Introduction to Reverse Engineering

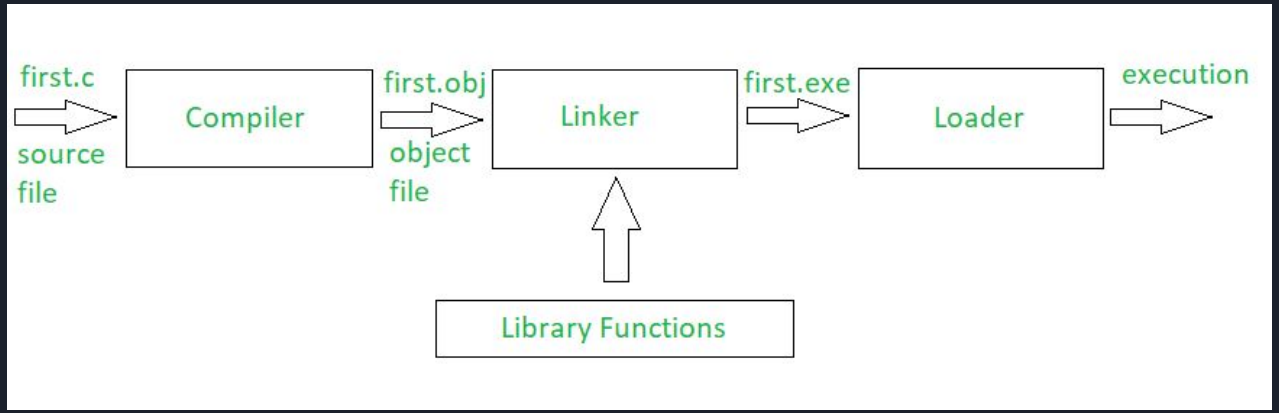
By Peter “Buffer Overflow my Toaster” K

Overview

- First C Program
  - The Bigger Picture
  - Assembly
  - Registers
  - x86 Architecture and Instructions
- Radare2
- Fuck R2 - We’re learning Ghidra
- Buffer Overflows
- Hating yourself for thinking this would be fun

First C Program

- gcc
- build-essential
- A text editor
- A will to live



First C Program - Demo + Debugging

```c
#include <stdio.h>

int main()
{
   int i;

   for(i = 0; i < 10; ++i)
   {
       printf("Alex - HTML isn't real programming\n");
   }

   return 0;
}
```

First C Program - Demo + Debugging - Commands

> gcc firstProg.c
> strings a.out
> objdump -D a.out | grep -A20 main.:
    AT&T syntax
    Memory on the left
> objdump -M intel -D a.out | grep -A20 main.:
    Intel syntax
> gdb -q ./a.out
>> break main
>> run
>> info registers
-> R denotes 64-bit processor
>> set disassembly intel
>> i r

The Bigger Picture

What is *firstProg.c*?

firstProg.c is *source code*, **not** a program. Once compiled, the source code is turned into a binary program that the CPU understands and executes.

Many programmers don’t understand (nor care) about what is happening low level. This creates opportunities for hackers to exploit their programs.

First C Program - Assembly

Type 1

operation <destination>, <source>

```
mov             ebp,esp
```

-> move value of esp into ebp

```
sub             esp,0x08
```

-> subtract 8 from esp and store in esp

Type 2

operation operand

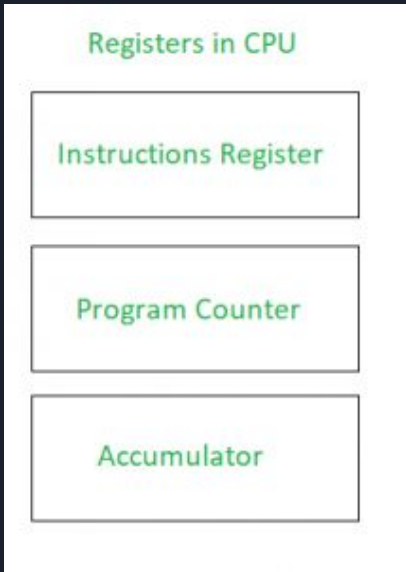
```
pop edx
```

-> remove top value from edx

First C Program - Registers

General Purpose Registers:

- EAX / RAX - Accumulator
- ECX / RCX - Counter
- EDX / RDX - Data
- EBX / RBX - Base
  - Used for variety of purposes, mainly helping CPU execute machine code



Pointers and Indexes (also GP Registers):

- ESP / RSP - Stack Pointer -> points to top of stack
- EBP / RBP - Base Pointer -> offset to access local variables, arguments, etc
- ESI / RSI - Source Index
- EDI / RDI - Destination Index

Pointers

- EIP / RIP - Instruction Pointer -> points to currently executing line of code
- EFLAGS - Bit Flag Register -> stores flags related to program (when cmp'ing for e.g)

First C Program - Flags and Jumps

Jumps:

- JMP - jump to memory address (also function)
- JZ - jump if zero
- JNZ - jump if not zero
- JL - jump less than
- JLE - jump less than or equal to

Flags:

- ZF - zero flag
- SF - signed flag
- OF - overflow flag
- CF - carry flag

First C Program - GCC Debug

> gcc -g firstProg.c
-> compile with debug flag
> gdb --quiet ./.a.out
>> set disassembly intel
-> make it Intel syntax
>> list
-> show debug source code
>> disassemble main
-> spit out assembly

Process Concept

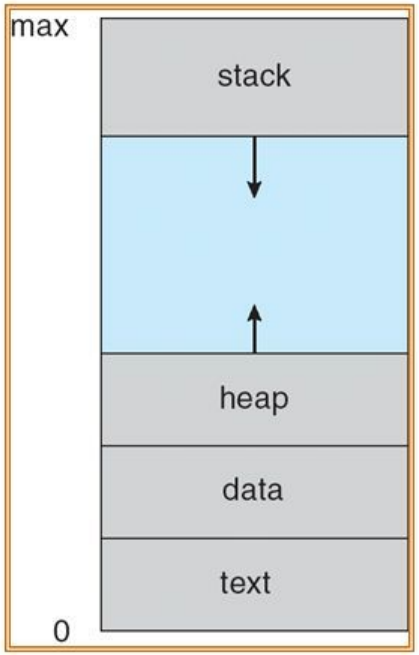
An operating system executes a variety of programs:

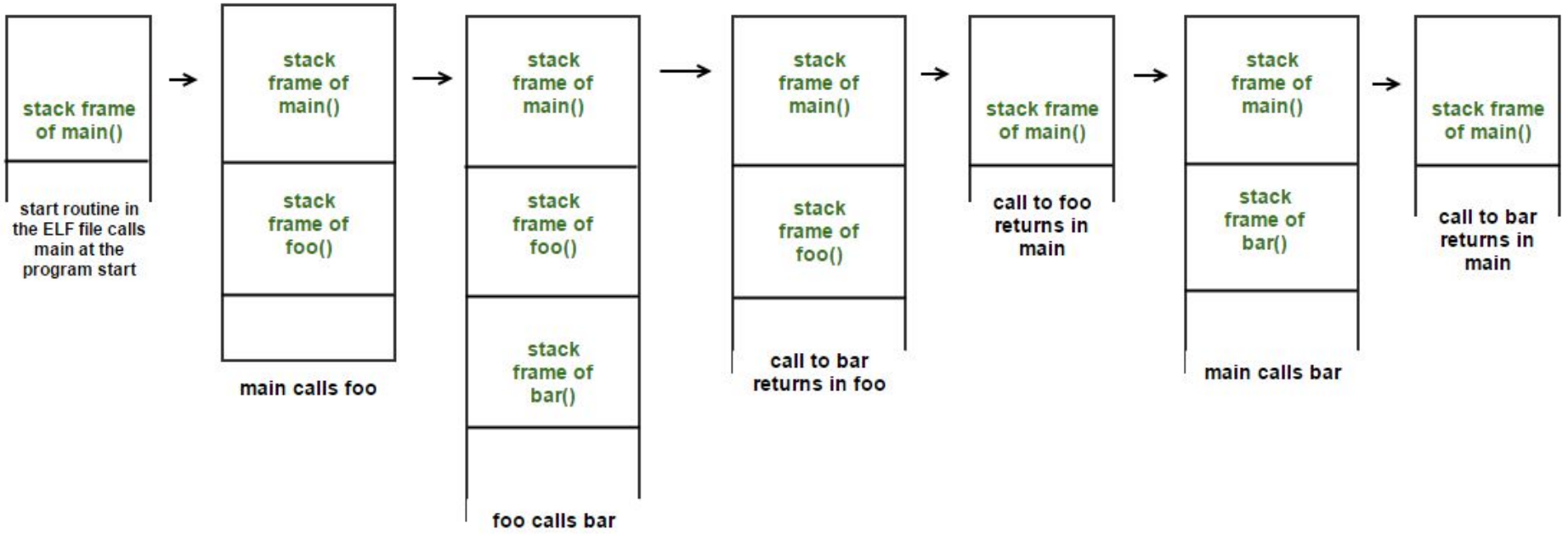
- Batch system – jobs
- Time-shared systems – user programs or tasks

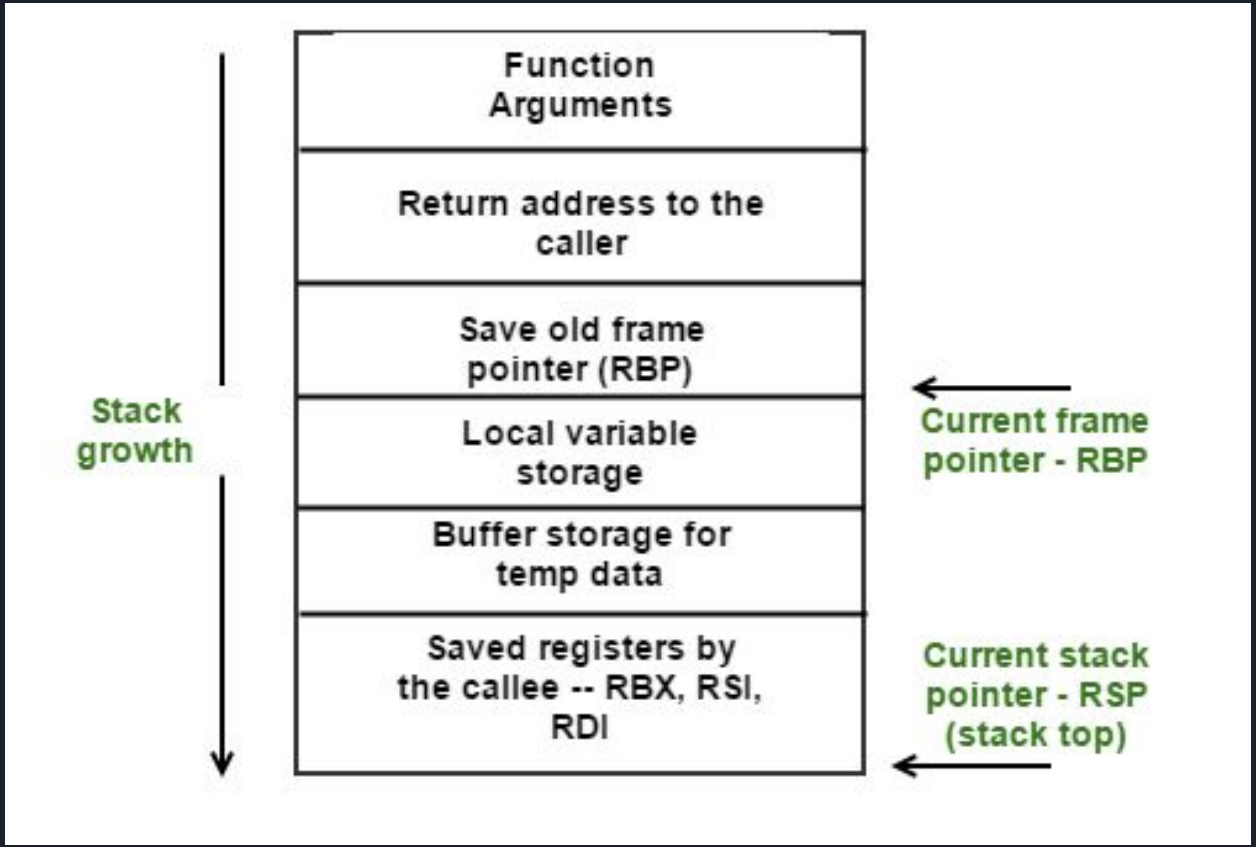
Process – a program in execution; process execution must progress in sequential fashion

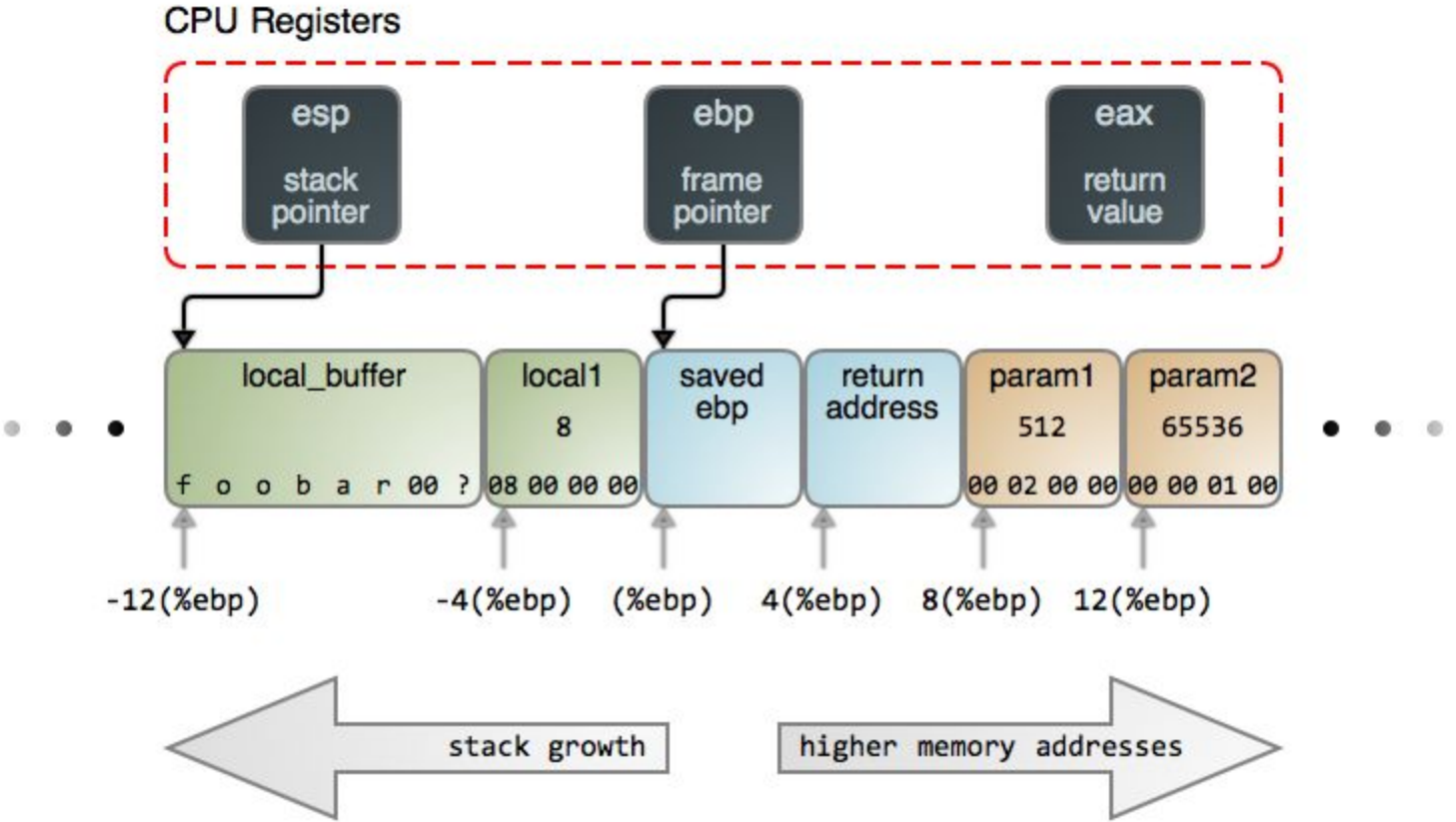
A process includes:

- program counter
- stack
- data section




stack frame of main()
start routine in the ELF file calls main at the program start
stack frame of main()
stack frame of foo()
main calls foo
stack frame of main()
stack frame of foo()
stack frame of bar()
foo calls bar
stack frame of main()
stack frame of foo()
call to bar returns in foo
stack frame of main()
call to foo returns in main
stack frame of main()
stack frame of bar()
main calls bar
stack frame of main()
call to bar returns in main


Function Arguments
Return address to the caller
Save old frame pointer (RBP)
Local variable storage
Buffer storage for temp data
Saved registers by the callee -- RBX, RSI, RDI
Stack growth
Current frame pointer - RBP
Current stack pointer - RSP (stack top)


CPU Registers
esp
stack pointer
ebp
frame pointer
eax
return value
local_buffer
f o o b a r 00 ?
local1
8
08 00 00 00
saved ebp
return address
param1
512
00 02 00 00
param2
65536
00 00 01 00
-12(%ebp)
-4(%ebp)
(%ebp)
4(%ebp)
8(%ebp)
12(%ebp)
stack growth
higher memory addresses

First C Program - Debugging Flags

https://www.youtube.com/watch?v=kSSxJTpoLGo

Reverse Engineering with radare2

Common Commands:

- aa
  - Analyze all
- aaa
  - Newer version. Does more analyzing!
- s
  - Seek to a position. Either offset in code or function name
  - s sym.main
- p
  - Show disassembly at current position
- V
  - Go into visual mode. Several different modes
- iz
  - Shows all strings where you are

Ghidra - Demo

- Don’t install on host machine
- Fuck Radare2
- crackMe

https://challenges.re/

http://crackmes.one/ <- currently down

https://ropemporium.com/ <- Return-Oriented Programming

Ghidra - General Outline

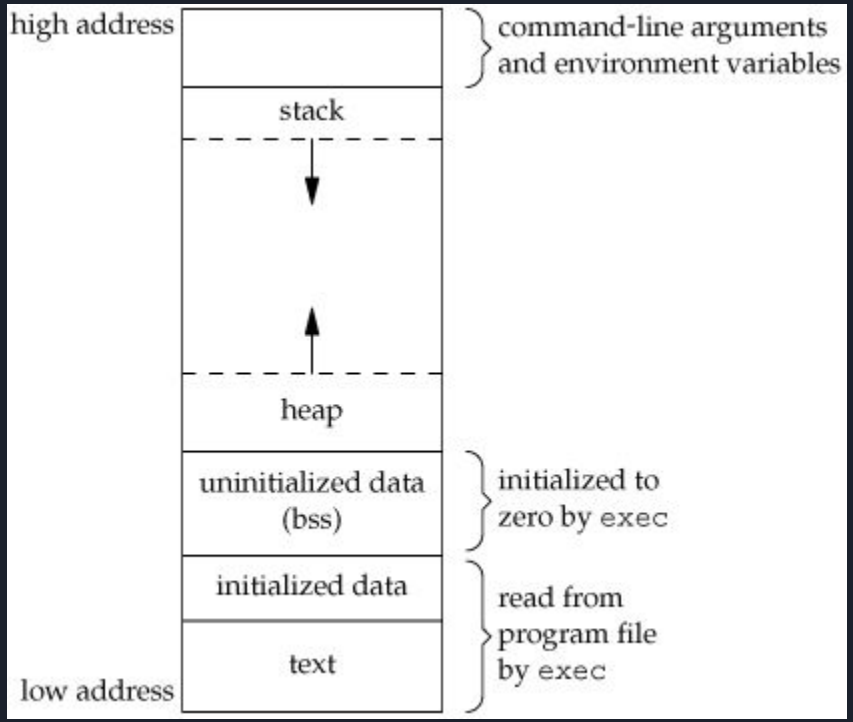
> load file into Ghidra (crackMe)
> analyze everything
>> tick the Decompiler option
> outline windows
> double-click main (or search)
> change to int main(int argc, char * argv[])
>> change to int main(int argc, char** argv)

Movfuscation

- Good luck!

https://github.com/xoreaxeaxeax/movfuscator

Buffer Overflows



Buffer Overflows - Program Memory

1. **Command line arguments and environment variables**: The arguments passed to a program before running and the environment variables are stored in this section.
2. **Stack**: This is the place where all the function parameters, return addresses and the local variables of the function are stored. It's a LIFO structure. It grows downward in memory(from higher address space to lower address space) as new function calls are made.
3. **Heap**: All the dynamically allocated memory resides here. Whenever we use malloc to get memory dynamically, it is allocated from the heap. The heap grows upwards in memory(from lower to higher memory addresses) as more and more memory is required.
4. **Uninitialized data(Bss Segment)**: All the uninitialized data is stored here. This consists of all global and static variables which are not initialized by the programmer. The kernel initializes them to arithmetic 0 by default.
5. **Initialized data(Data Segment)**: All the initialized data is stored here. This consists of all global and static variables which are initialised by the programmer.
6. **Text**: This is the section where the executable code is stored. The loader loads instructions from here and executes them. It is often read only.

Buffer Overflows - Stack Pointers

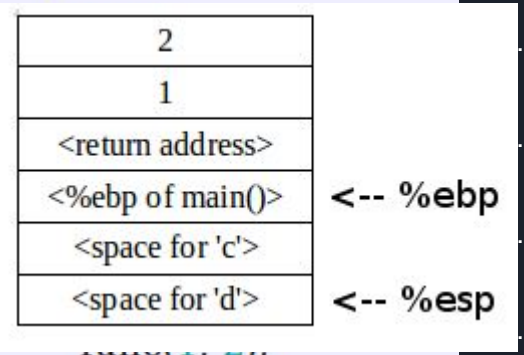
1. **%eip**: The **Instruction pointer register**. It stores the address of the next instruction to be executed. After every instruction execution it's value is incremented depending upon the size of an instruction.

2. **%esp**: The **Stack pointer register**. It stores the address of the top of the stack. This is the address of the last element on the stack. The stack grows downward in memory(from higher address values to lower address values). So the %esp points to the value in stack at the lowest memory address.

3. **%ebp**: The **Base pointer register**. The %ebp register usually set to %esp at the start of the function. This is done to keep tab of function parameters and local variables. Local variables are accessed by subtracting offsets from %ebp and function parameters are accessed by adding offsets to it as you shall see in the next section.

Buffer Overflows - Example Execution

```
void func(int a, int b)
{
```




```
    // next instruction
}
```

1. A function call is found, push parameters on the stack from right to left(in reverse order). So 2 will be pushed first and then 1.
2. We need to know where to return after func is completed, so push the address of the next instruction on the stack.
3. Find the address of func and set %eip to that value. The control has been transferred to func().
4. As we are in a new function we need to update %ebp. Before updating we save it on the stack so that we can return later back to main. So %ebp is pushed on the stack.
5. Set %ebp to be equal to %esp. %ebp now points to current stack pointer.
6. Push local variables onto the stack/reserver space for them on stack. %esp will be changed in this step.
7. After func gets over we need to reset the previous stack frame. So set %esp back to %ebp. Then pop the earlier %ebp from stack, store it back in %ebp. So the base pointer register points back to where it pointed in main.
8. Pop the return address from stack and set %eip to it. The control flow comes back to main, just after the func function call.

Buffer Overflows - Demo

```c
#include <stdio.h>

void petersSuperSecretFunction(){
   printf("Congratulations Shinji!\n");
   printf("You have entered my secret function!\n");
}

void echo(){
   char buffer[20];
   printf("Smash my stack by entering some text:\n");
   scanf("%s", buffer);
   printf("You entered: %s\n", buffer);
}

int main(){
   echo();
   return 0;
}
```

Buffer Overflows - Example Execution

gcc bof.c -o bof -fno-stack-protector -m32 -no-pie
-> disable PIE, disable stack protection, compile in 32bit mode
setarch `uname -m` -R /bin/bash
-> disable ASLR in shell
objdump -M intel -D bof | grep -A60 petersSuperSecretFunction
python -c 'print "a"*32 + "\xa6\x84\x04\x08"'

Buffer Overflows - Example Execution

```
>> get memory address -> 0000059d
>  5eb:   83 ec 0c                 sub      esp,0xc
> 616:   8d 45 e4                 lea      eax,[ebp-0x1c]
>> loading data into buffer -> 28 bytes
```

Now we know that 28 bytes are reserved for buffer, it is right next to %ebp(the Base pointer of the main function). Hence the next 4 bytes will store that %ebp and the next 4 bytes will store the return address(the address that %eip is going to jump to after it completes the function). Now it is pretty obvious how our payload would look like. The first 28+4=32 bytes would be any random characters and the next 4 bytes will be the address of the secret function.

```
> little-endian
> 00 00 05 9d
>> 9d 05 00 00
> python -c 'print "a"*32 + "\x9d\x05\x00\x00"'
```