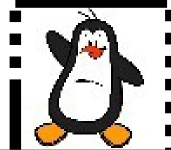


Lecture # 8.1
Exploiting Buffer Overflow Vulnerability Part-I

Course: Advance Operating System

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Source Code files available at: `https://bitbucket.org/arifpucit/spvl-repo/src`
Lecture Slides available at: `http://arifbutt.me`

Today's Agenda

- What is Buffer Overflow?
- A simple stack buffer overflow example
- How a stack based buffer overflow is exploited?
- Exploit mitigation techniques
- Recap: x86-64 Architecture
- Recap: x86-64 Assembly
- Function Calling Convention x86-64
- Proof of Concepts
- Installing and using PEDA (a gdb plugin)
- Changing control of flow of execution of a program

Cyber-Security and Vulnerabilities

- **Cyber-security** encompasses all the techniques for protecting computers, networks, programs, and data from unauthorized access or attacks that are aimed for exploitation
- A **vulnerability** is a flaw/weakness in a system design, implementation or security procedure that could be exploited resulting in notable damage. Example is a house with a weak lock on the main door. A *zero-day vulnerability* is a vulnerability that has been disclosed but is not yet patched. An exploit that attacks a zero-day vulnerability is called a *zero-day exploit*
- An **exploit** is a software that take advantage of a vulnerability leading to privilege escalation on the target. Example of an exploit is the duplicate key with the robber using which he/she can enter the house
- A **payload** is actual code which runs on the compromised system after exploitation. Example is the task that the robber will perform inside the house, i.e., stealing jewelry and cash

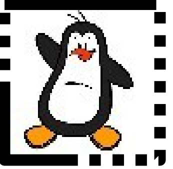
List of Common S/W Security Vulnerabilities

This is a brief list of types of vulnerabilities that compromise integrity, availability and confidentiality

- Buffer overflow
- Missing data encryption
- OS command injection
- SQL injection
- Missing authentication for critical function
- Missing authorization
- Unrestricted upload of dangerous file types
- Reliance on untrusted inputs in a security decision
- Cross-site scripting and forgery
- Download of codes without integrity checks
- Use of broken algorithms
- URL redirection to untrusted sites
- Path traversal
- Weak passwords
- Software that is already infected with virus

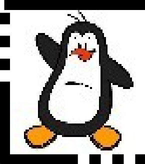
The list grows larger every year as new ways to steal and corrupt data are discovered

The vulnerability we are going to talk about today is **Buffer Overflow**



Buffer Overflow Example

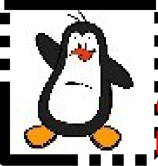
`stackoverflow.c`



Introduction to Buffer Overflow

- A buffer overflow is a bug in a program, which occurs when more data is written to a block of memory than it can handle. This can be stack based, heap based, integer overflow, off-by-one, and a format string
- The first published paper on this vulnerability was published in 1996 by Aleph One with the title of "**Smashing The Stack For Fun And Profit**", and later revived by Avicoder in 2017. (Both are a must read)
- Buffer overflow exploit was first used by **Morris Worm** (sendmail) in 1988, followed by **Code Red Worm** (IIS web server) in 2001 and **Slammer worm** (dos attk) in 2003. It is still one of the top vulnerability which cover a wide range of computer applications, libraries, operating systems and networking
- Hackers mostly use buffer overflows to corrupt the execution stack of a web app. By transferring fully crafted input to a web app, a hacker can make the web app to execute arbitrary code and probably taking over the server
- Although there are many h/w and s/w based techniques and tools that have been proposed and developed to detect and protect from buffer overflow vulnerability, but based on the trend it look likes this problem will continue to happen

https://www.eecs.umich.edu/courses/eecs588/static/stack_smashing.pdf
https://avicoder.me/papers/pdf/smashthestack.pdf



Stack Based Buffer Overflow

Hi address

Stack
...
Function Parameters / AAAAAAAA
Return Address RIP / AAAAAAAA
Caller's RBP / AAAAAAAA
Alignment Space / AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA

Low address

Stack grows towards smaller addresses

Return Address

Buffer ends here

Buffer grows towards higher addresses

Buffer starts here

```
int f1(){
   char buff[48];
   gets(buff);
   printf("%s\n", buff);
   return 0;
}
int main(){
   int rv = f1();
   exit(0);
}
```

Stack Based Buffer Overflow Exploit

Security Protection Mechanisms:

- NX bit

```
gcc -z execstack prog.c
```

- Stack Canary

```
gcc -fno-stack-protector prog.c
```

- ASLR

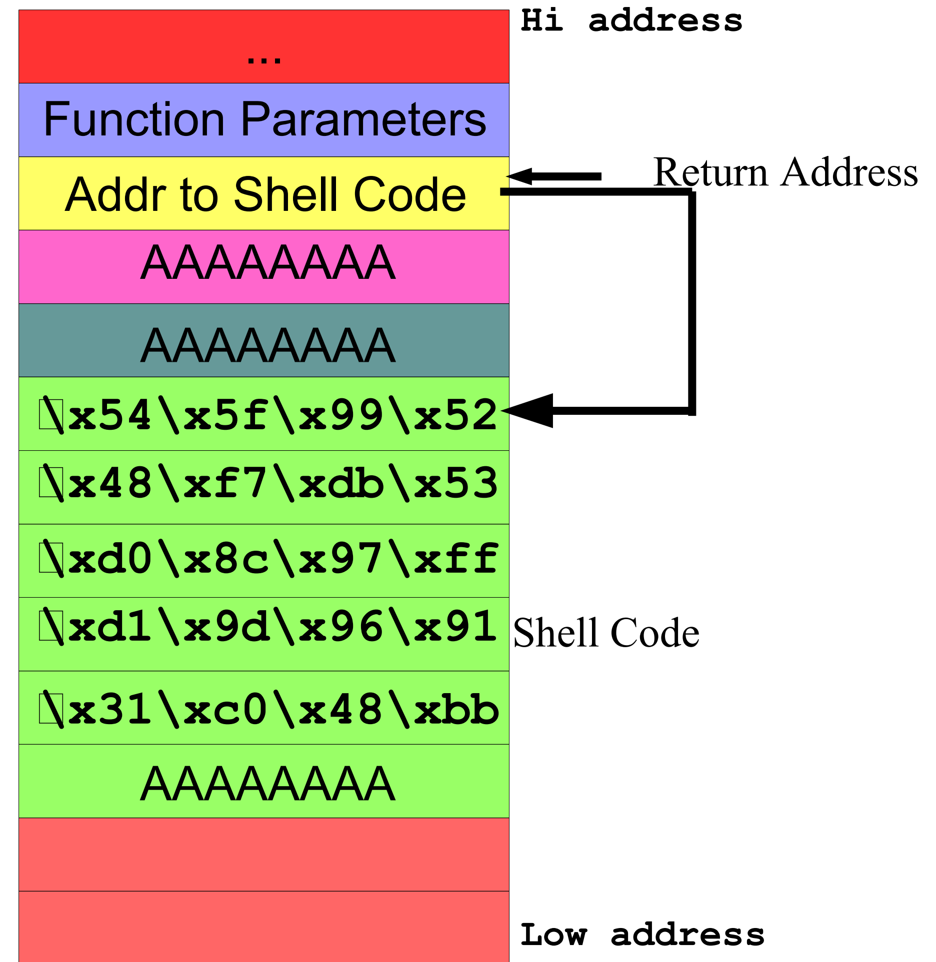
```
echo 0 | sudo tee /proc/sys/kernel/
                 randomize_va_space
```

- PIE

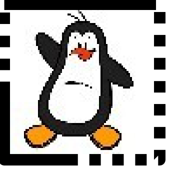
```
gcc -no-pie prog.c
```

- FORTIFY_SOURCE

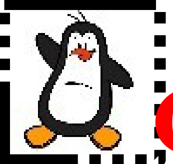
```
gcc -D_FORTIFY_SOURCE -O2 prog.c
```



However, none of these exploit mitigation techniques is completely foolproof, and can be bypassed by using a bit of intelligence



RECAP
Architecture and Assembly of x86-64

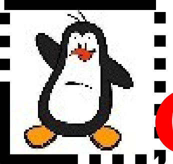


General Purpose Registers

There are sixteen 64 bit general purpose registers (GPRs) and we can access their lower 32, 16 as well as 8 bits. In assembly programs we use these registers as variables

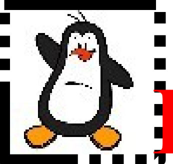
- **rsp:** is the stack pointer and is used to point to the current top of the stack. Whether it point to the last allocated address on the stack or to the next free address, is implementation dependent. It should not be used for data or other uses
- **rbp:** is the base/frame pointer and always points to a fixed location within a frame. All the local variables and parameters within a frame are referenced by giving their offset from rbp. It should not be used for data or other uses
- **rip:** is the instruction pointer that stores the address of the next instruction to be executed

64 bit	32 bit	16 bit	8 bit
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b



Code Segment Registers

- There are 6 x 16 bit segment registers (CS, DS, SS, ES, FS, GS), each holding a 16 bit segment selector. The four segment registers (CS, DS, SS, ES) are the same as found in Intel 8086, while FS and GS were introduced in IA-32
- Each of the segment registers is associated with one of the three types of storage: code, data or stack
- **CS** register contain the segment selector for the **code segment**, where the instructions being executed are stored. So the rip register contains the offset within the code segment of the next instruction to be executed
- **SS** register contain the segment selector for the **stack segment** containing the function stack frames or activation records
- The **DS**, **ES**, **FS**, and **GS** registers points to the **four data segments**. The availability of four data segments permits efficient and secure access to different types of data structures. For example, four separate data segments might be created: one for the data structures of the current module, another for the data exported from a higher-level module, a third for a dynamically created data structure, and a fourth for data shared with another program

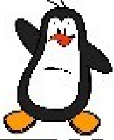


Flags Register

rflags	1	1	1	0	1	0	1	1	0	0	0	1	1	0	0	1	1	0	0	1	0	0	0	1	1	0	0	0	1	0	1	0

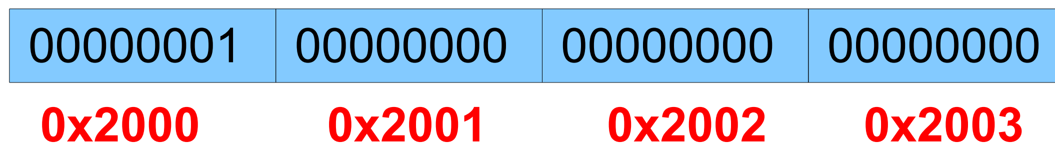
The `rflags` register is used for status and CPU control information. It is updated by the CPU after each instruction is executed. Out of the 64 bits mostly are unused and reserved for future use. Some important flags are mentioned below:

- Status Flags
  - Carry flag (CF). Bit 0 is set if the previous operation resulted in a carry or a borrow out of the msb of result
  - Parity flag (PF). Bit 2 is set if the LSB of the result contains an even number of 1s
  - Auxiliary flag (AF). Bit 4 is set if an arithmetic op generates a carry or a borrow out of bit 3 of result (BCD)
  - Zero flag (ZF). Bit 6 is set if the previous operation resulted in a zero result
  - Sign flag (SF). Bit 7 is set equal to the msb of the result, which is the sign bit of a signed integer
  - Overflow flag (OF). Bit 11 is set if the previous operation resulted in an overflow
- Control Flags
  - Direction flag (DF). Bit 10 is used to specify direction (inc or dec) for string operations
- System Flags
  - Interrupt enable flag (IF). Bit 9 is set to respond to maskable interrupts
  - Trap flag (TF). Bit 8 is set to enable single step mode for debugging
  - Resume flag (RF). Bit 16 is used to control the cpu response to debug exceptions

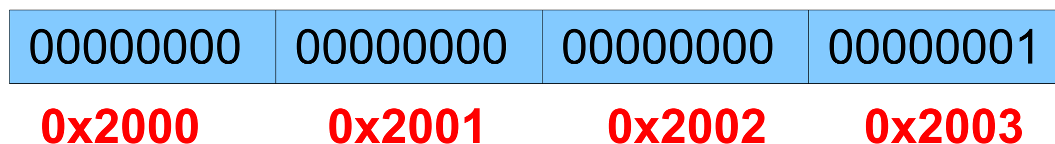


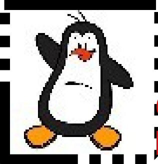
IA-64 is Little Endian

- Byte order is the attribute of a processor that indicates whether integers are represented from left to right or right to left in the memory
- In **Little Endian Byte Order**, the low-order byte of the number is stored in memory at the *lowest address* and the high-order byte of the same number is stored at the highest address. The figure shows how a decimal number one is store in 4 bytes in Little Endian scheme



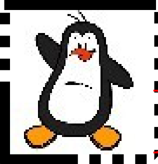
- In **Big Endian Byte Order**, the low-order byte of the number is stored in memory at the *highest address* and the high-order byte of the same number is stored at the lowest address. The figure shows how a decimal number one is store in 4 bytes in Big Endian scheme





Summary of Basic Assembly Instructions

Instr Category	Meaning	Example
Data transfer	Move from source to destination	mov, lea, lds, les, push, pop, pushf, popf
Arithmetic	Arithmetic on integers	add, addc, sub, subb, mul, imul, div, idiv, cmp, neg, inc, dec, xadd, cmpxchg
Floating point	Arithmetic on floating pt	fadd, fsub, fmul, fdiv
Shift, Rotate	Bit wise logic operations	and, or, xor, not, shl/sal, shr, sar, ror, rol, rcr, rcl
Control transfer	Conditional and unconditional jumps, and procedure calls	jz, jnz, jo, jno, jp, jnp... jmp call, ret
String	Move, compare, input and output	movs, lods, stos, scas, cmps, outs, rep, repz, repe, repnz, repne, ins
I/O	For input and output	in, out
Conversion	Assembly data type conversions	movzs, movsx, cbw, cwd, cwde, cdq, bswap, xlat
Miscellaneous	Manipulate individual flags	clc, stc, cmc, cld, std, cl, sti

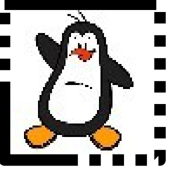


Machine, Assembly and Hi Level Languages

Language	Example

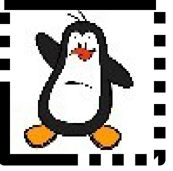
High Level Language	#include <stdio.h>
	#include <stdlib.h>
	int main(){
	   printf("Learning is fun with Arif\n");
	   exit(0);
	}

Assembly Language	0x0...468a <+0> : push rbp 0x0...468b <+1> : mov rbp, rsp 0x0...468e <+4> : lea rdi, [rip+0x9f] 0x0...4695 <+11>: call 0x555555554550 <puts@plt> 0x0...469a <+16>: mov edi, 0x0 0x0...469f <+21>: call 0x555555554560 <exit@plt>
Machine Language	0: 55 1: 48 89 e5 4: 48 8d 3d 00 00 00 00 b: e8 00 00 00 00 10: bf 00 00 00 00 15: e8 00 00 00 00

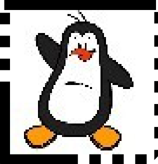


C vs Assembly vs Machine Language
Proof of Concept

prog1.c - prog3.c
prog4.nasm, prog5.nasm

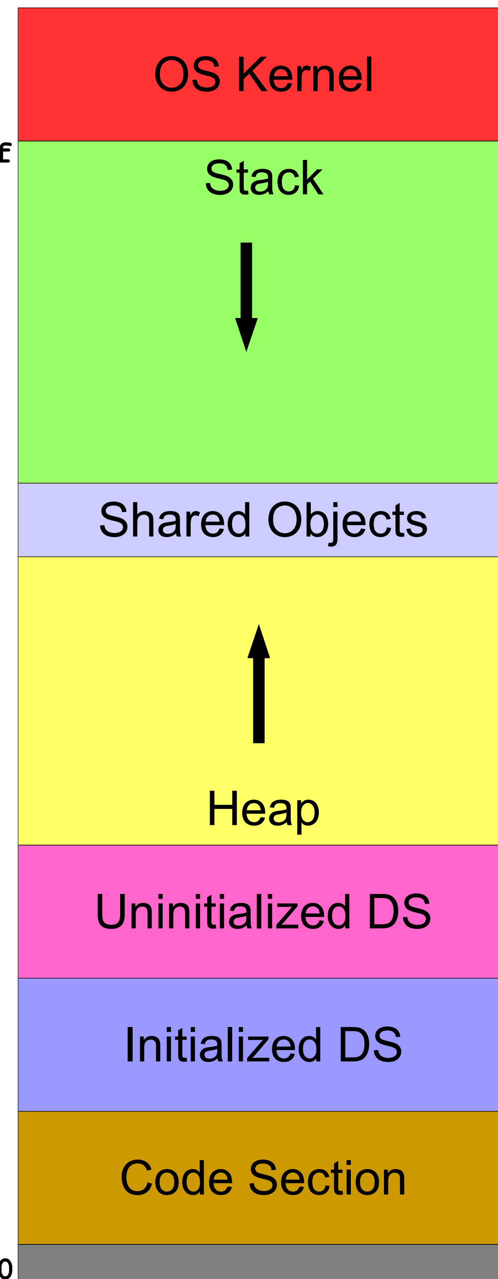


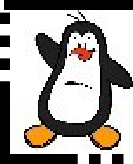
x86-64
Function Calling Convention



Logical Layout of a Process Address Space

- The diagram shows the logical process address space of a process
- Code section contains machine code instructions of your program
- Above code section we have initialized and uninitialized data sections for global variables
- Heap is used for dynamic memory allocation and it grows towards higher addresses
- The stack is at the top of memory below the kernel code and grows from higher memory addresses to lower memory addresses in architectures like Intel, MIPS, Motorola, SPARC
- All the push/pop on the stack are 8 Bytes wide on x86_64
- Whenever a function is called, a new function stack frame or activation record is created





Function Calling Convention

- A function stack frame or activation record is pushed/created on the stack when a function is called and popped/removed from the stack when a function returns. Function calling convention means:
  - How the function arguments are passed?
  - In which order the function arguments are passed?
  - Who is responsible for creating the FSF?
  - Who is responsible for unwinding the FSF?
- On x86_64, the first six integer or pointer arguments are passed via registers (`rdi, rsi, rdx, rcx, r8, r9`). Any floating point values are passed in `xmm0, xmm1, xmm2,` …. The seventh argument onwards are passed via stack in reverse order. The function return value is placed in register `rax`
- On x86_64, the FSF is created by callee using **procedure prolog**
- On x86_64, the FSF is unwinded by the callee using **procedure epilog**

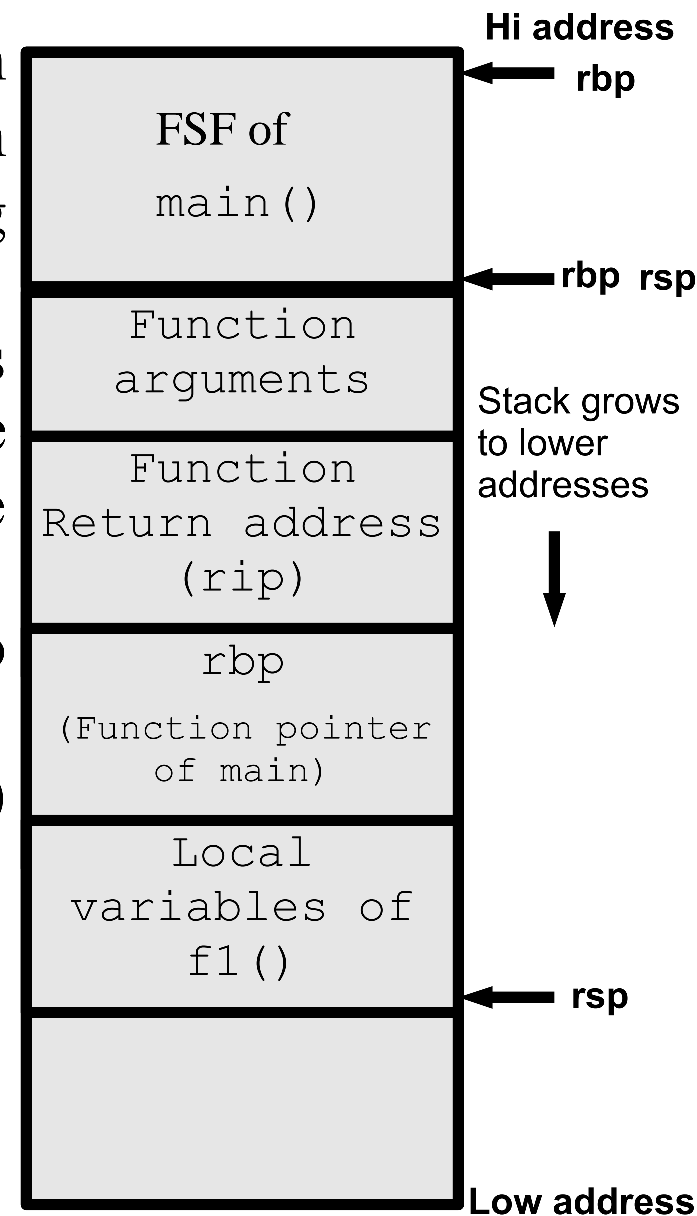
Creation of FSF on Stack

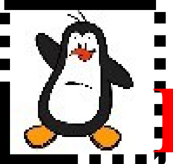
Suppose the `main()` (caller) calls another function `f1()`(callee). Before the control goes to function `f1()`, the code inside `main()` performs following two tasks:

- Places the function arguments in six registers (`rdi, rsi, rdx, rcx, r8, r9`) and the rest (if any) are pushed on the stack in reverse order
- The contents of `rip` (return address) is also pushed on the stack

Then the control shifts to the called function (callee) `f1()`, which perform a **procedure prolog**:

```
PUSH rbp
MOV rbp, rsp
SUB rsp, 0X20
```

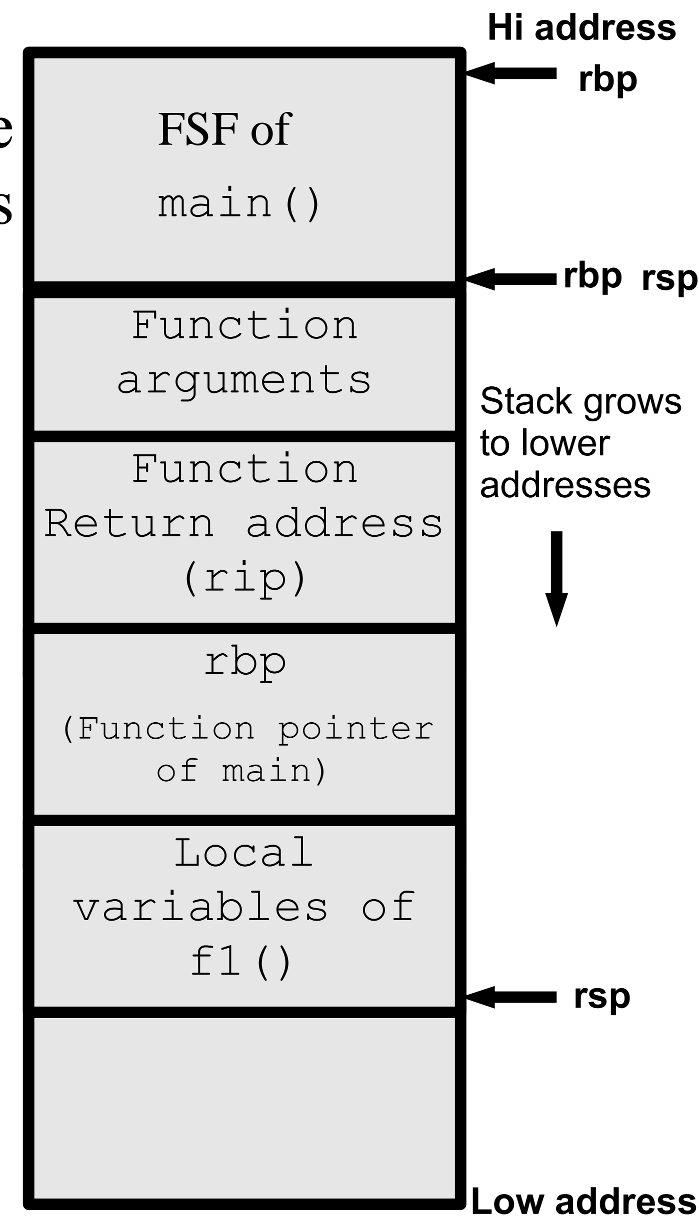




Removal of FSF from Stack

After the function `f1()` is done with its execution, it finally calls the `return` statement to return the control to the `main()` function, and clean up its function stack. We say it performs **procedure epilog**:

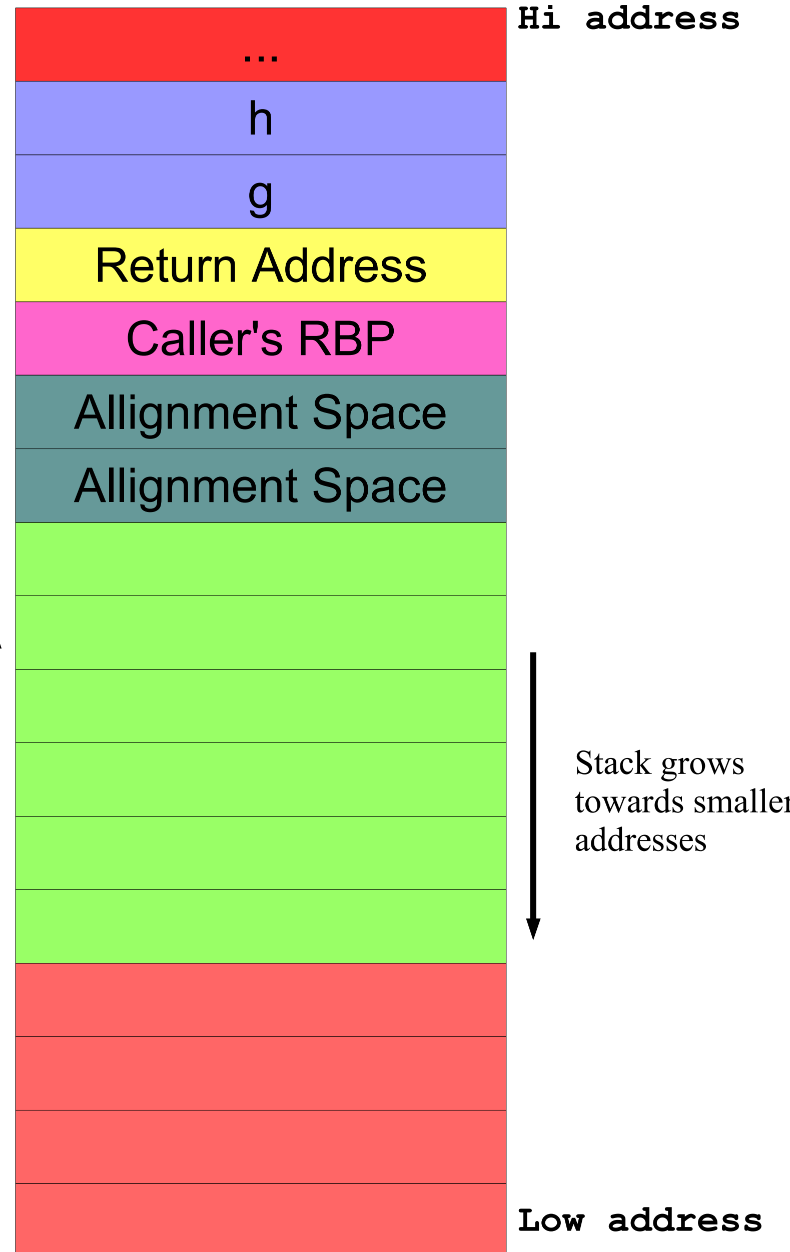
```
LEAVE  ->  MOV rsp, rbp
           POP rbp
RET    ->  POP rip
```

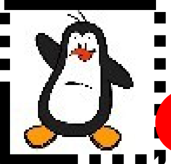


Function Stack Frame on x86-64

```
int f1(long a, long b, long c, long d,
       long e, long f, long g, long h)
{
    char buff[48];
    gets(buff);
    return 1;
}
```

RDI:	a
RSI:	b
RDX:	c
RCX:	d
R8:	e
R9:	f

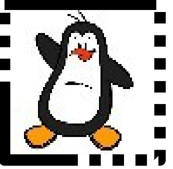




Common Debuggers / Code Analyzers

There is a long list of debugging and reverse engineering tools that can be used to debug, decompile, disassemble and analyzing binaries like hopper, edb, IDA Pro, Radare2

- **GNU Debugger (GDB)** is a portable debugger that runs on many UNIX-like systems and works for many programming languages, including Ada, C, C++, Objective-C, Java, Fortran, Pascal, Go, and many others. The target processors include IA-32, x86-64, alpha, arm, mips, powerpc, sparc and many others. It offers a command line interface, but several front ends have been built for it like Data Display Debugger (DDD), Xcode debugger. IDEs like Visual Studio, NetBeans, Eclipse, Code::Blocks, Dev-C++, and Geany can interface with gdb. Too good a debugger, however, it lacks intuitive interface, do not have a smart context display, do not have commands for exploit development, and has weak scripting support
- **PEDA:** Python Exploit Development Assistance is a plugin for GDB used extensively in exploit development. It is supported by gdb 7.x and Python2.6+



Function Calling Convention Proof of Concept

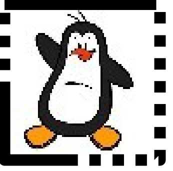
`func_calling.c`

Link for gdb:

`https://www.youtube.com/watch?v=7D3R65Vm3B8&t=0s&list=PL7B2bn3G_wfD8xy4lUaoItwwJ3zKlpuUe&index=3`

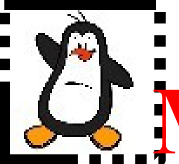
Download and install PEDA:

```
$ mkdir   ~/peda
$ git clone   https://bitbucket.org/arifpucit/peda.git ~/peda/
$ echo    "source ~/peda/peda.py"   >>   ~/.gdbinit
```



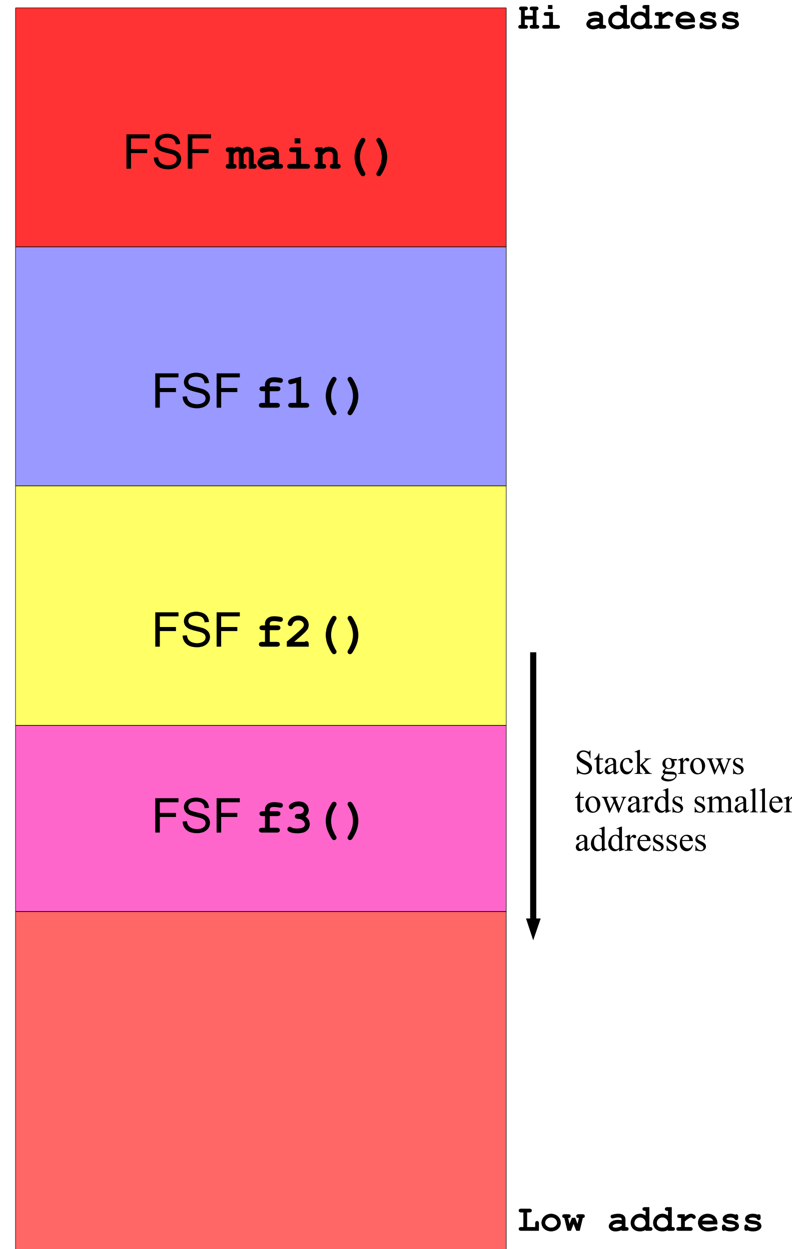
Modify the Return Address

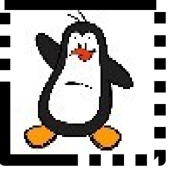
`stackoverflow.c`



Modifying the Function Return Address

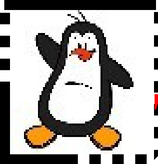
```
void f3(){ return;}
void f2(){ f3();  }
void f1(){ f2();  }
int main(){
   f1();
   return 0;
}
int virus(){
   printf("Let us Hack planet earth with
          Arif Butt.\n");
   exit(0);
}
```





Changing Control of Flow
Proof of Concept

`virus.c`



Things To Do

O.k., and now you'll do
exactly what I'm telling you !

Access
denied

If you have problems visit me in counseling hours. . . .