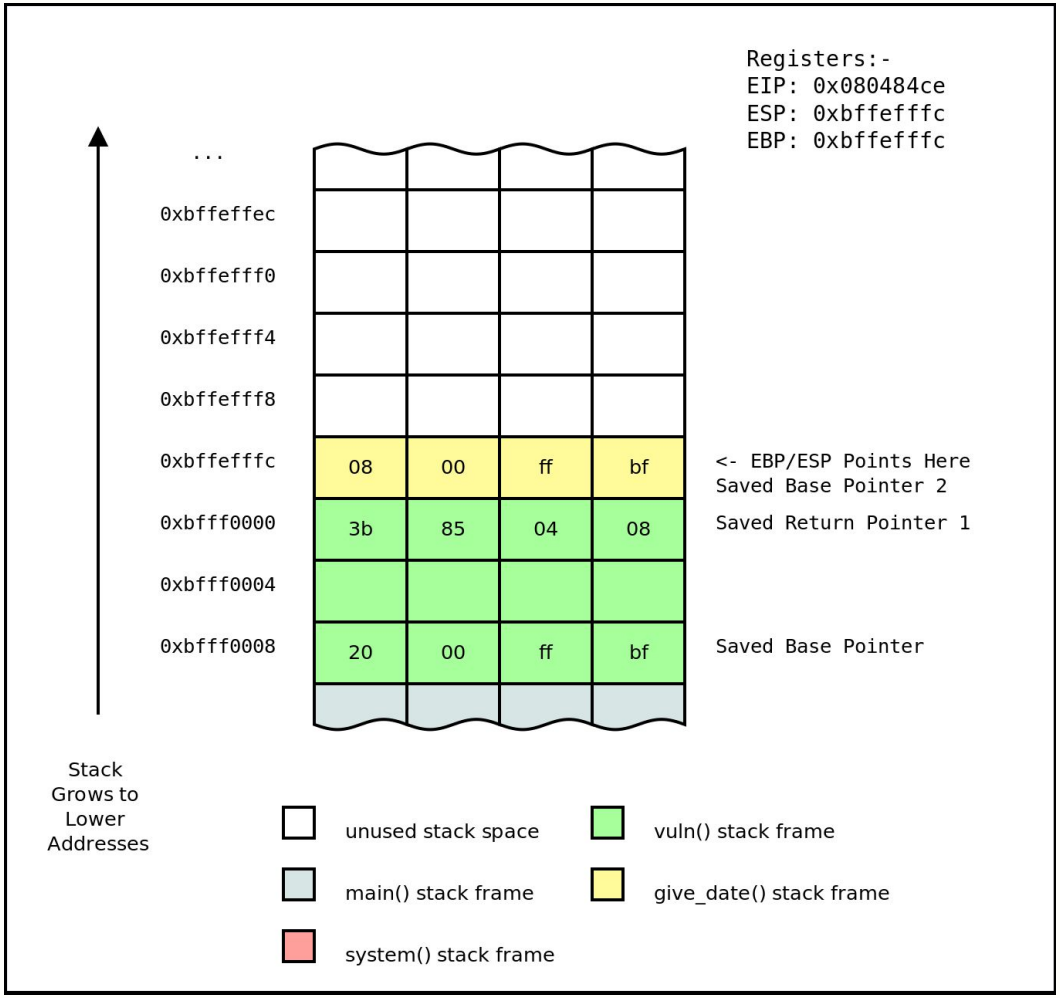
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...


080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



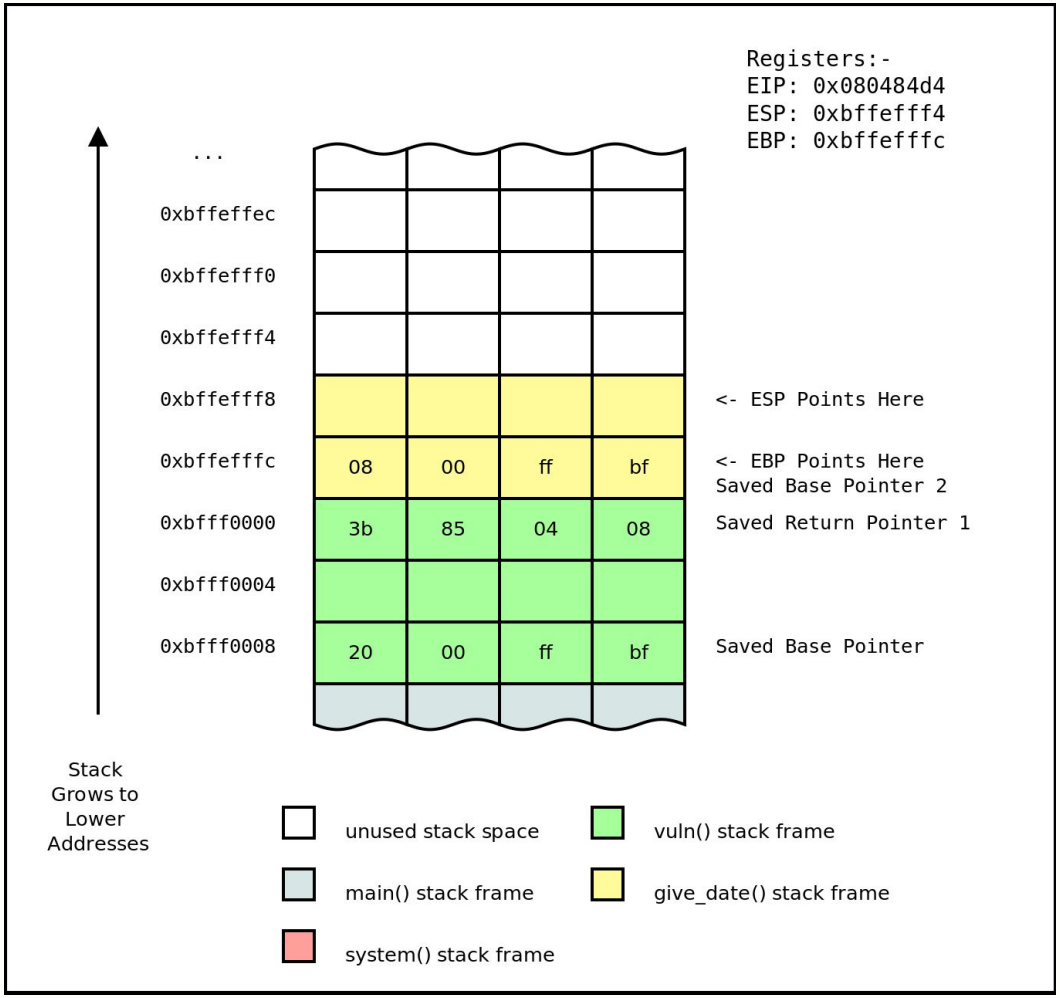
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...

080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



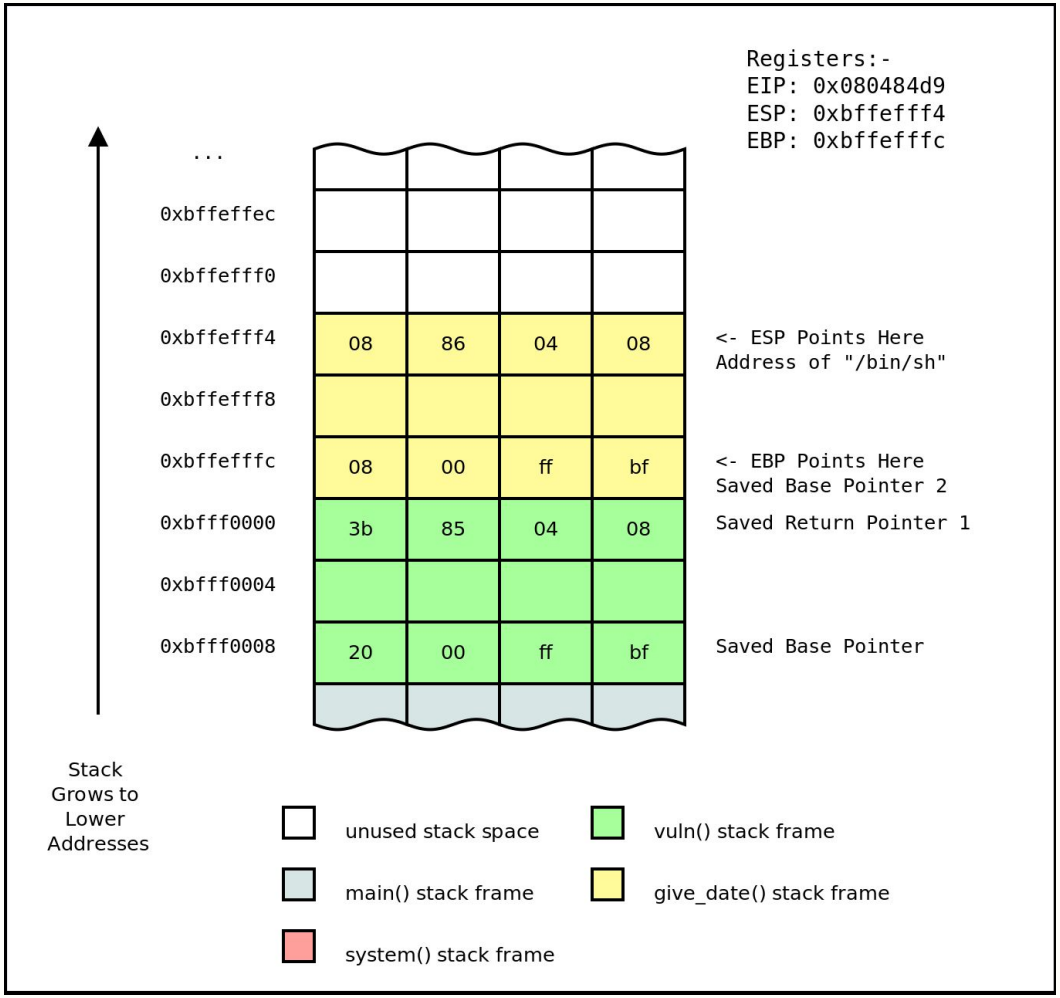
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...

080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



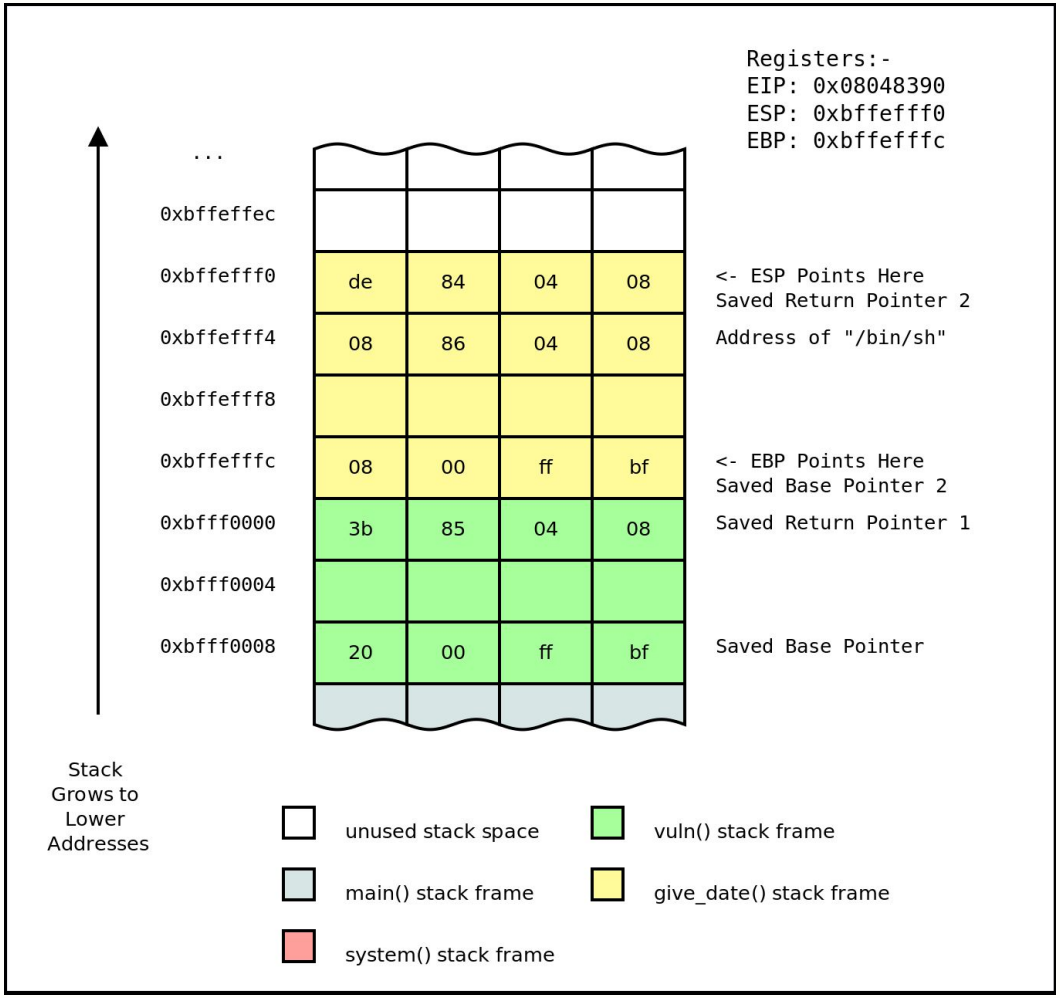
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...


080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



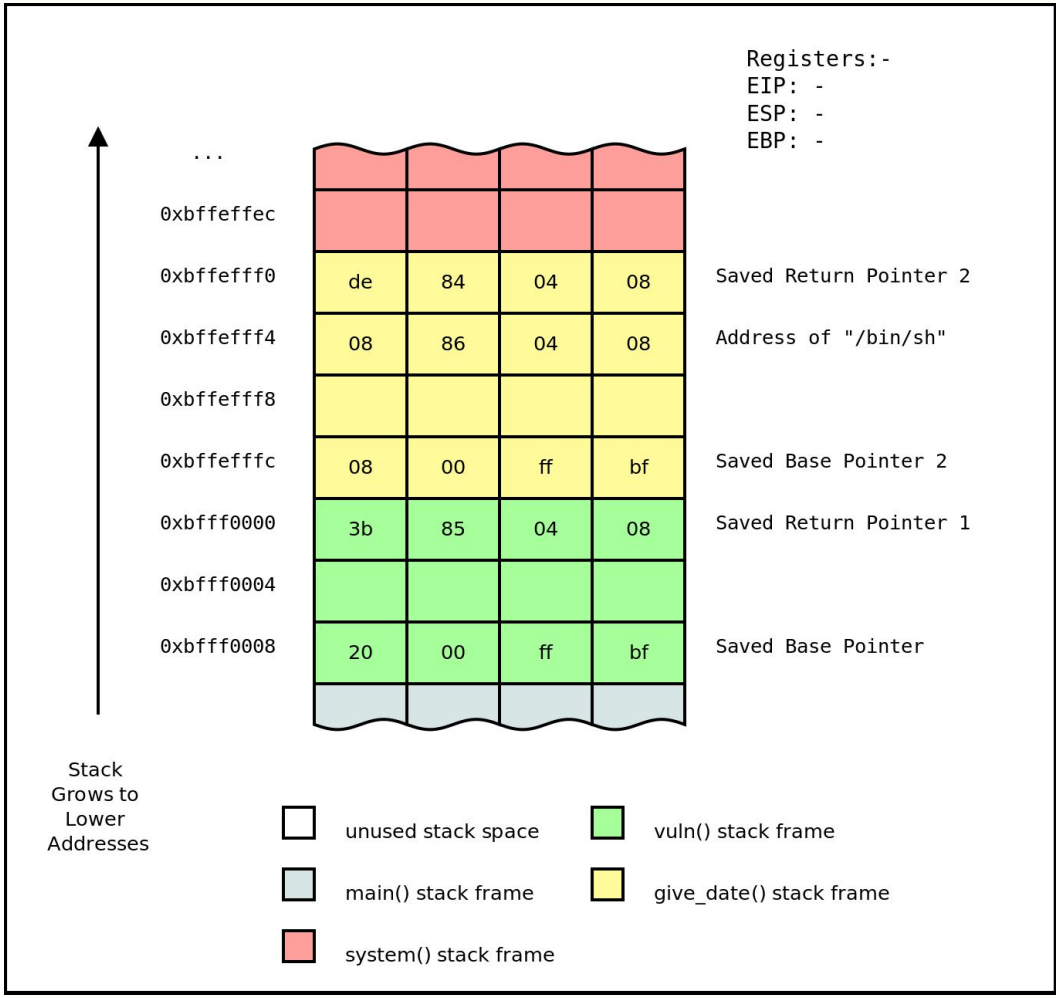
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...


080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret

08048390 <system@plt>:
```



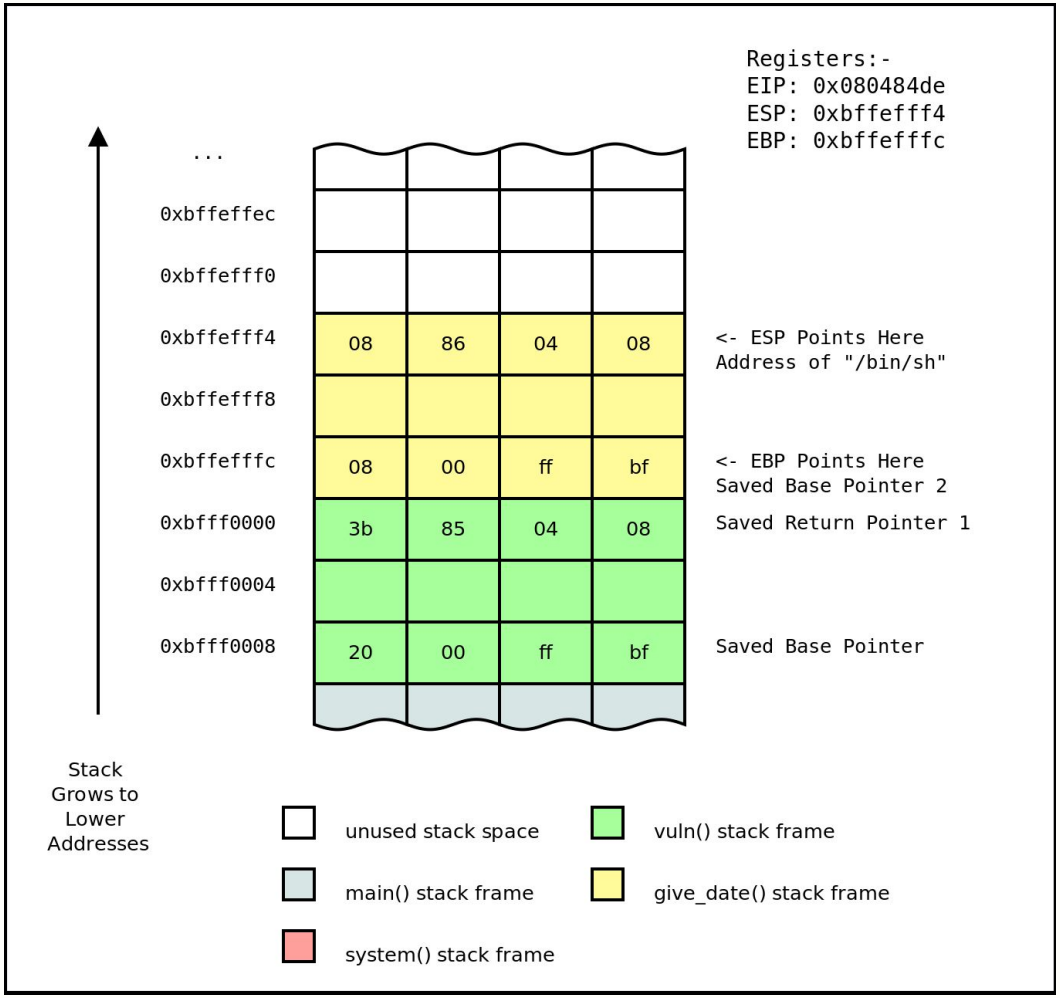
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...


080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret

08048390 <system@plt>:
```



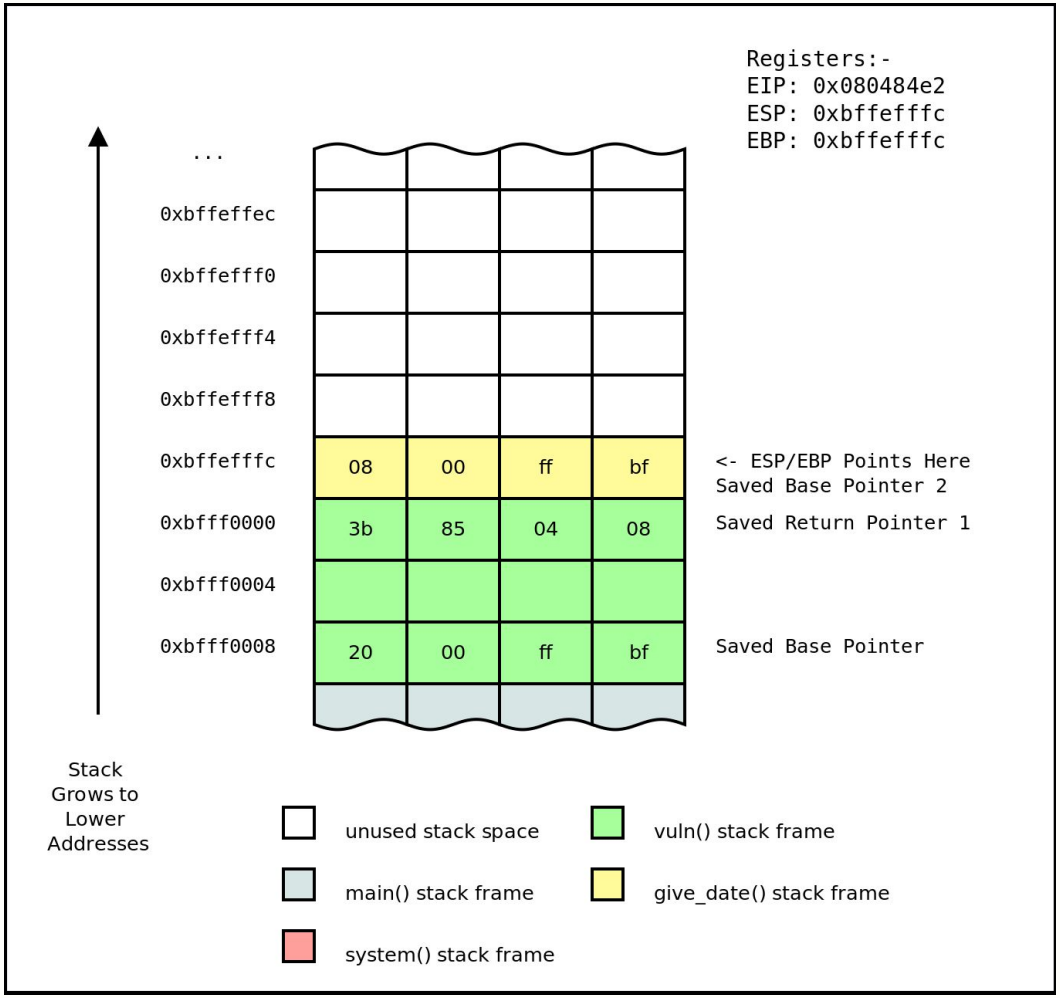
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...


080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



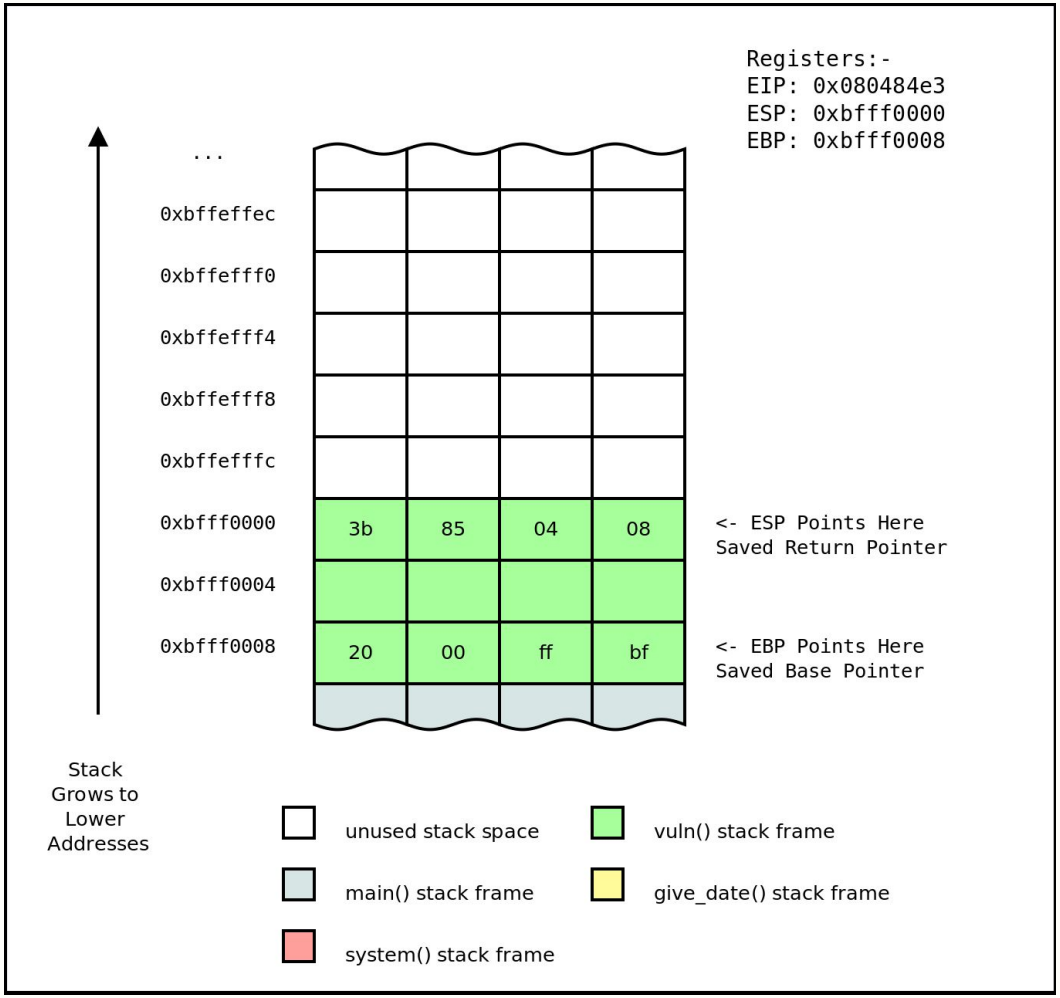
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...

080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



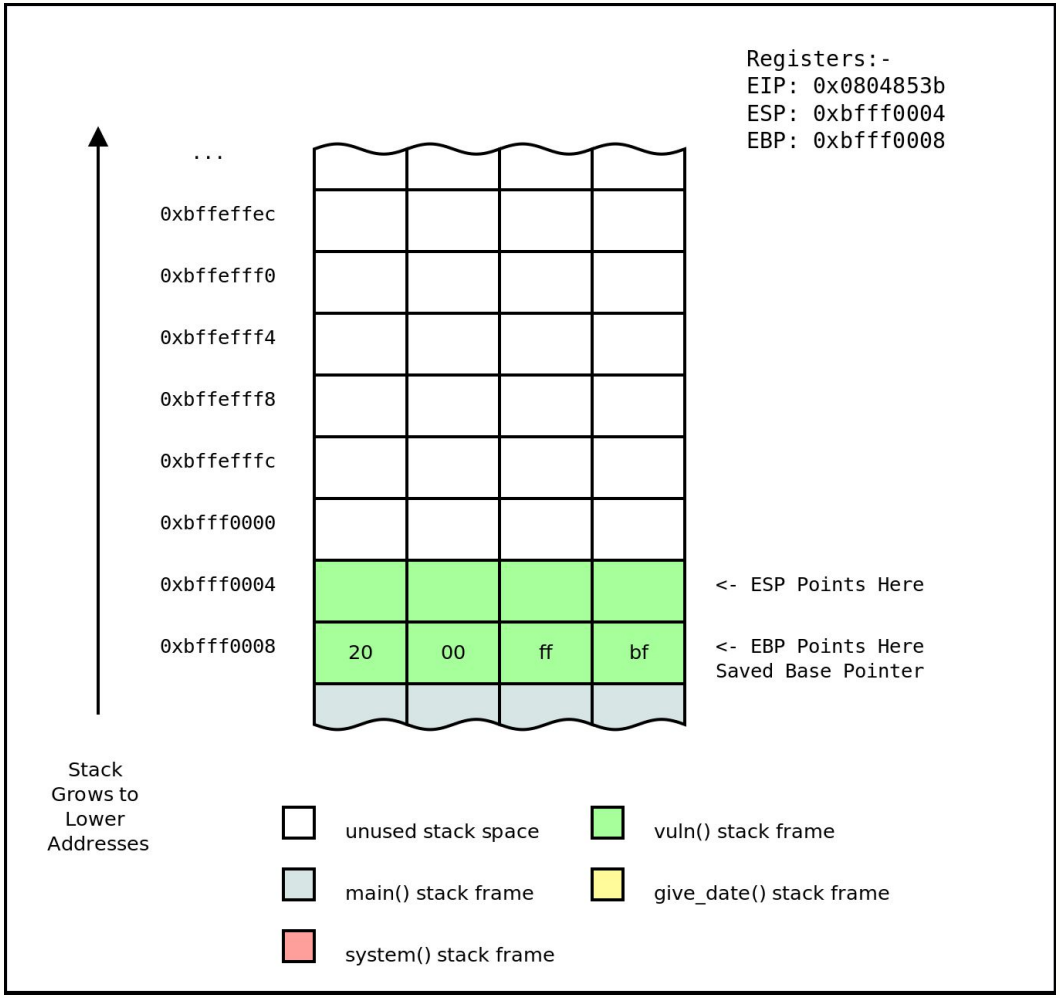
Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...


080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



Disassembly Listing

```
080484e4 <vuln>:
...
 8048536: call 80484cb <give_date>
 804853b: jmp  804854d <vuln+0x69>
...

080484cb <give_date>:
 80484cb: push %ebp
 80484cc: mov  %esp,%ebp
 80484ce: sub  $0x4,%esp
 80484d4: push $0x8048608
 80484d9: call 8048390 <system@plt>
 80484de: add  $0x8,%esp
 80484e2: leave
 80484e3: ret
```



Faking Stack Frames

Faking Stack Frames

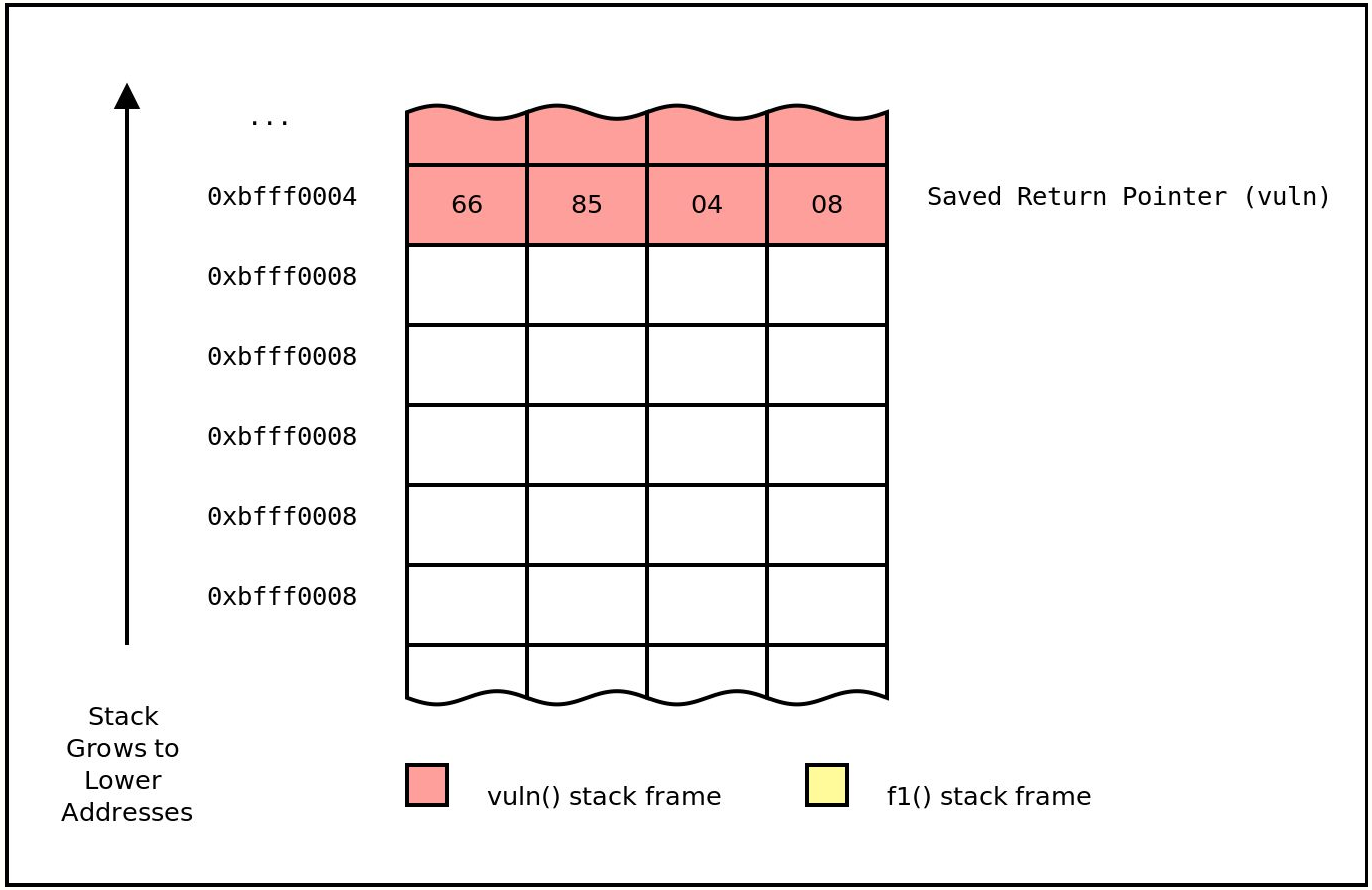
- We showed how stack frames control how a program unwinds
- What are the implications if we could construct and fake our own stack frames?

Implications

- Control the parameters to a function (f1) we jump to
- Decide where f1 returns to (f2)
- If we can manipulate the stack between f1 returning into f2, we can control f2's parameters
- Repeating this process lets us arbitrarily chain functions

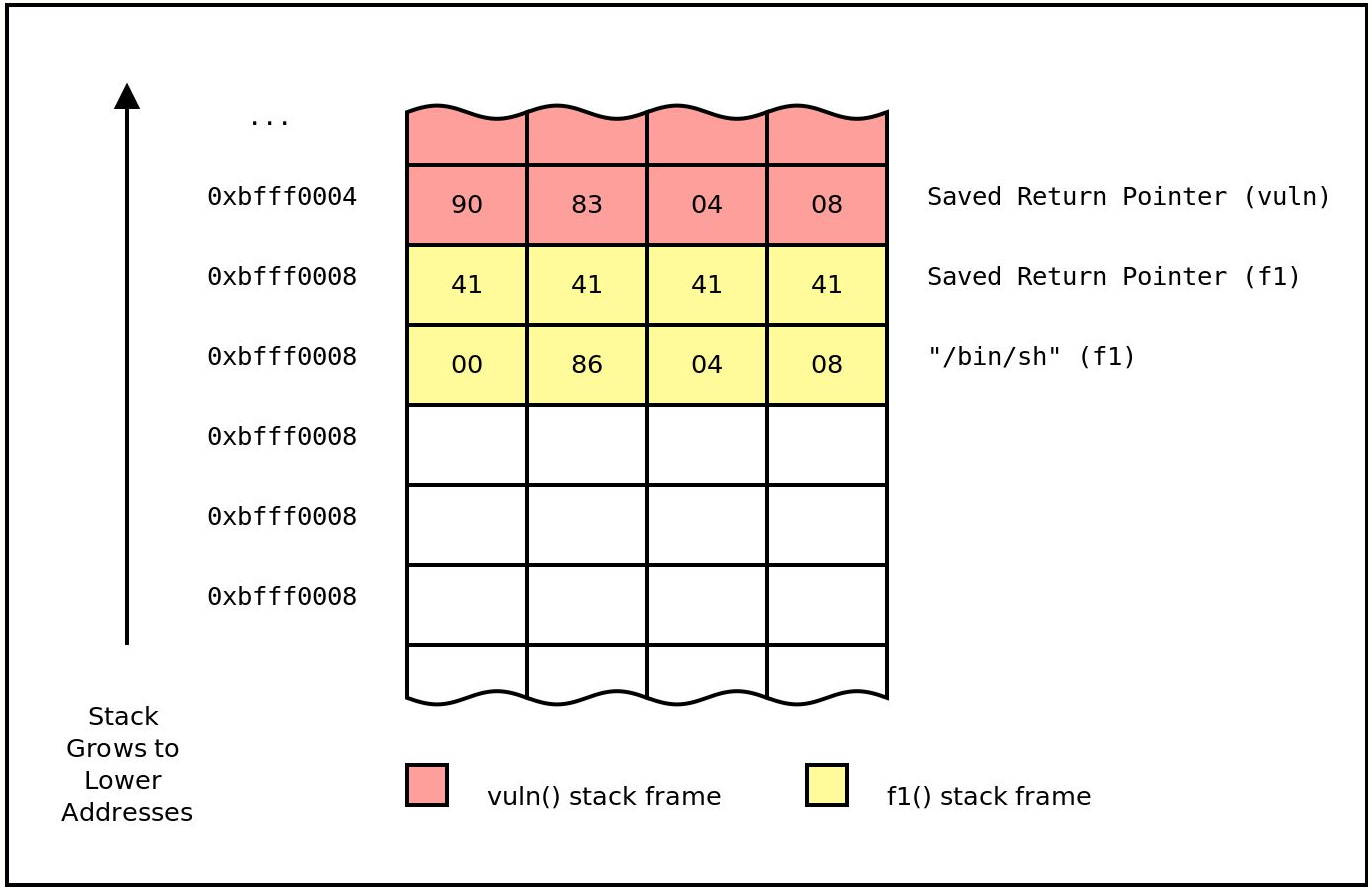
Let’s illustrate this concept

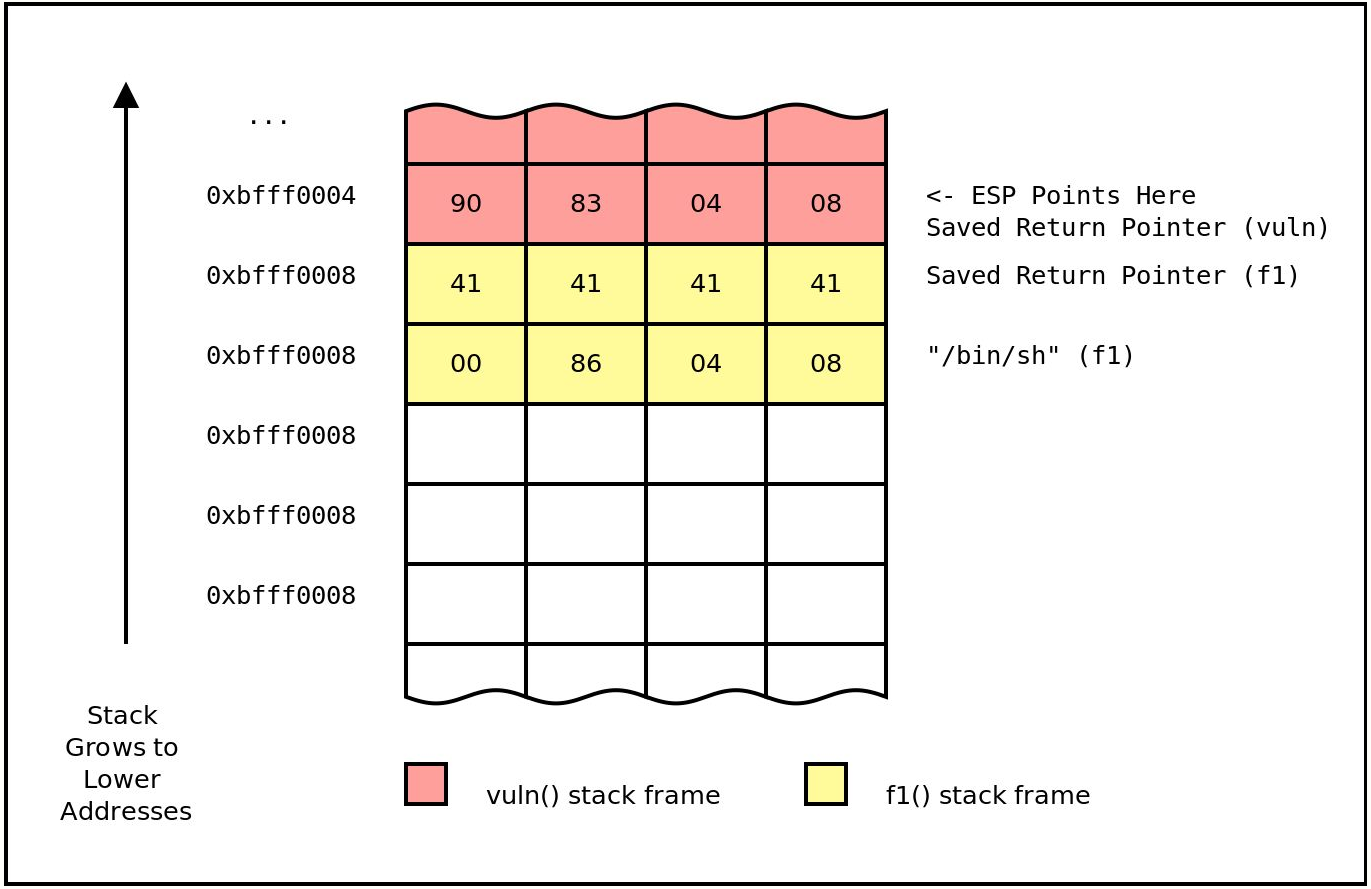
We want to call a single function, system@plt with the parameter “/bin/sh”

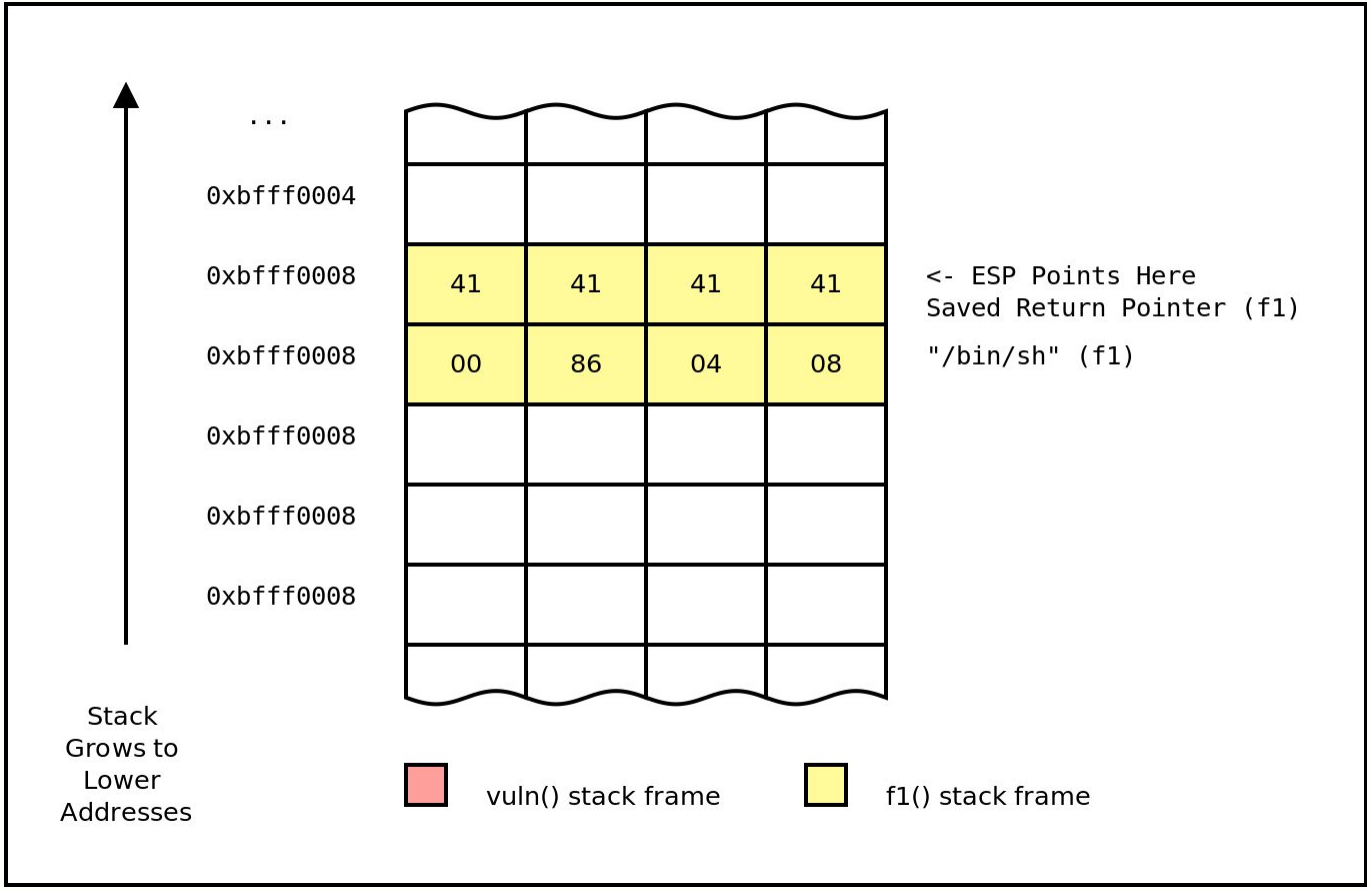

...
0xbfff0004
66
85
04
08
Saved Return Pointer (vuln)
0xbfff0008
0xbfff0008
0xbfff0008
0xbfff0008
0xbfff0008
Stack Grows to Lower Addresses
vuln() stack frame
f1() stack frame

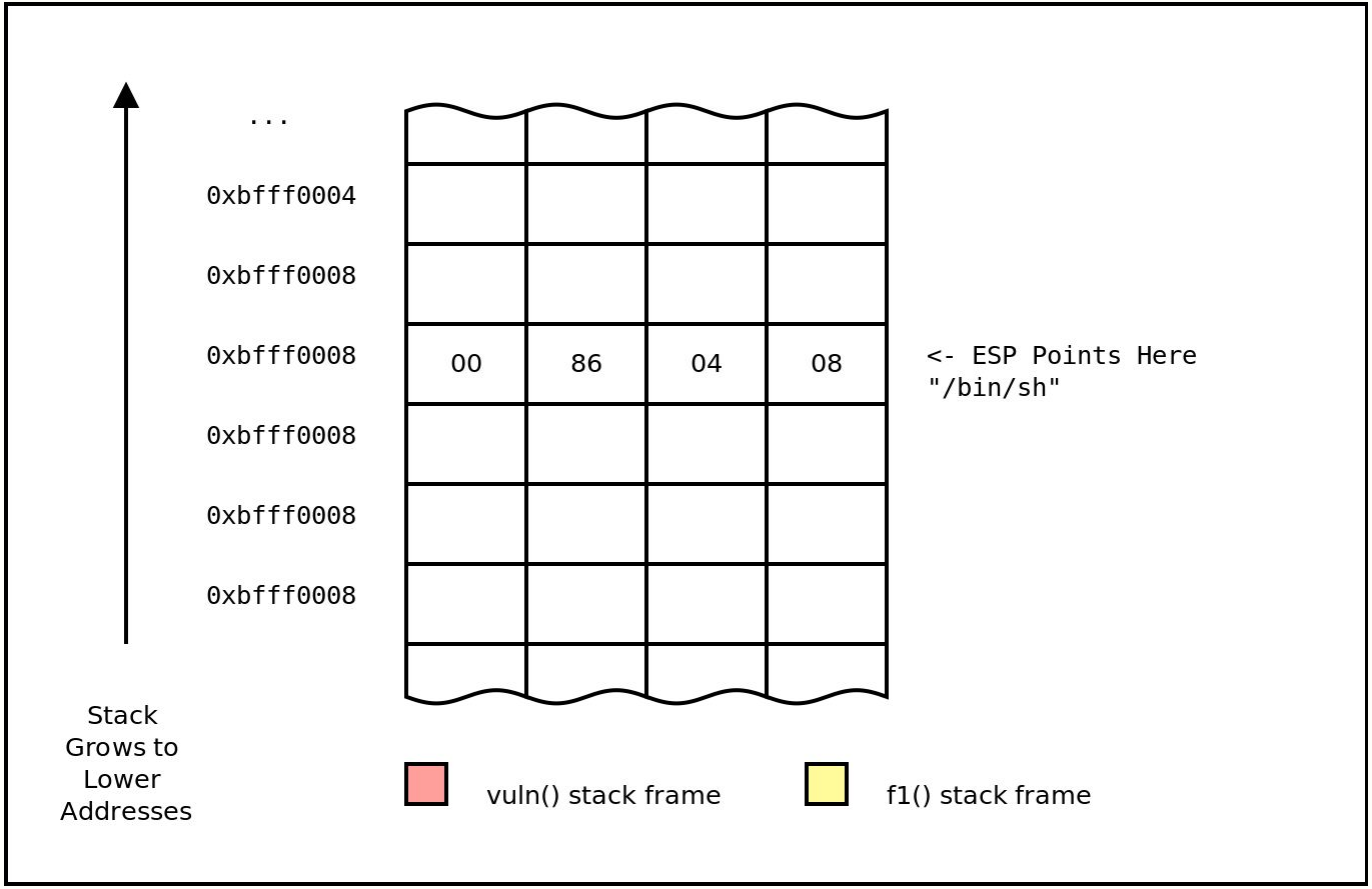
Before we proceed, we need a couple of values:

1. Address to Jump To: 0x08048390 (system@plt)
2. Address of “/bin/sh”: 0x8048600


...
0xbfff0004
90
83
04
08
Saved Return Pointer (vuln)
0xbfff0008
41
41
41
41
Saved Return Pointer (f1)
0xbfff0008
00
86
04
08
"/bin/sh" (f1)
0xbfff0008
0xbfff0008
0xbfff0008
Stack Grows to Lower Addresses
vuln() stack frame
f1() stack frame


...
0xbfff0004
90
83
04
08
<- ESP Points Here
Saved Return Pointer (vuln)
0xbfff0008
41
41
41
41
Saved Return Pointer (f1)
0xbfff0008
00
86
04
08
"/bin/sh" (f1)
0xbfff0008
0xbfff0008
0xbfff0008
Stack Grows to Lower Addresses
vuln() stack frame
f1() stack frame


...
0xbfff0004
0xbfff0008
0xbfff0008
0xbfff0008
0xbfff0008
0xbfff0008
41
41
41
41
00
86
04
08
<- ESP Points Here
Saved Return Pointer (f1)
"/bin/sh" (f1)
Stack Grows to Lower Addresses
vuln() stack frame
f1() stack frame


...
0xbfff0004
0xbfff0008
0xbfff0008
00
86
04
08
<- ESP Points Here
"/bin/sh"
0xbfff0008
0xbfff0008
0xbfff0008
Stack
Grows to
Lower
Addresses
vuln() stack frame
f1() stack frame

Objective: Putting It All Together

amon@pwned: ~ $

Fin

Stay tuned for more with ‘Practical Ret2Libc’