Ethical Student Hackers

Introduction to Binary Exploitation

SHEFFIELD Ethical Student Hackers
Breaking into security.

The Legal Bit

- The skills taught in these sessions allow identification and exploitation of security vulnerabilities in systems. We strive to give you a place to practice legally, and can point you to other places to practice. These skills should not be used on systems where you do not have explicit permission from the owner of the system. It is VERY easy to end up in breach of relevant laws, and we can accept no responsibility for anything you do with the skills learnt here.

- If we have reason to believe that you are utilising these skills against systems where you are not authorised you will be banned from our events, and if necessary the relevant authorities will be alerted.

- Remember, if you have any doubts as to if something is legal or authorised, just don't do it until you are able to confirm you are allowed to.

SHEFFIELD Ethical Student Hackers
Breaking into security.

Code of Conduct

- Before proceeding past this point you must read and agree to our Code of Conduct - this is a requirement from the University for us to operate as a society.
- If you have any doubts or need anything clarified, please ask a member of the committee.
- Breaching the Code of Conduct = immediate ejection and further consequences.
- Code of Conduct can be found at https://shefesh.com/downloads/SESH%20Code%20of%20Conduct.pdf

SHEFFIELD Ethical Student Hackers
Breaking into security.

Before We Begin

- This session has previously been run as 2 parts
- Therefore condensing it to 1 has required some assumption of knowledge
- So if something isn't clear, please ask!

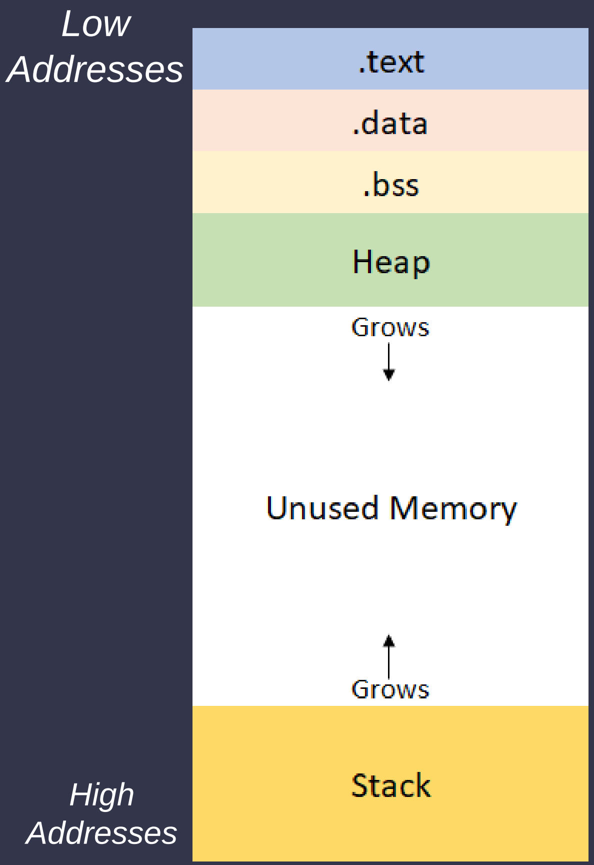
Memory

Memory - Overview

- When we talk about memory, we mean RAM not storage
- When a binary is executed it needs to be loaded into memory
- Stores instructions and data
- We address with hex, e.g 0xbfab15ce
- Operating system maps virtual memory onto physical
- Remember at the end of the day,
- Data, instructions, are all just 0s and 1s (we'll look at them in hex)
- EVERYTHING is just data (example later)

Memory - Layout

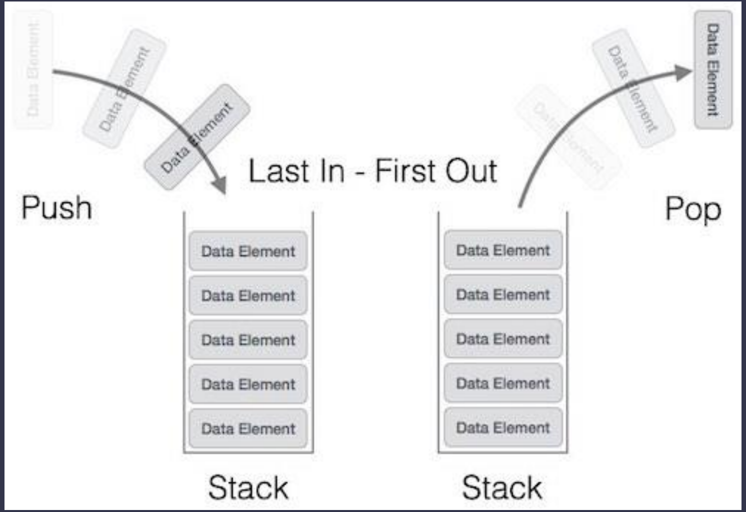
- Process memory is segmented into various sections
- .text - Basically where a programs code is
- .data - Initialised global variables
- .bss - Uninitalised global variables
- The heap - Where dynamically allocated memory goes
- The stack - Contains details about subroutines (functions) of a program



Ethical Student Hackers
SHEFFIELD
Breaking into security.

Memory – The Stack

- Used for data in program functions
- Last in First out (LIFO) structure
- Grows from high memory to low
- Stores local variables
- Made up of various "stack frames"
- The exploits we are looking at use the stack

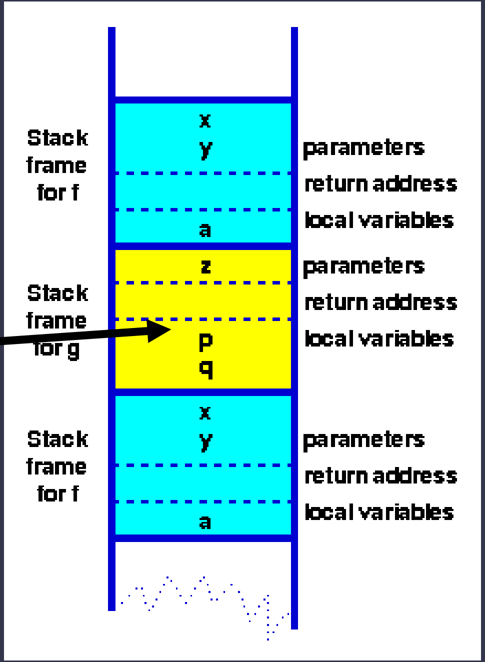


https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm

Memory – Stack Frames

- Collection of data for one function call
- Includes - return address, argument variables, local variables, saved registers
- The important parts for us
- Local variables and the return address BOTH reside on a stack frame
- The return address ends up in the EIP register!

Note: LIFO means p was declared BEFORE q



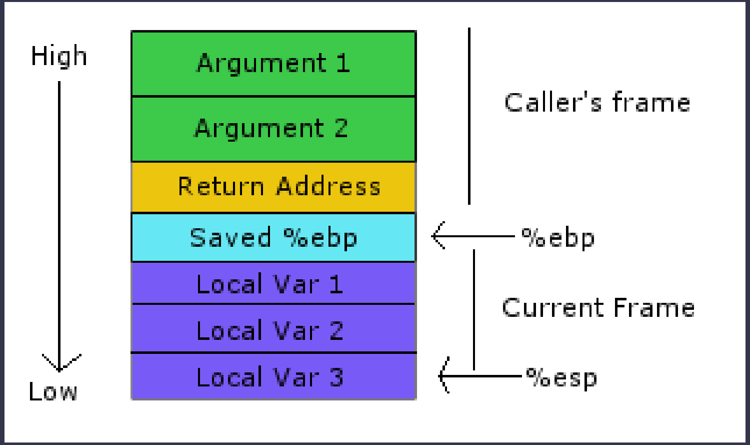
https://www.cs.auckland.ac.nz/software/AlgAnim/stacks.html

Note this graphic, and the next few have high addresses at the top, not bottom

SHEFFIELD Ethical Student Hackers
Breaking into security.

Memory – Stack Frames in Practice

- The local variables are on the stack frame
- Along with the return address
- Return address ends up in EIP register (The one that controls execution)
- Remember, data will be PUSHED onto the stack
- E.g arg1 went on first, local var 3 last
- Can anyone guess the potential problem here?

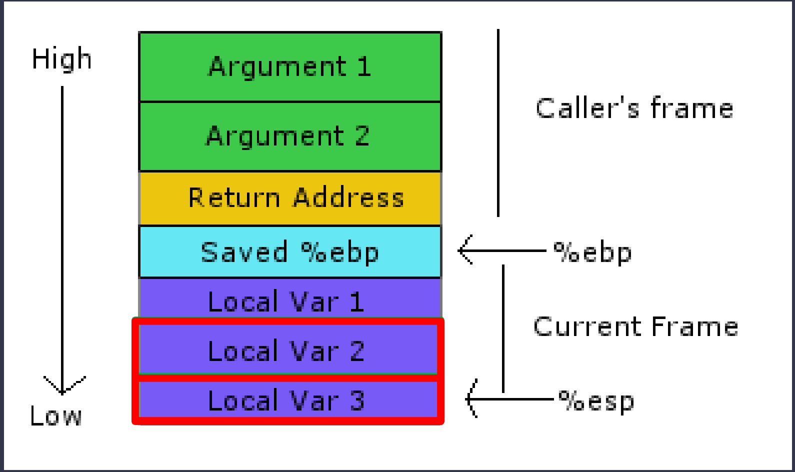


https://i.stack.imgur.com/XDoh3.png

Stack Overflows

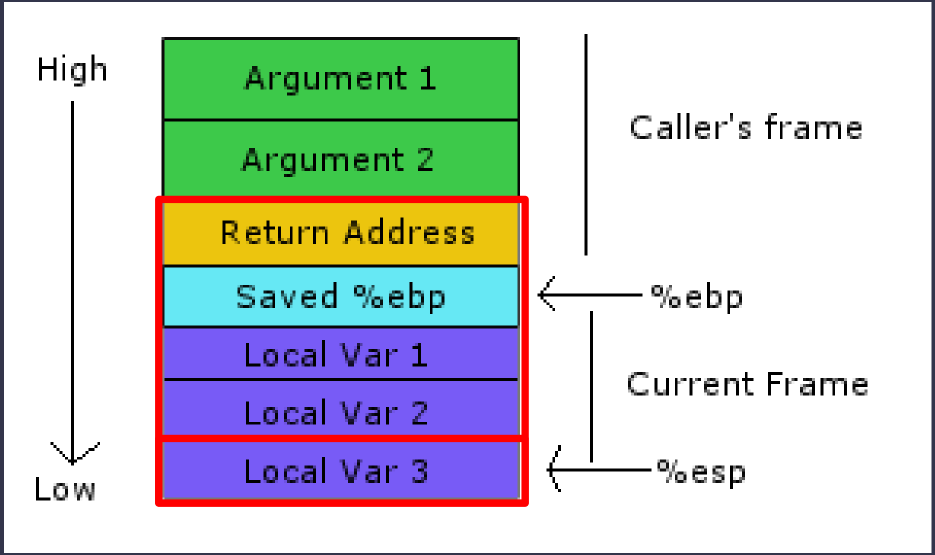
Stack Overflows – It Begins

- This first pushes space (4 bytes) for the int to the frame
- Then 16 bytes for the char buffer
- What if we give it more than 16 bytes?
- If we imagine that "number" is "local var 2"
- And "buf" is "local var 3"
- What if we give "buf" 20 bytes of data?
- It will ***OVERFLOW*** into the space for "number"
- Anyone see where this is going?



Stack Overflows - Gaining Control

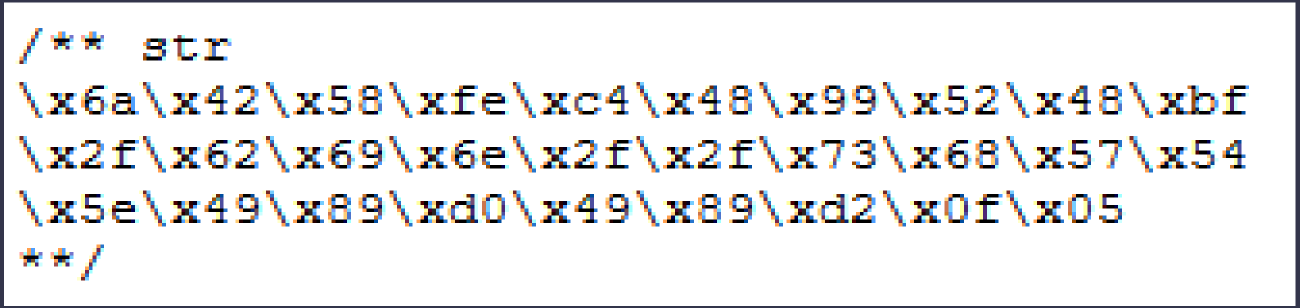
- If we can control EIP
- We can control the next instruction
- And redirect execution
- EIPs value comes from the stack
- The "Return Address"
- If we overflow a variable enough
- We can overwrite EIP
- And then we own the program
- That is a stack overflow.... But what do we do with it?



Shellcode

Shellcode – What is It?

- Shellcode is the payload we want to execute
- We redirect execution into the shellcode to run it
- Often the hex representation of instructions
- This executes execve to spawn a shell



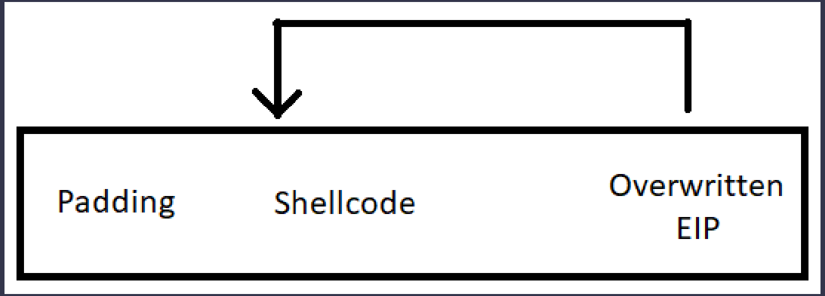
```
/** str
\x6a\x42\x58\xfe\xc4\x48\x99\x52\x48\xbf
\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54
\x5e\x49\x89\xd0\x49\x89\xd2\x0f\x05
**/
```

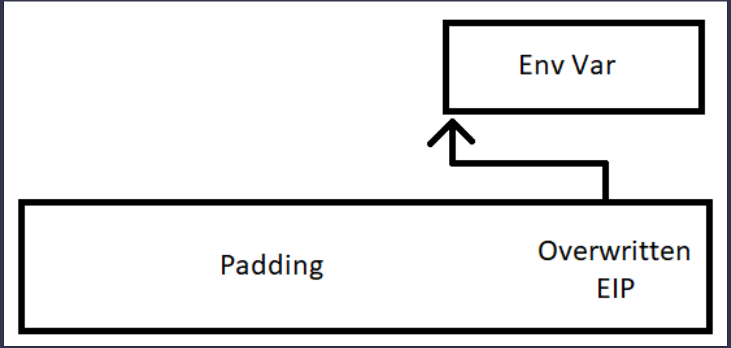
Shellcode – What Can it Do?

- Pop a shell
- Spawn a reverse shell
- Elevate privileges
- Add a user
- Well….anything really
- http://shell-storm.org/shellcode/
- Msfvenom

Shellcode – Where Can it Go?

- There are 2 common places to store shellcode
- In the buffer, before the EIP overwrite
- Where we point EIP back into the buffer
- (Sometimes shellcode goes after EIP)
- Or in an environments variable
- It is possible to locate where an environment variable will be in memory!
- https://github.com/Partyschaum/haxe/blob/master/getenvaddr.c
- Then point EIP at the environment variable

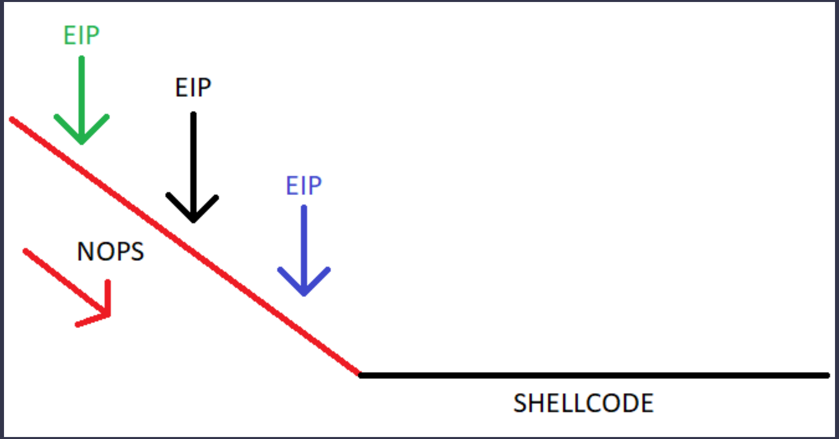




Sheffield Ethical Student Hackers
Breaking into security.

Shellcode – NOP Sleds

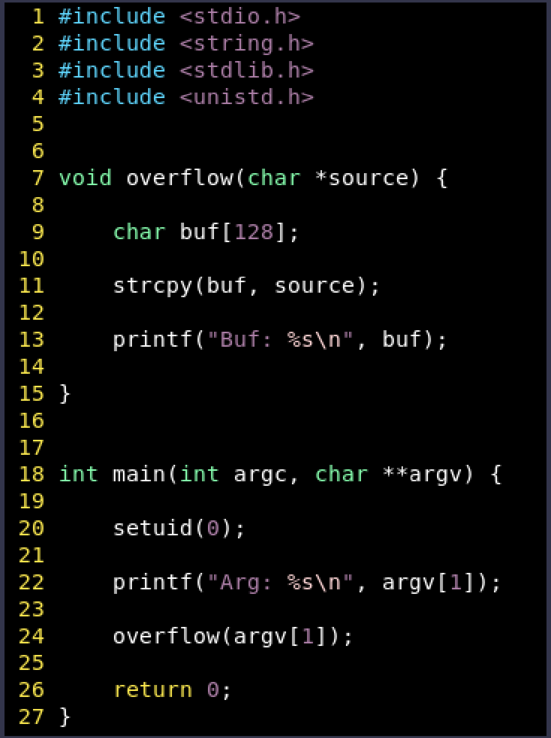
- It can be hard to get exact How can we account for this
- Enter the NOP
- Just moves onto the next instruction
- So use the dif between buf and shellcode len
- Point EIP in the middle
- And hopefully you hit the sled!
- memory locations
- GDB environment != real



Ethical Student Hackers
SHEFFIELD
Breaking into security.

Practical Demo

Demo – The Binary



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


void overflow(char *source) {

    char buf[128];

    strcpy(buf, source);

    printf("Buf: %s\n", buf);

}


int main(int argc, char **argv) {

    setuid(0);

    printf("Arg: %s\n", argv[1]);

    overflow(argv[1]);

    return 0;
}
```

Ethical Student Hackers
SHEFFIELD
Breaking into security.

Demo – Compilation and Setup

- Need to turn off some modern protections
- ASLR needs to be off
- **echo 0 > /proc/sys/kernel/randomize_va_space**
- And compile with
- **gcc vuln.c -o vuln -m32 -fno-stack-protector -z execstack**
- Give it the setuid bit so it can run as root (the setuid function in C earlier)
- **chmod 4777 ./vuln**

- Now lets use this to go from user “lowpriv" to root!
- Demo time
- Shellcode used: http://shell-storm.org/shellcode/files/shellcode-811.php
- And there we go, a simple stack overflow

SHEFFIELD Ethical Student Hackers

Breaking into security.

Any Questions?

www.shefesh.com

Thanks for coming!

SHEFFIELD Ethical Student Hackers

Breaking into security.