INTRODUCTION TO EXPLOIT DEVELOPMENT

Nathan Ritchey and Michael Tucker

Who Am I (Nathan Ritchey)

- Have Bachelors in Computer Science
- Member of CSG
- Working on Masters with focus on Information Assurance
- Some Interests

Who Am I (Michael Tucker)

- Graduate from UTD
- Member of CSG and the CTF team
- Vulnerability analysis for Raytheon

Definitions

- Reverse Engineering
- Vulnerability Analysis
- Exploitation

Reverse Engineering (RE)

- A systematic methodology for analyzing the design of an existing device or system, either as an approach to study the design or as a prerequisite for re-design.

Vulnerability Analysis (VA)

- Vulnerability analysis, also known as vulnerability assessment, is a process that defines, identifies, and classifies the security holes (vulnerabilities) in a computer, network, or communications infrastructure.

Exploitation

- "An exploit (from the verb to exploit, in the meaning of using something to one's own advantage) is a piece of software, a chunk of data, or sequence of commands that takes advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic (usually computerized). Such behavior frequently includes such things as gaining control of a computer system or allowing privilege escalation or a denial-of-service attack." - Wikipedia

What's the Difference?

- Reverse Engineering
  - The act of figuring out the design and implementation of the system.
- Vulnerability Analysis
  - The act of finding flaws and weaknesses in any part of said system.
- Exploitation Development
  - The act of turning said vulnerability into an actual means of compromising the system's confidentiality, integrity, and/or availability.
- Hacking
  - Utilization of the exploit.

The Payoff

- “Turning a software vulnerability into an exploit can be hard. Google, for example, rewards security researchers for finding vulnerabilities in its Chrome web browser. The payouts Google make are in the range of $500 to $3000. However it also runs competitions for security specialists to present exploited vulnerabilities. These exploits are rewarded much larger sums, as much as $60,000. The difference in payouts reflects the magnitude of the task when trying to exploit a vulnerability.”

  -livehacking.com

Legality

- It's alright to develop, but seek legal expertise to implement.
  - Are you connected to the internet?
  - Are you accessing a remote system?
  - Do you have permission to access that system?
- Look at "How to Disclose or Sell an Exploit Without Getting in Trouble" by Jim Denaro

Illegal Examples

- Sony PlayStation 3
- Target
- Heartland
- Home Depot
- Adobe

Pinball on Windows XP

- First hands-on example
  - Reverse Engineer the Pinball game
  - Conduct Vulnerability Analysis
  - Exploit the Pinball Game

More In Depth Example

- Exploitation
  - Memory Corruption
  - Buffer Overflow
  - Shell Code
  - NOP Sled

What is Memory Corruption

- **Memory corruption** is one of the most intractable class of programming errors, for two reasons: The source of the **memory corruption** and its manifestation may be far apart, making it hard to correlate the cause and the effect.

Memory Corruption

- Code Injection
  - Where do we inject the malicious code?
  - How should we generate malicious code (Shellcode)?
  - How should we redirect execution flow?

Memory Corruption

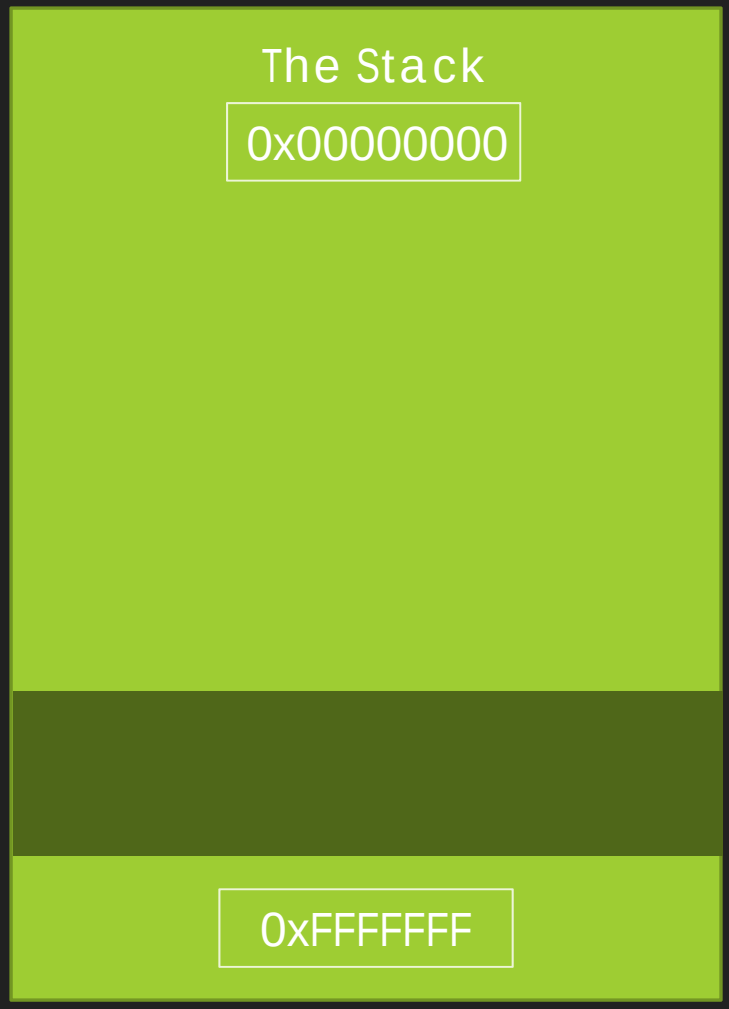
- Redirection of execution flow
  - In x86, one way is to control a register called EIP, also known as the instruction pointer register.
  - This register is how the x86 architecture knows which instruction to run next.
  - EIP, however, is not directly controlled by the user.
- But how does one control EIP?
  - With a vulnerability of course!

Buffer Overflows

- Any instance where a program writes beyond the end of the allocated memory for any buffer.
  - A perfect example can be shown with strcpy() stack overflow.
  - gets() and read() are other examples

Stack Overflow

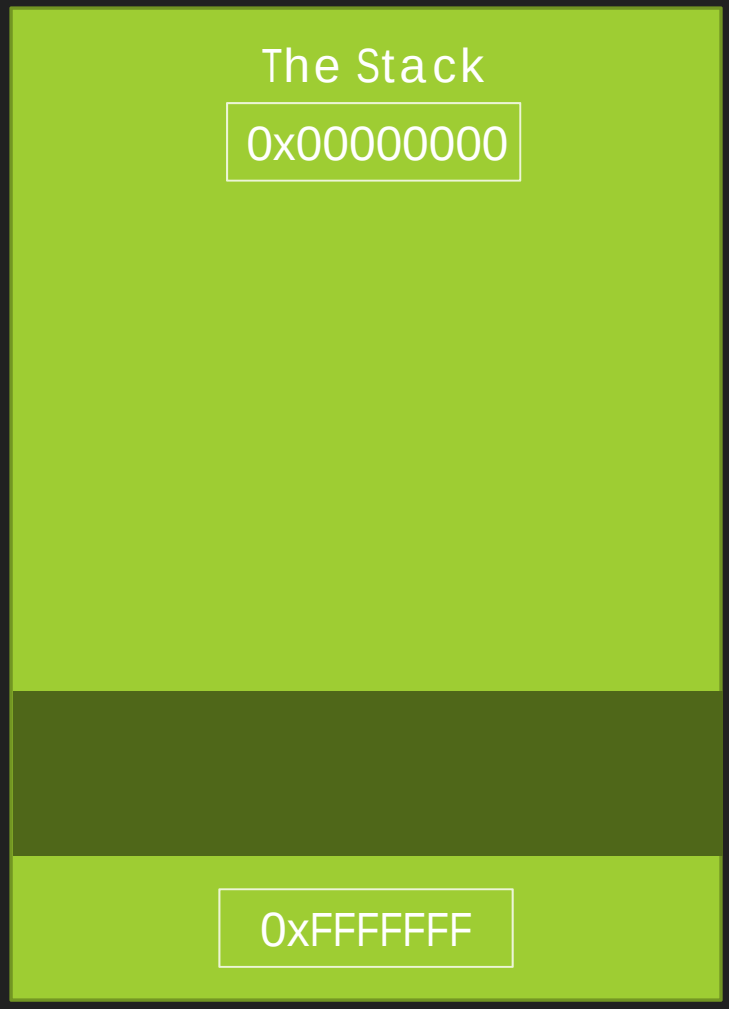
```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```



Stack Overflow

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

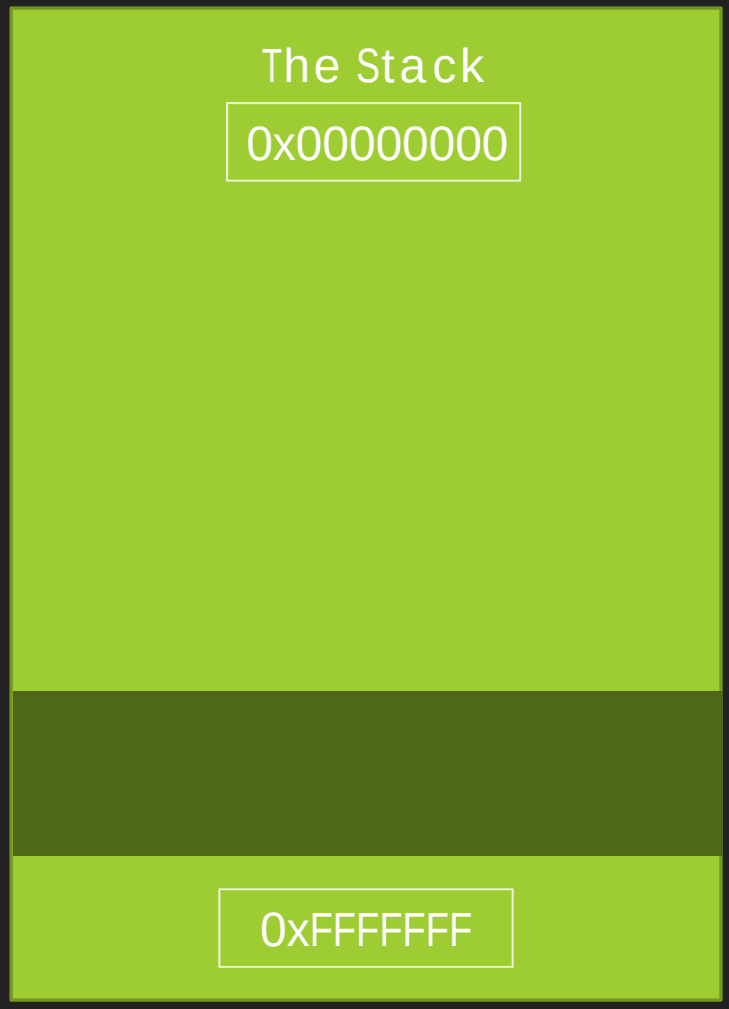
1st Step: Mark controlled input



Stack Overflow

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

2nd Step: Mark Vulnerable code

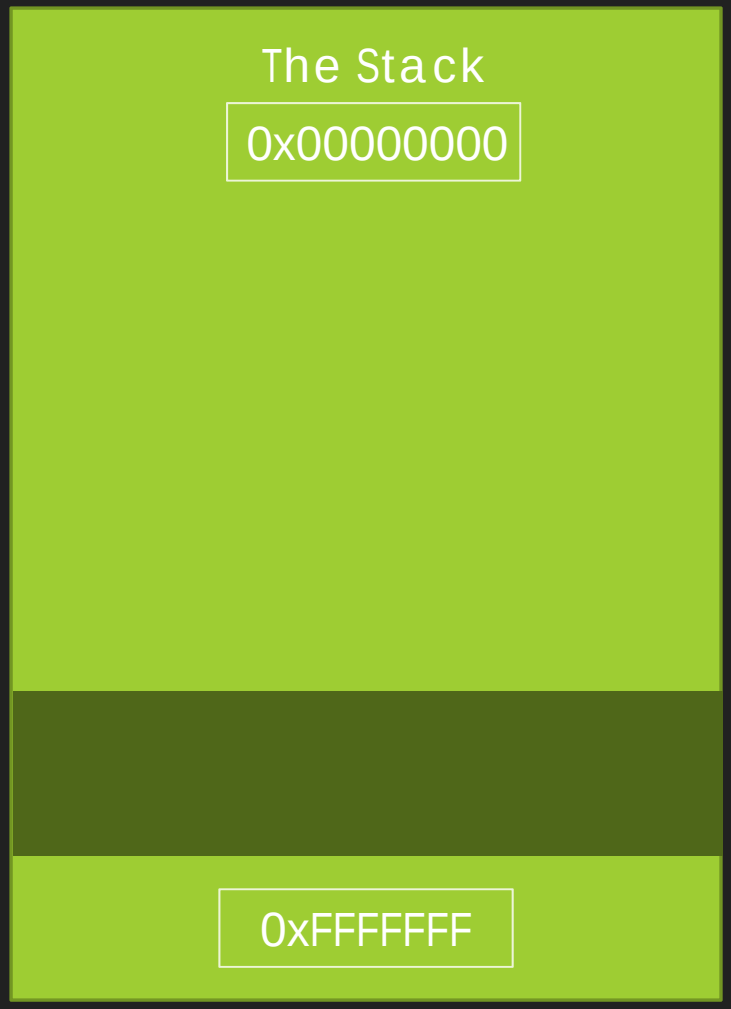


Stack Overflow

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

ESP ->
EBP ->

Last Step: Analyze!

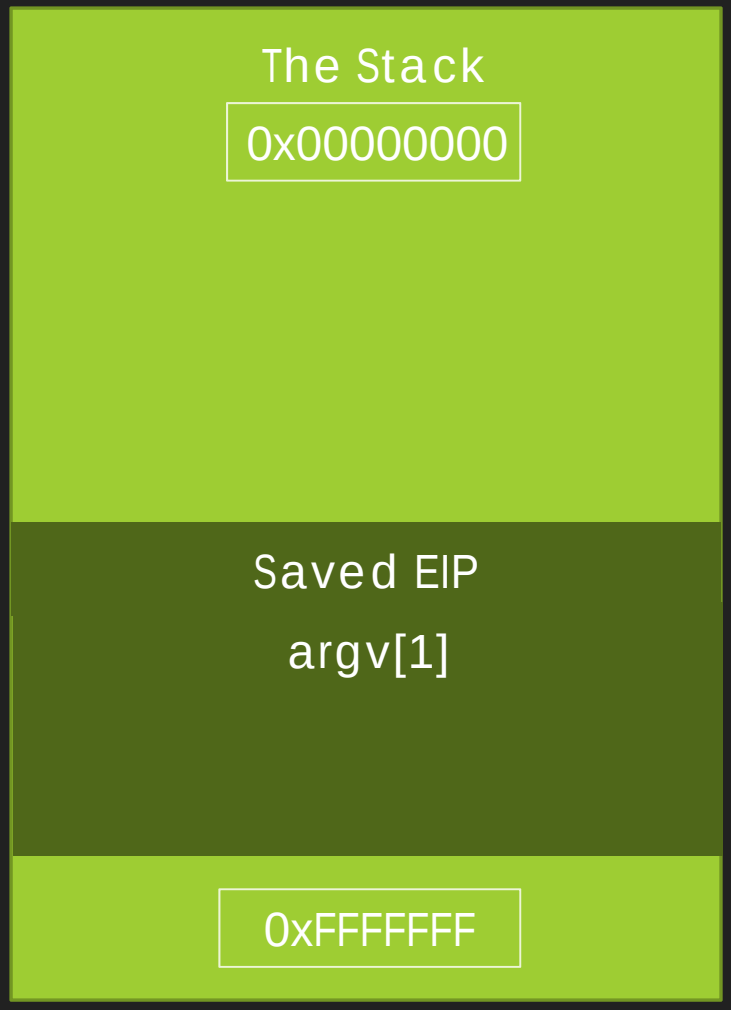


Stack Overflow

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

ESP ->

EBP ->



Stack Overflow

```
#include <string.h>
void do_something(char *Buffer)
{
    char MyVar[100];
    strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
    do_something(argv[1]);
}
```

ESP ->

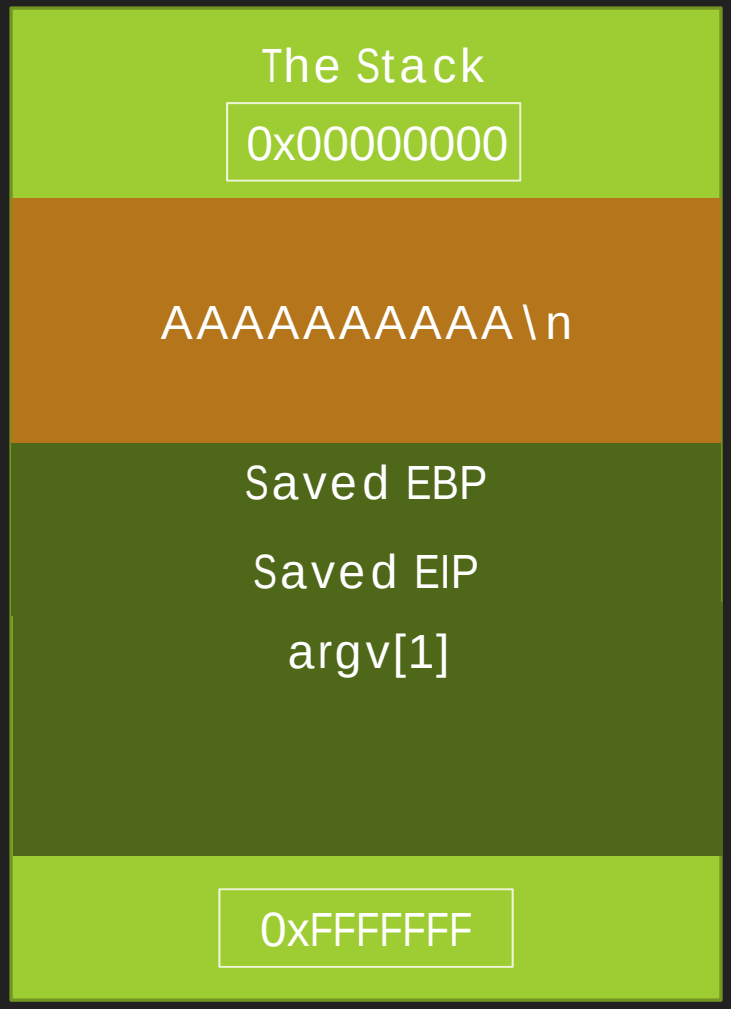
EBP ->

Stack Overflow

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

ESP ->

EBP ->



Case 1: Input “A” ten times

Stack Overflow

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

ESP ->

EBP ->

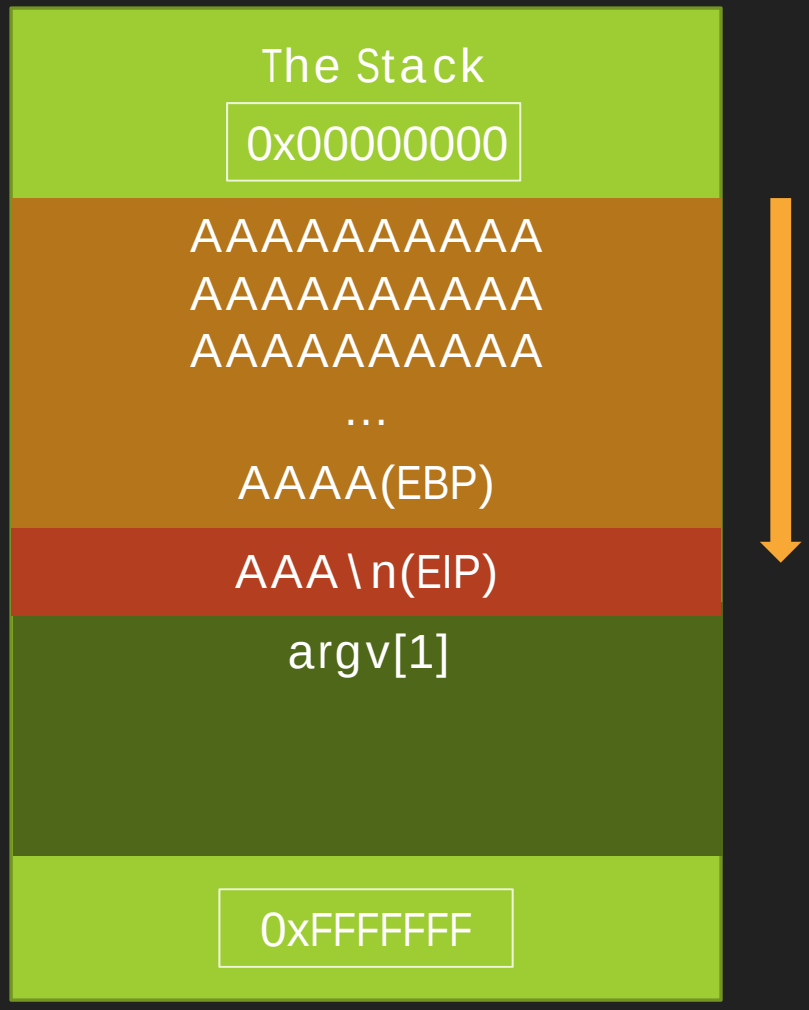
Case 2: Input “A” 103 times

Stack Overflow

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

ESP ->

EBP ->



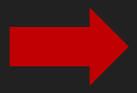
Case 3: Input “A” 107 times

Stack Overflow

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

ESP ->

EBP ->



EIP Control: But now what?

Stack Overflow Hands-on

- Desktop/Simple/Stack 2(White Box)
- Desktop/Simple/Stack 1(Black Box)
- How much harder is it to do without source code?
- Can you think of other ways to get control?

Shell Code(Code Injection)

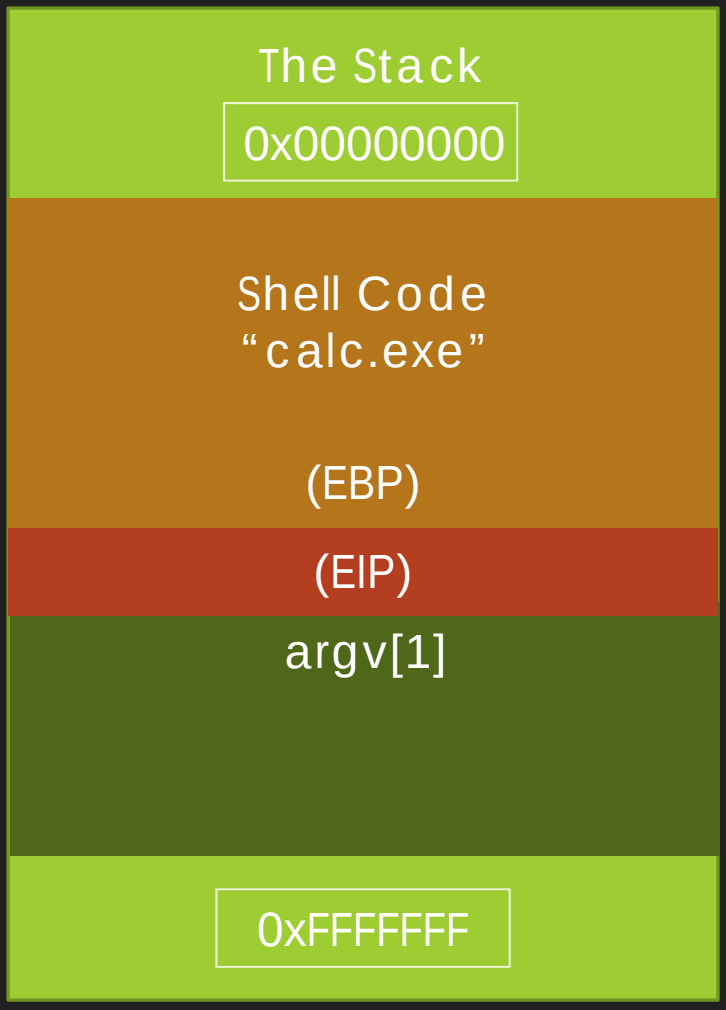
- Machine code used as the payload in the exploitation of a software bug. While in a program flow, shell code becomes its natural continuation.
- Example

Shell Code

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

ESP ->

EBP ->



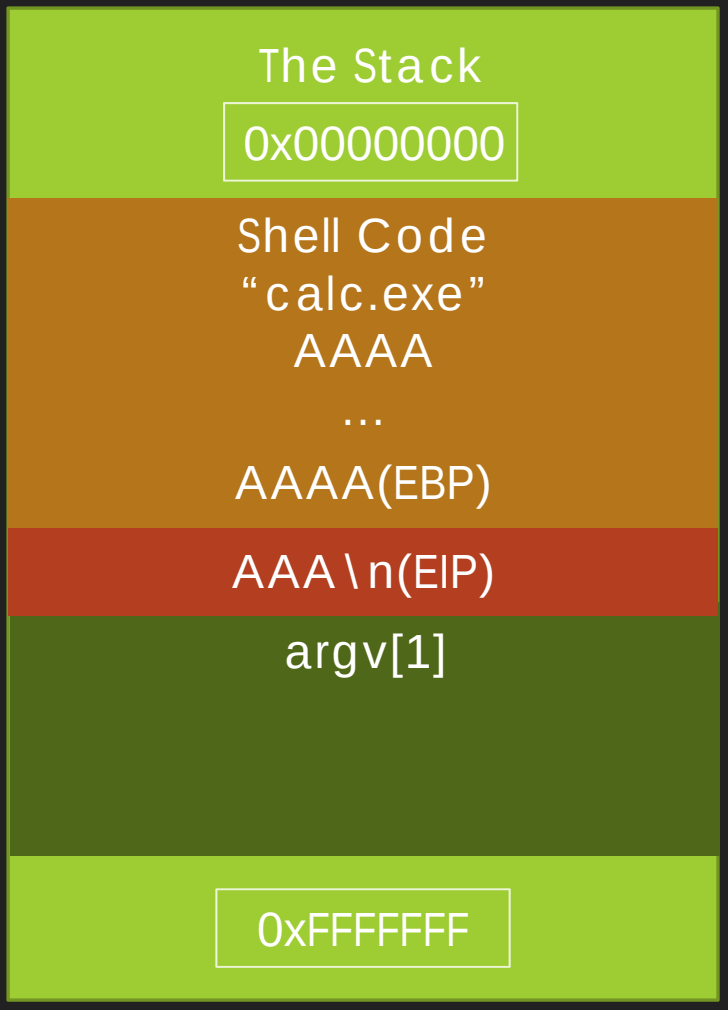
Change: Put Shell Code in place of “A”’s

Shell Code

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

ESP ->

EBP ->

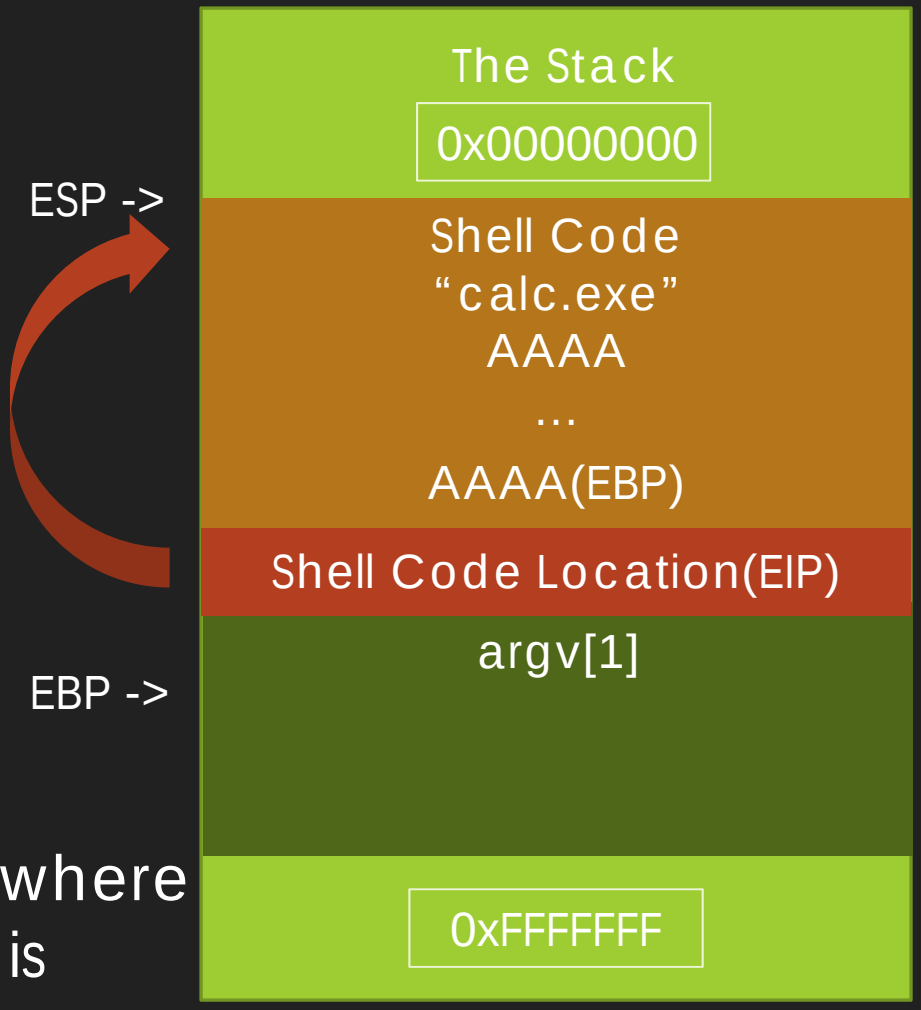


Change: Put padding to still cause overflow

Shell Code

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

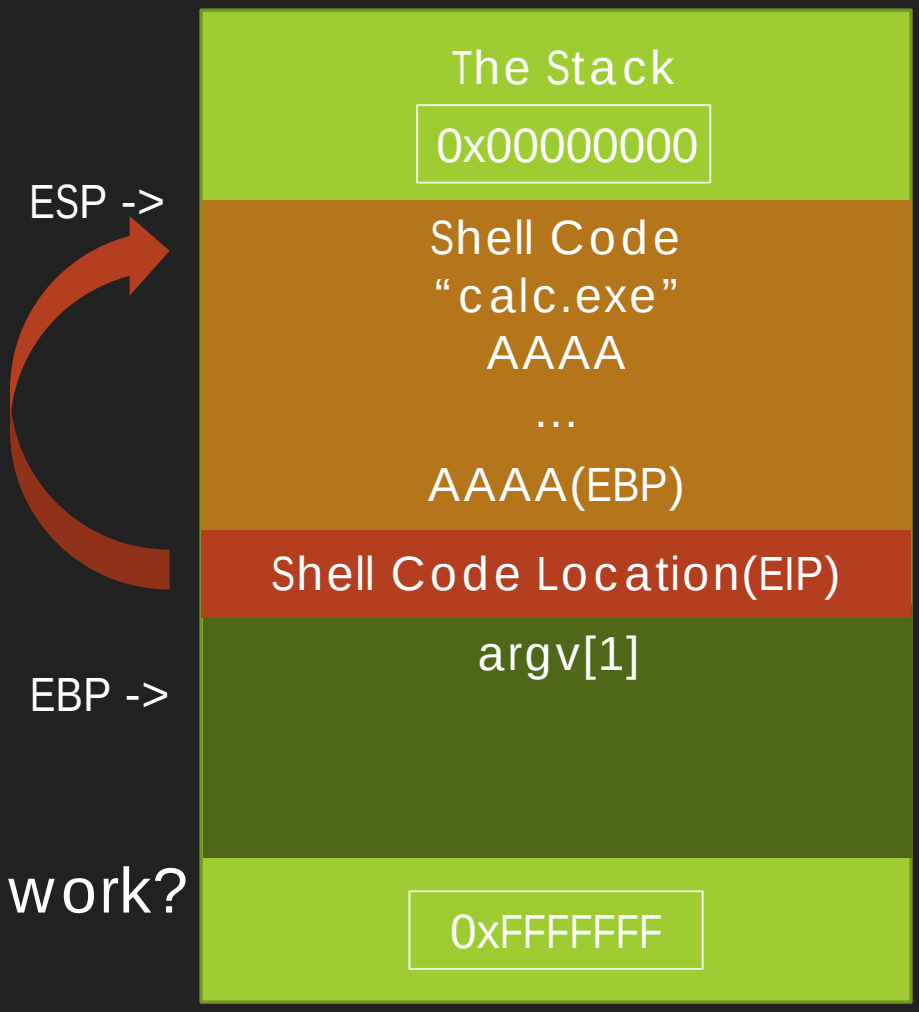
Modify: Change EIP to where the Shell Code is



Shell Code

```
#include <string.h>
void do_something(char *Buffer)
{
        char MyVar[100];
        strcpy(MyVar,Buffer);
}
 int main (int argc, char **argv)
{
        do_something(argv[1]);
}
```

Problem!: Why won't this work?



Stack Armor

- Windows has a native defense that adds “0x00” to the front addresses in the stack.
- Strcpy, will stop on any “0x00” that is comes across because it is considered end of string.
- This prevents us from just pointing to our shell code!
- Now what?

Gadgets

- **Gadgets** are pieces of code borrowed from the loaded program image or libraries to circumvent the defenses.
- Used heavily in “Return to libc” and “ROP/JOP”
- So all we need is a simple gadget to get us back to our Shell Code!

Gadgets

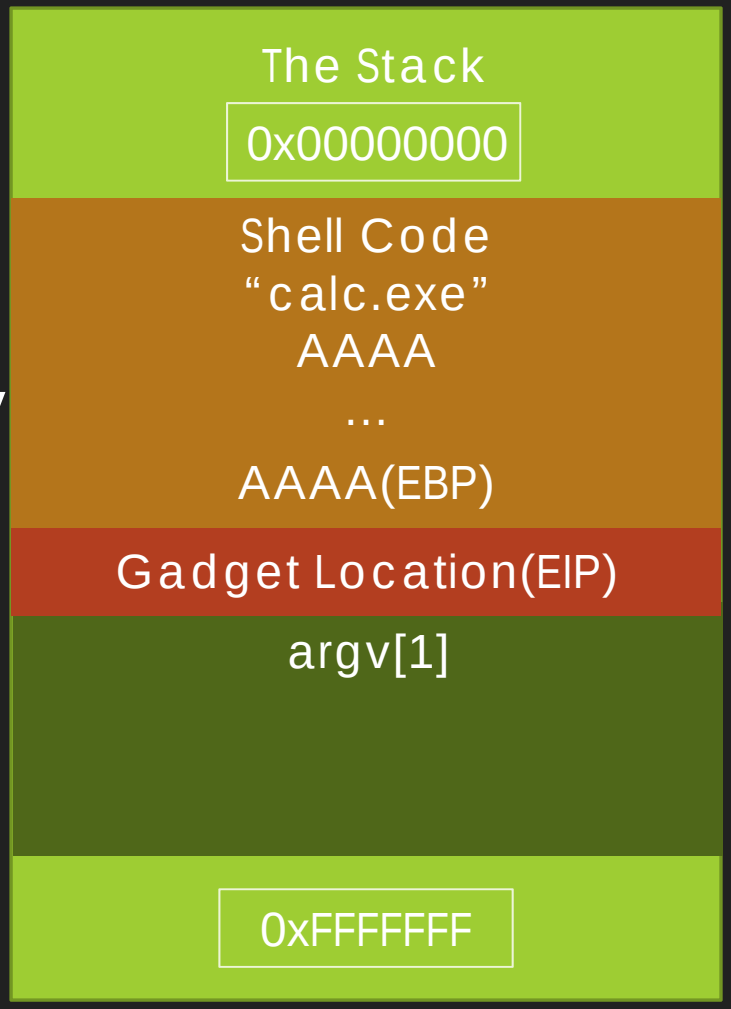
- A simple gadget that we can use is “jmp esp”
- Also known as a “Return to register”.
- This gadget allows us to go to the top of the stack, where our shell code just happens to be located.
- So what we must do is find where a jmp esp is and then have EIP pointed there.
- How do we find such a gadget though?
- Mona.py from the Immunity Debugger can help us here!
  - !mona jmp –r esp

Shell Code with Gadget

Change: Modify EIP to gadget

Possibilities: Found from mona.py

- jmp esp
- call esp
- push esp ; ret(ROP)

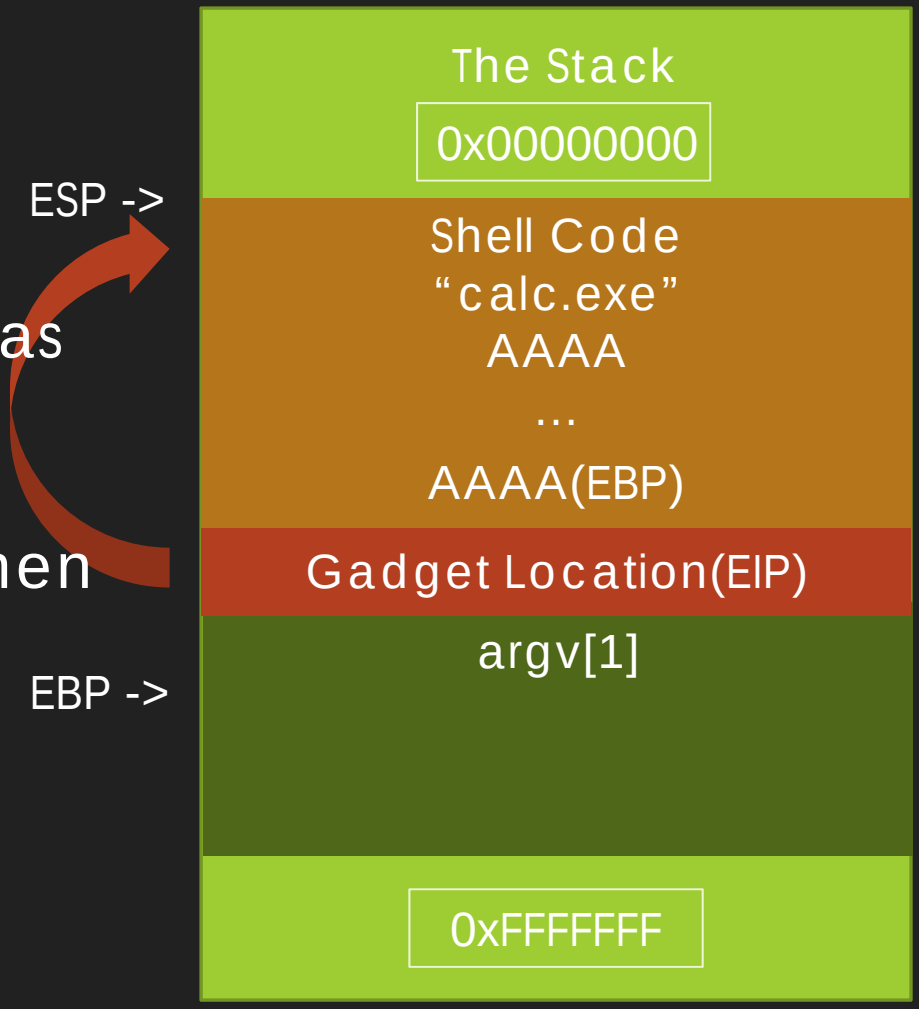


Shell Code with Gadget

Success!

Defeat: The stack armor was defeated using the gadget!

Pwn!: The shell code is then executed



Shell Code Hands-on

- Desktop/Advanced/abo1.exe
- Using your new knowledge of buffer overflows, shell code, and gadgets get "calc.exe" to run by controlling abo1.exe
- One thing to note is not all of the addresses mona.py finds are usable, why?
- How could we improve reliability of our exploits?

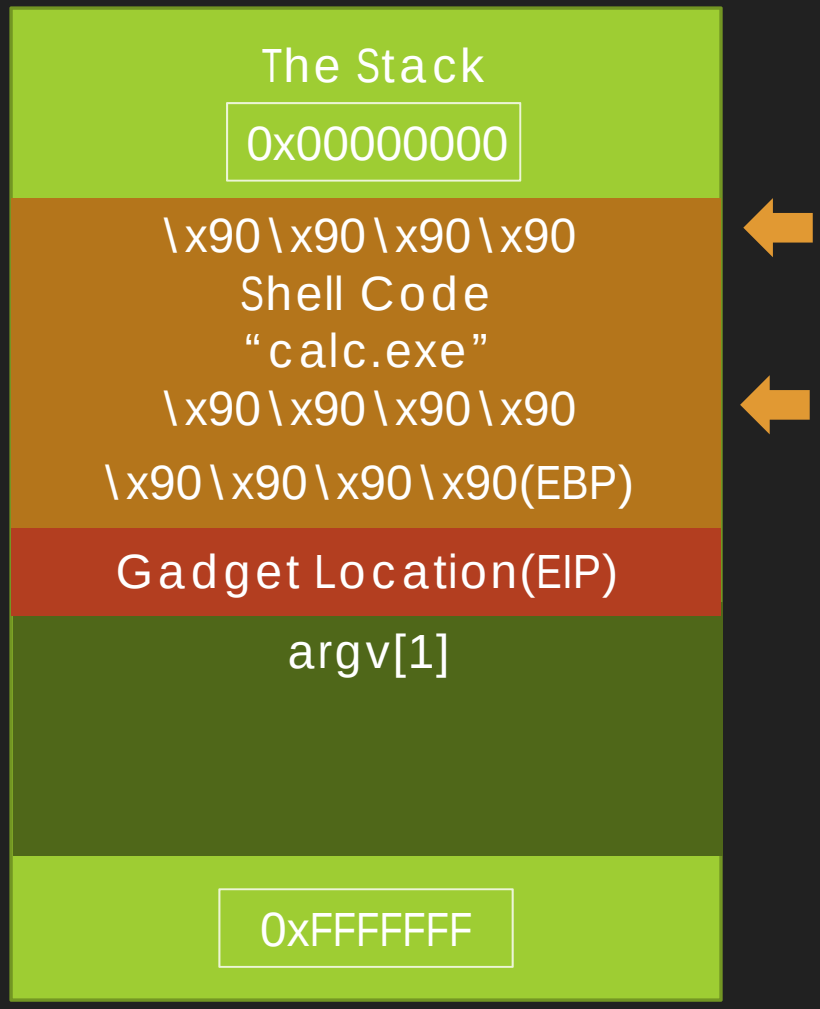
NOP Sled

- Easy to jump to the wrong address where shell code is located.
- The Address can change per system!
- NOP ("no operation") helps with this issue
  - Can jump anywhere in NOP Sled and just slide into the malicious shell code.
  - In x86 this is 0x90

NOP Sled

Change: Add 0x90 before and after shell code

Finalized exploit, with reliability!



Preventing Stack Overflow

- Stack Guard(Stack cookies)
- Stack Shield
- ProPolice
- DEP(W XOR X)
- ASLR

Easy RM to MP3 Converter

- Going to use knowledge of buffer overflows in a practical example.
- Goals:
  1. Figure out what files the converter can take
  2. Crash the Converter using malicious input within the files you've scoped.
  3. Take control and execute the "calc.exe" shell code!
- ~Hints~
  1. Sometimes not all gadgets will work.
  2. Mona.py/Immunity is your friend, use it!

The Game of Defenses

- So what does one do when there are so many defenses in place?
- Defeat them one at a time of course!
  - Sadly we do not have enough time to show how to defeat all defenses, but at least there's time for one more.

Stack Cookie

- Stack cookies are a defense in which in the case that a buffer overflow were to occur, the canary would trip a function call into preventing the vulnerability from happening.
- In other words, it's like a trip-wire mechanism.

Stack Cookie

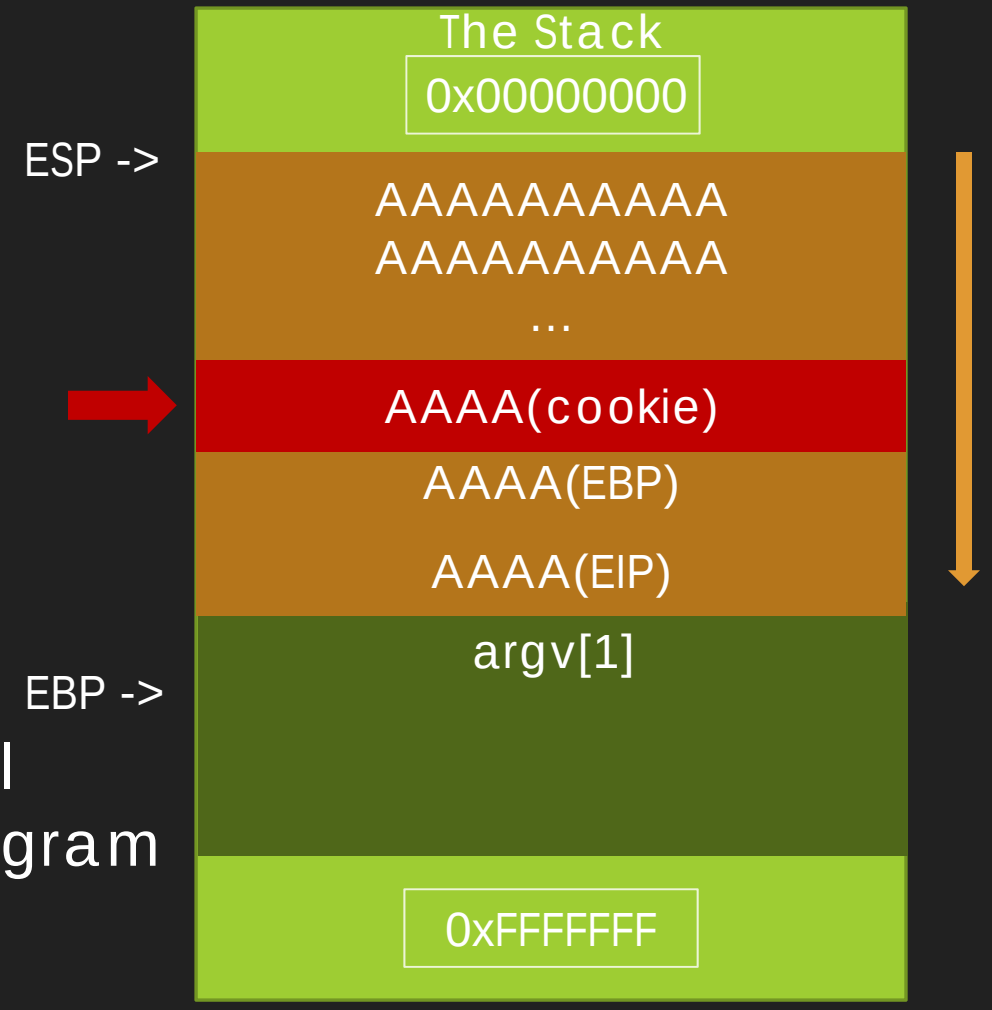
Change: Now there's a cookie in the stack.

What happens if we try to overflow MyVar again?

Stack Cookie

Uh Oh!: We did our buffer overflow, but the cookie also got overwritten.

Failed: The stack cookie will now cause the program to exit.



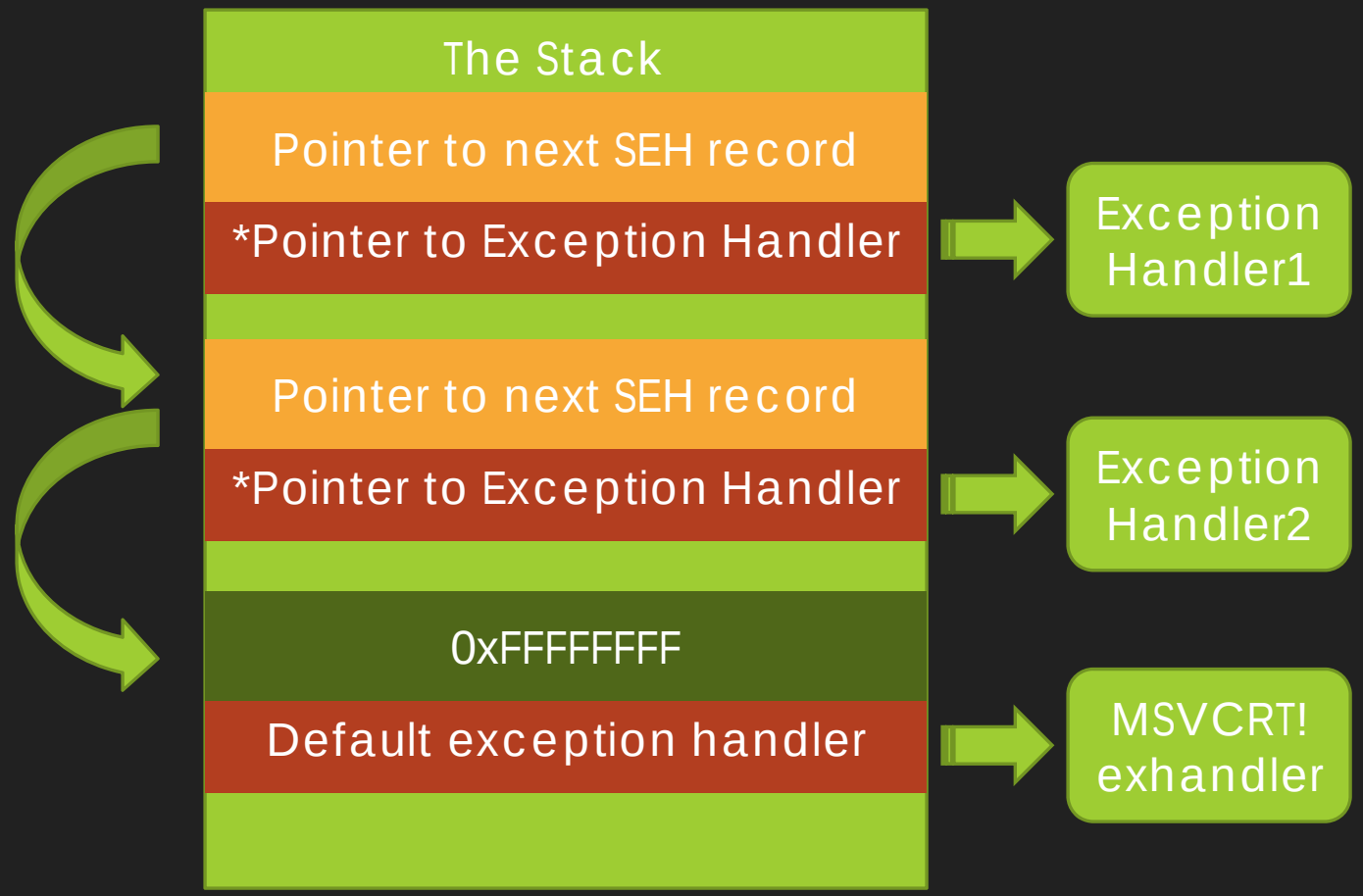
Stack Cookie By-Pass

- So now what? The cookie has foiled our malicious plans of running calculator!
- Well it just so happens that there's not only one way to control the flow of code in a program.

SEH, Exception Handlers

- It just so happens that below us in the stack are exception handler chains.
- Exception handlers are special subroutines called into execution when exceptions occur during the state of the program.
- Some examples would be division by zero or out of memory conditions.

Exception Handler Chain



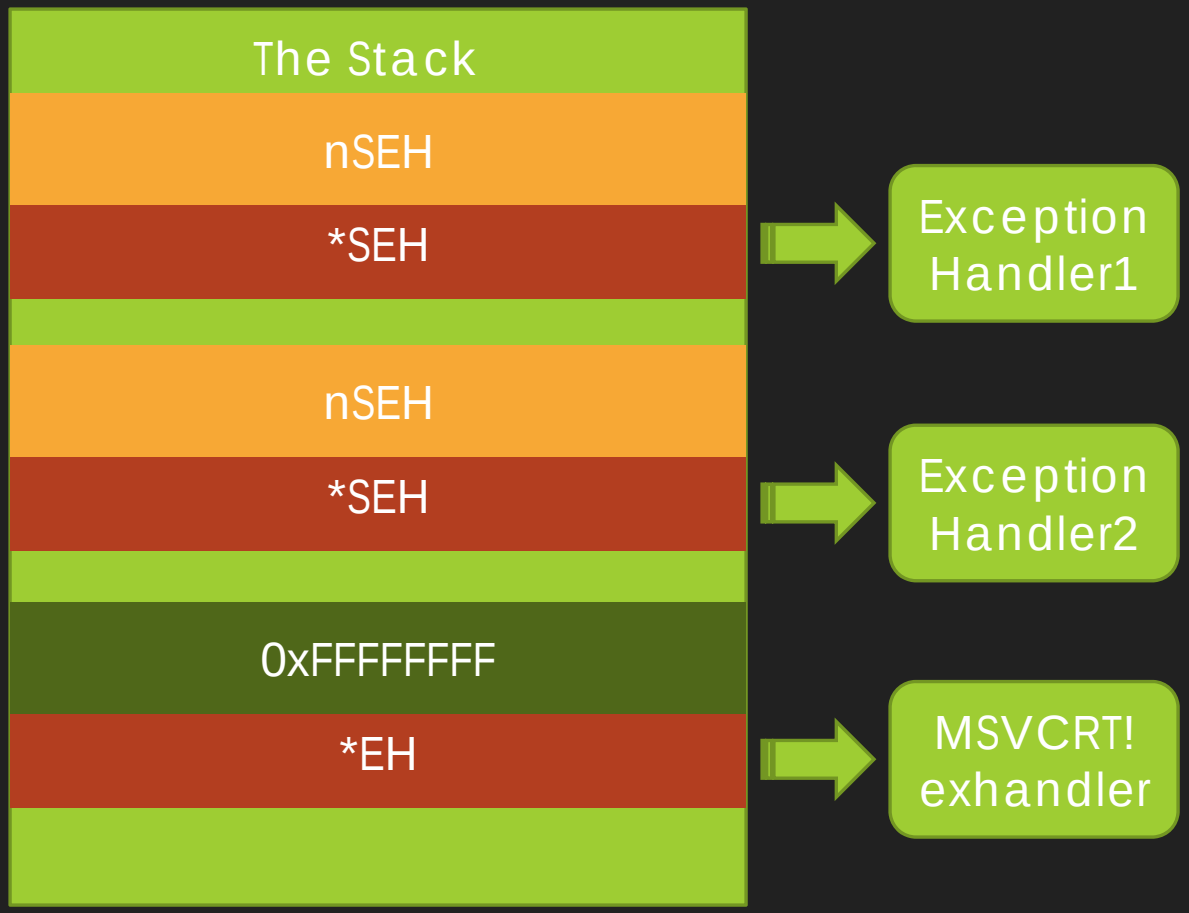
Exception Handler Chain

So what is our goal?

Well it just so happens that if you control *SEH, you once again control the flow of the program(EIP).

But how does one do that?

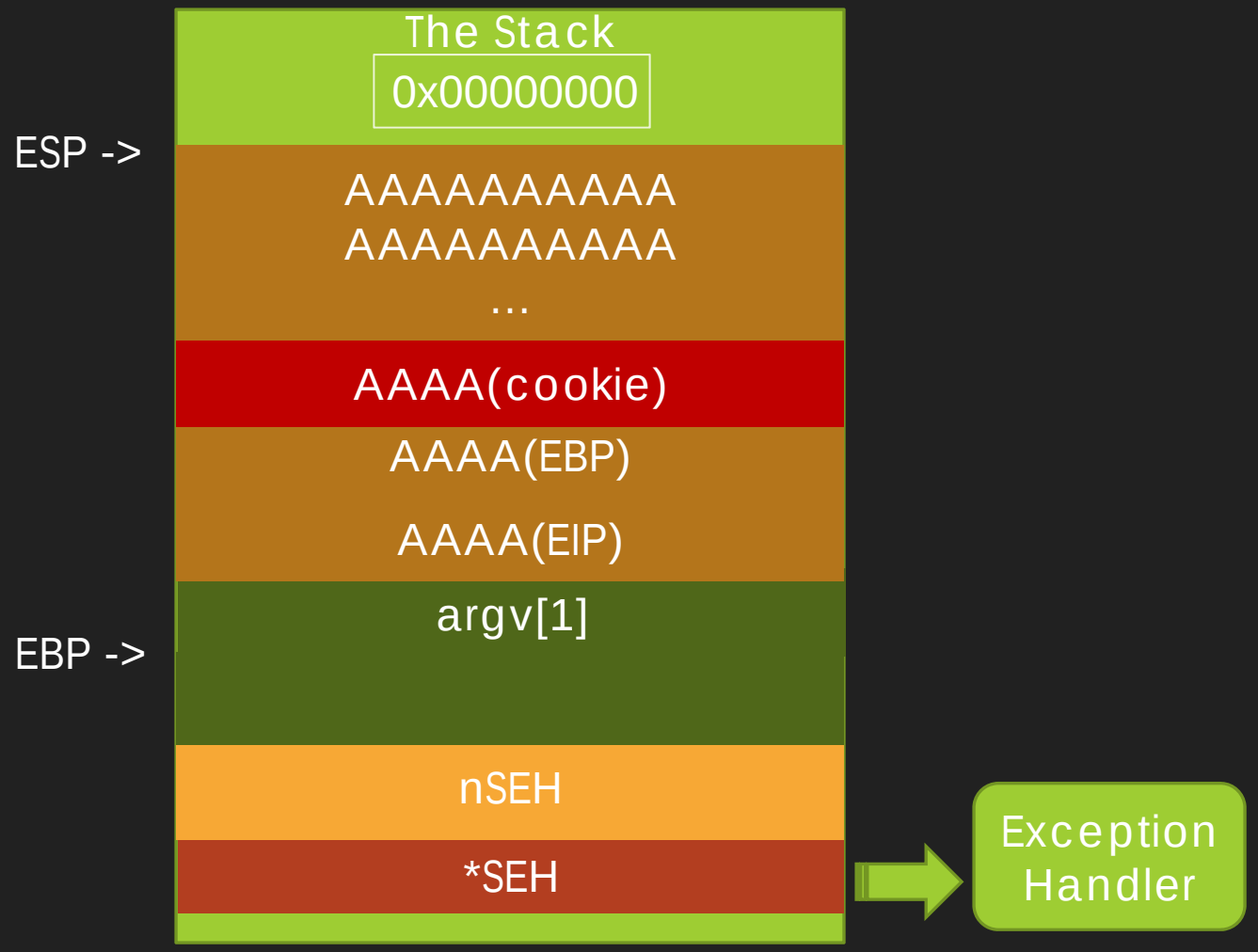
With a vulnerability of course!



Stack Cookie By-Pass

Change: Now let's add the chain to the stack.

Let's continue our vulnerability by continuing the overflow even further than before.

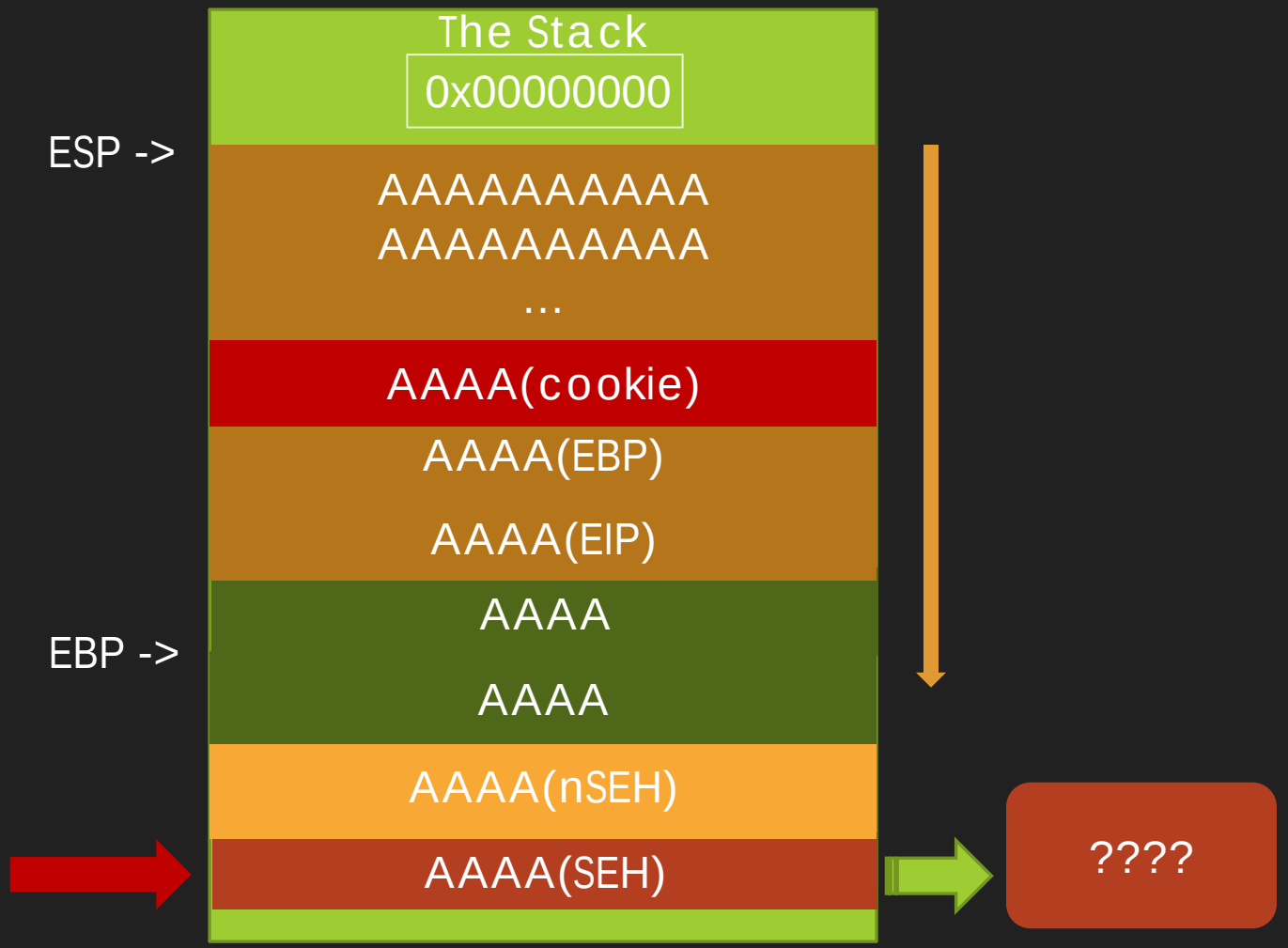


Stack Cookie By-Pass

Flow Hijack: We now control the SEH's pointer!

Using this we can now use a gadget to get back to Shell Code.

Let's modify our exploit a bit.



Stack Cookie By-Pass

Change: We moved the shell code down below our SEH chain.

So what kind of gadget do we want in this case?

Well Pop Pop Ret will do!

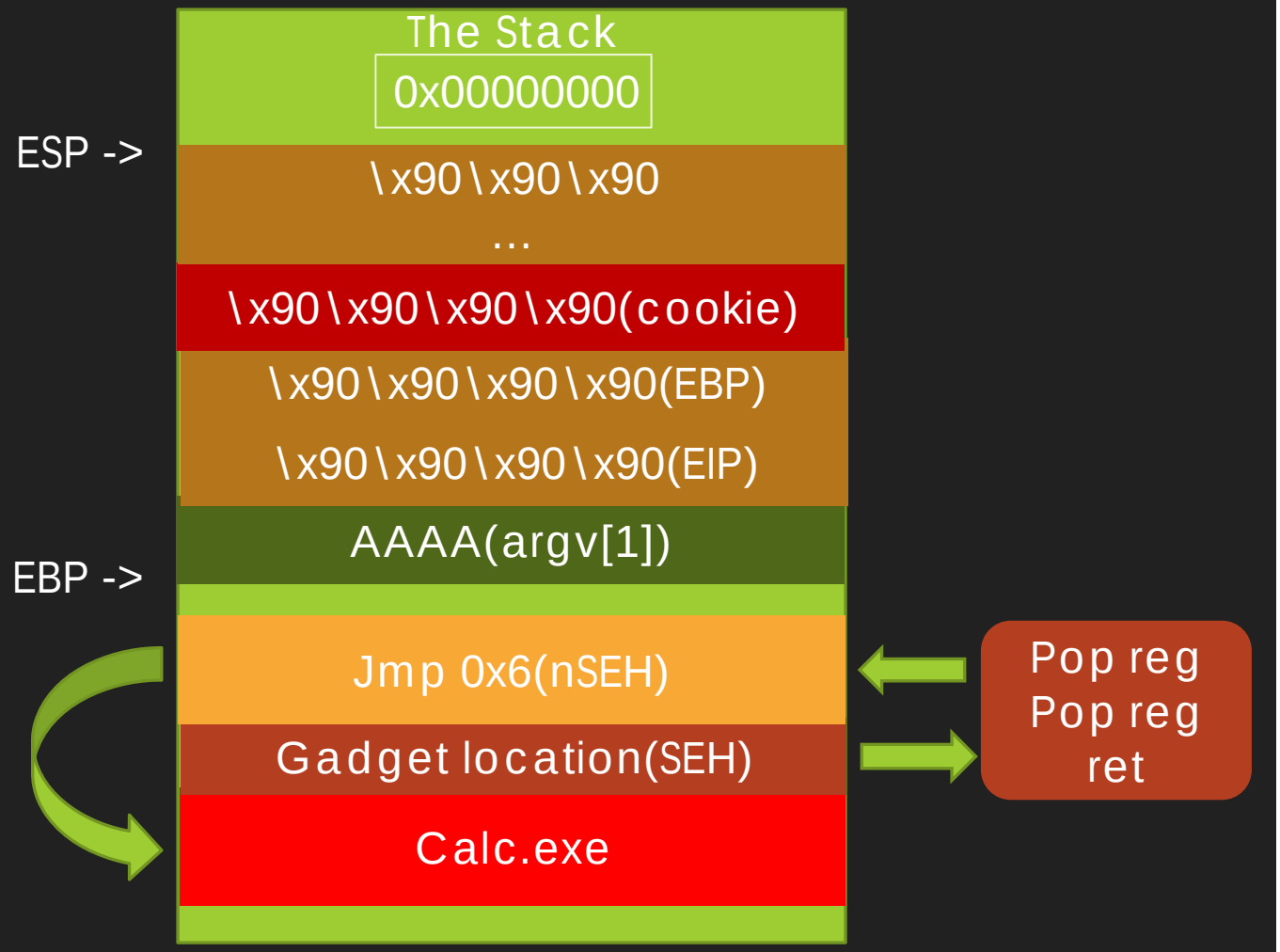
Pop reg
Pop reg
ret

Stack Cookie By-Pass

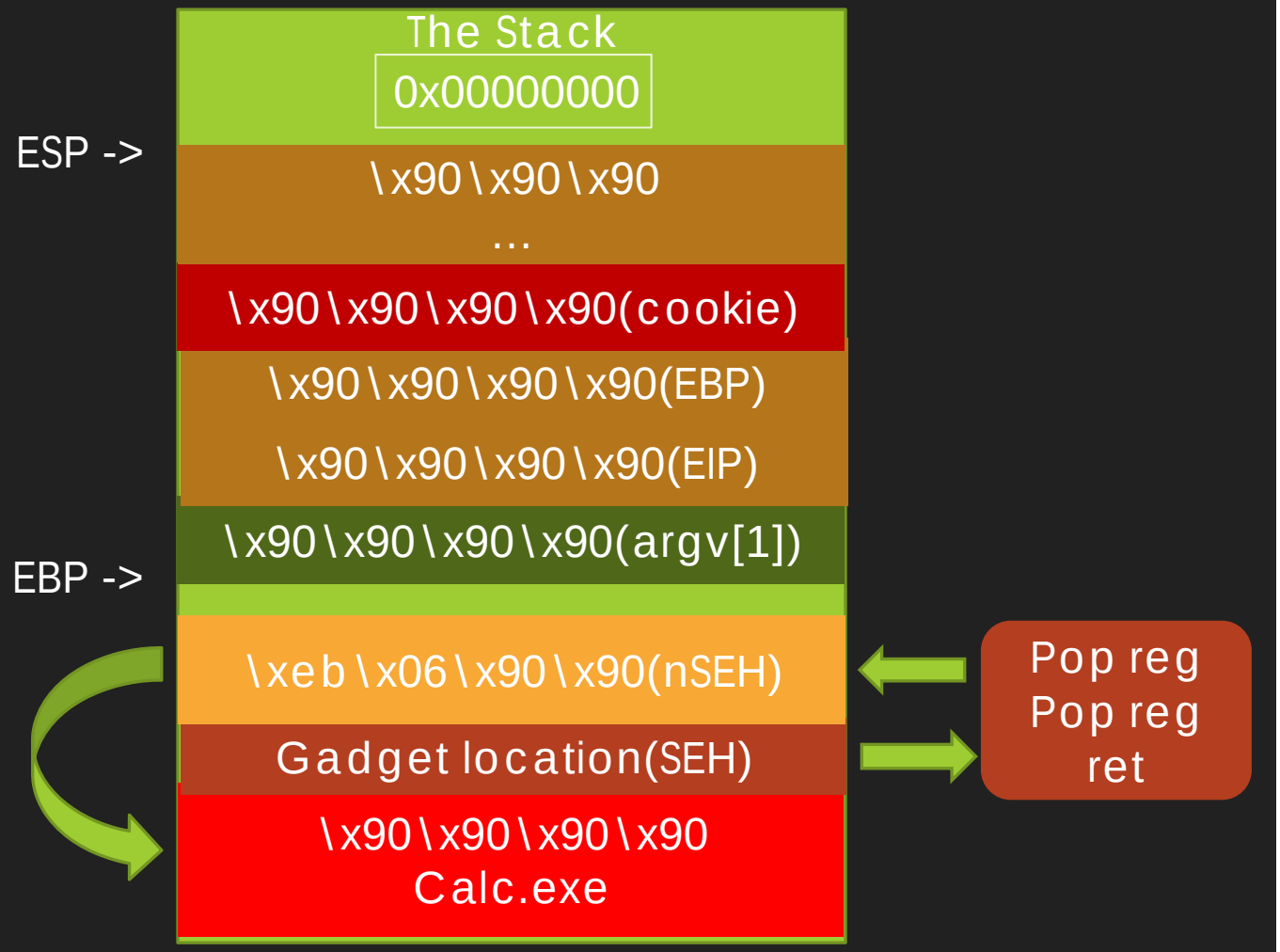
Using “pop pop ret” will let us move back to the nSEH in the stack.

If it just so happens that if we replace nSEH with “jmp 0x6”

We then get to hop exactly six places foward, and run calc.exe!



Final Exploit



DVD X Player 5.5 Pro

- Stack Cookie and Armor? Oh my!
- Can you get around it?
- ~Hints~
  1. For an Exception handler to trigger you must cause an exception in the first place.
  2. Mona.py has a command that may help in this case!

Questions?

Mona.py cheat sheet

- Mona's Help command:
  - !mona help
- Create a pattern: !mona pattern_create <size>
  - !mona pattern_create 512
- Find offset in pattern: !mona pattern_offset <hex>
  - !mona pattern_offset 41314132, finds the offset in the pattern of A1A2
- Find all jump based gadgets(jmp esp, push esp retrn):
  - !mona jmp –r esp, finds all jump gadgets for the register esp.
- Find all seh gadgets(pop pop retn):
  - !mona seh