OS Security: Stack Overflow

OS Exploits

- OS is a software and has bugs that can be exploited for attacking its internal data structures for privilege escalation
- Exploits could result from
  - Stack overflow or buffer overflow
  - Heap overflow
  - Integer overflow
  - String format attacks
  - Return to libc
  - Gadgets or Return oriented programming

Buffer Overflow: Basics

- Application reserves adjacent memory locations (buffer) to store function parameters or variable(s).
  - buffer can be on stack, heap, global data
- Attacker gives a value too long to fit in the buffer
- Application copies value (without validating size), overflowing buffer (and memory) leading to
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker with the same privileges of the original application

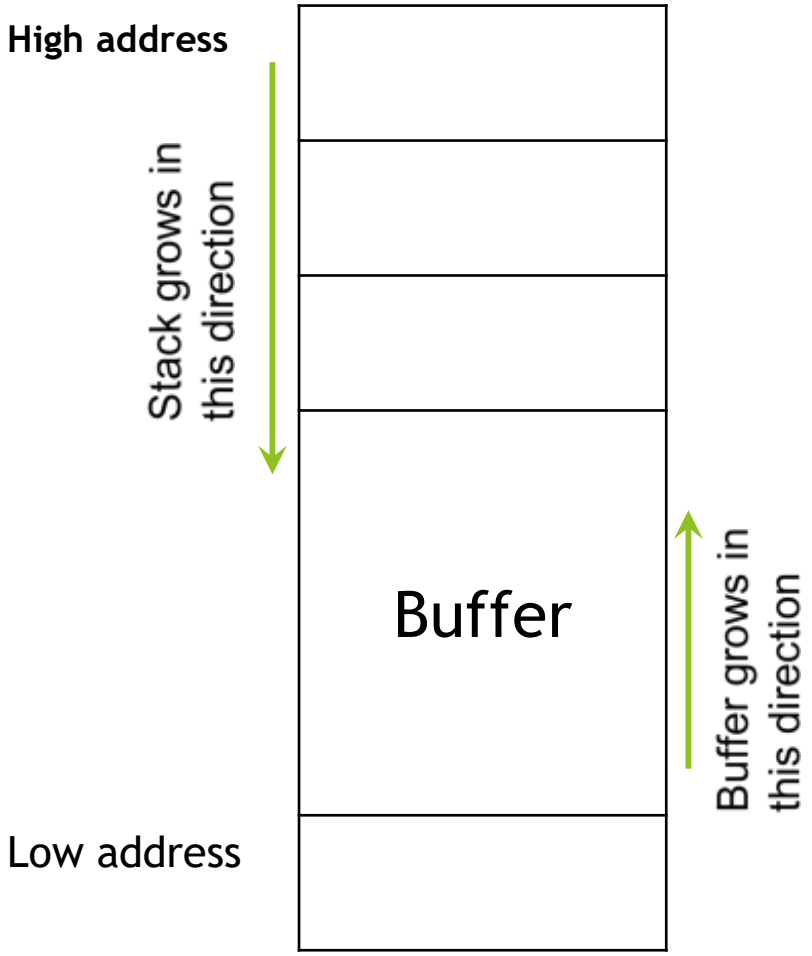
Stack Overflow

- Most common vulnerability
- Buffer on stack used for overflow
- Stack used for implementation of functions
- F1() calls F2() and F2() calls F3(); when F3() returns control should go back to F2() and on return from F2(), it should go back to F1()
- Functions calls are executed in LIFO so stack is suited for implementation

Activation Frame

- On each function call, an activation record or activation record is pushed onto stack
- Activation stack/frame consists of
  - parameters to be passed to the called function,
  - return address in callee function (instruction address where control shall return to after the called function has been executed),
  - old stack pointer value local variables of the called function,

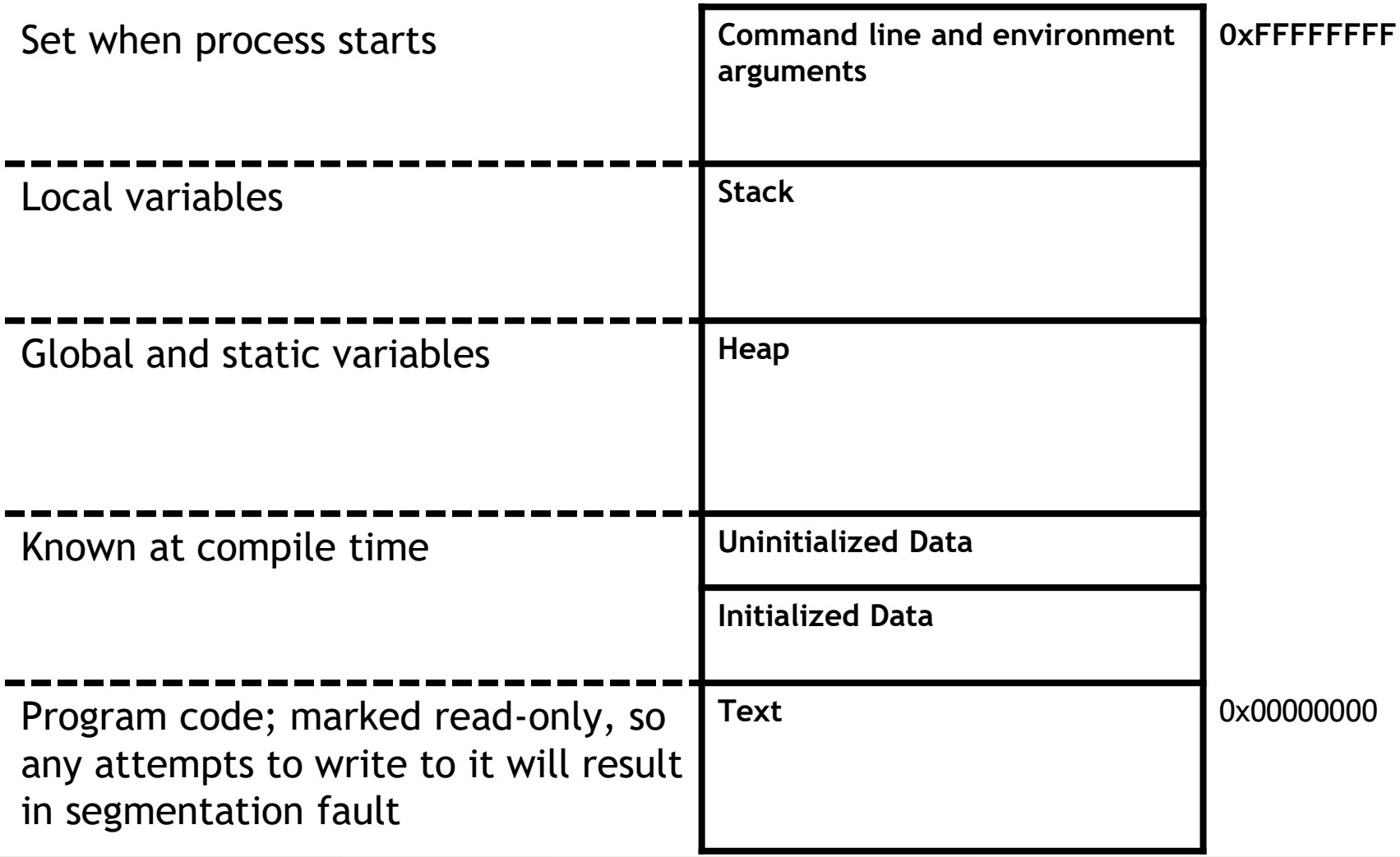
Activation Frame



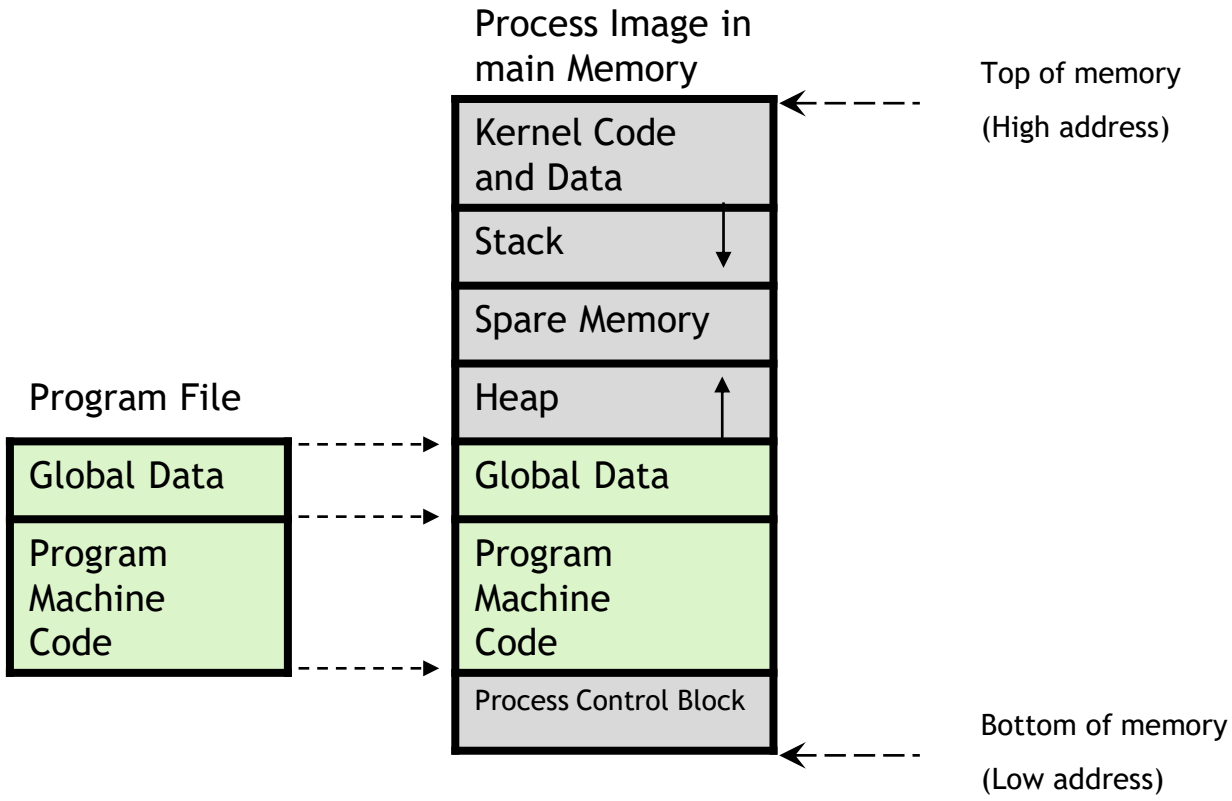
- Stack grows from higher to lower memory address
- Conventionally, top maps to the higher memory and bottom maps to the lower
- A buffer on stack grows in opposite direction i.e. lower to higher memory address.
- Address(*buffer*[*k*+*1*]) > Address(*buffer*[*k*])

Process Virtual Address Space

Stack and heap grow in opposite directions

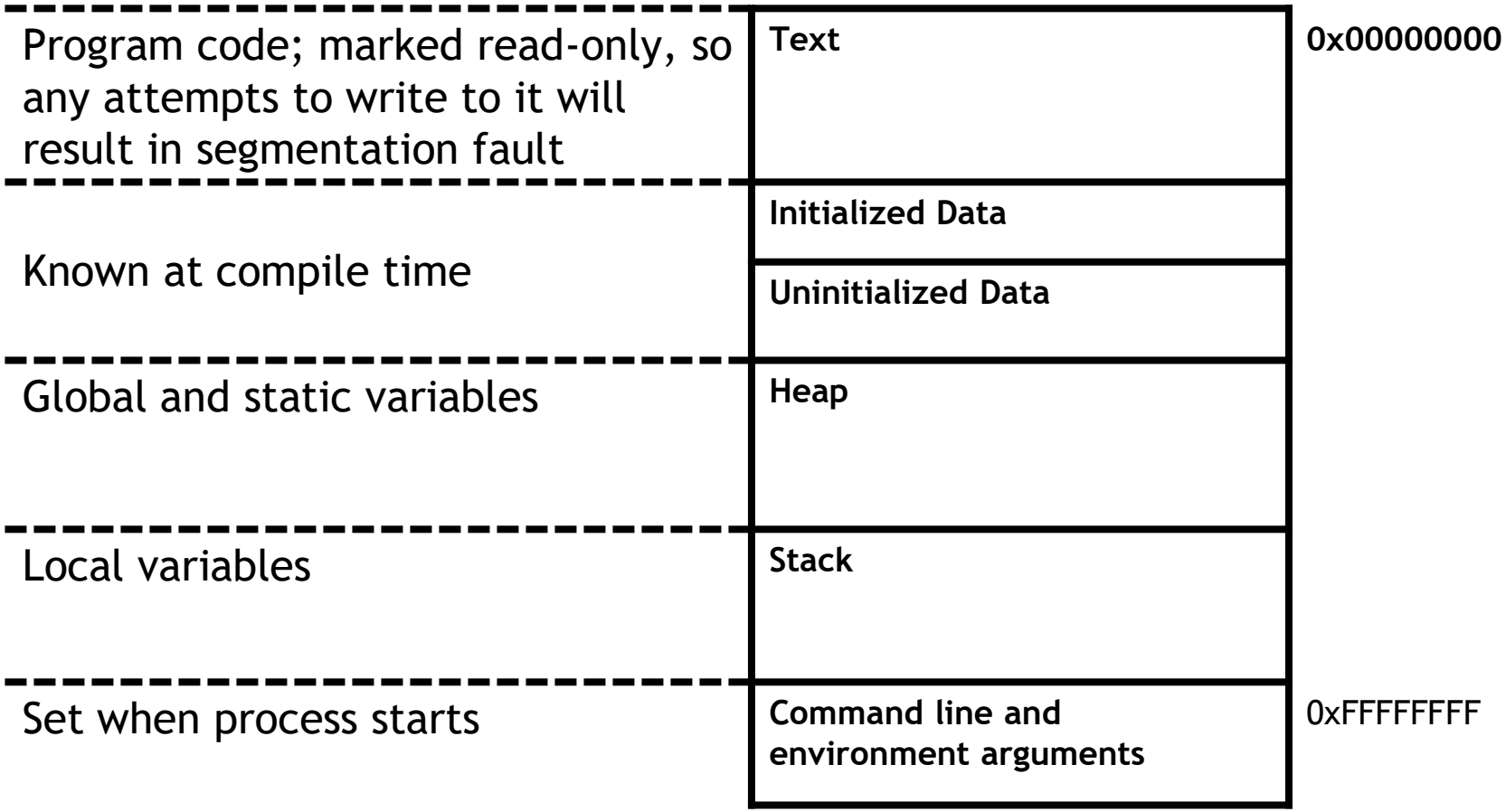


Programs and Processes



Process Virtual Address Space

Stack and heap grow in opposite directions



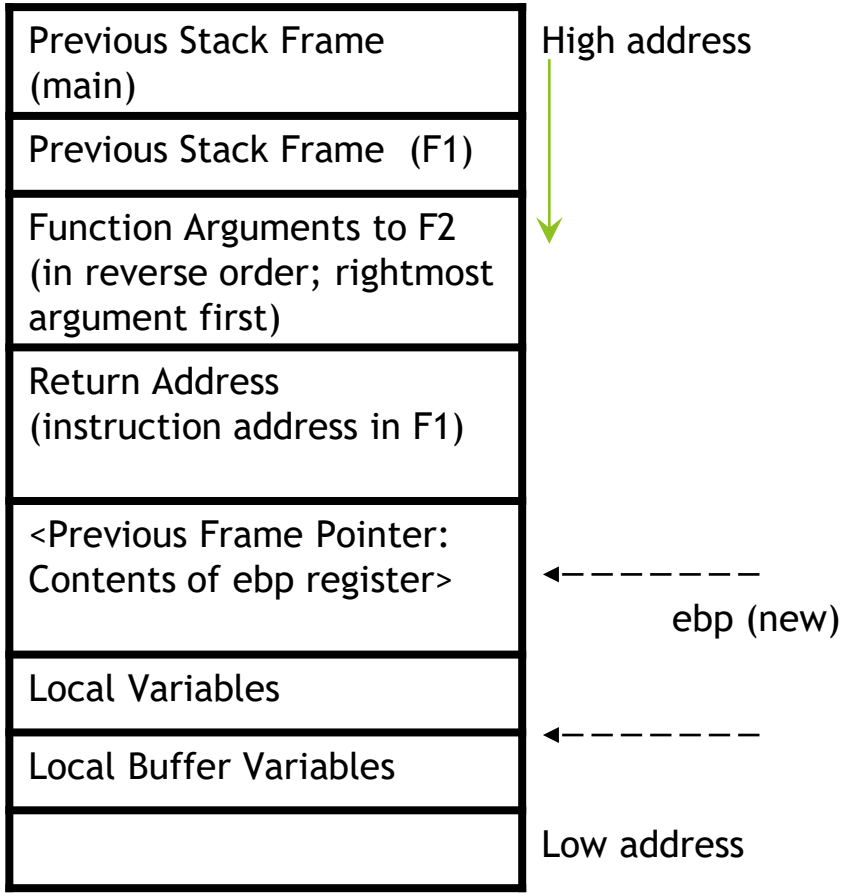
Stack Layout: Activation Frame

- What do we do when we call a function?
  - What data need to be stored?
  - Where do they go?
- How do we return from a function?
  - What data need to be restored?
  - Where do they come from?

Creating New Activation Frame

- Calling function:
  - Push arguments onto the stack (in reverse order)
  - Push the address of the instruction to run after control returns to you
  - Jump to the function
- Called function:
  - Push the old frame pointer onto the stack (%ebp)
  - Set current frame pointer (%ebp) to where the end of the stack is right now (%esp)
  - Push local variables onto the stack

Activation Frame



- Arguments in reverse order
- Return address
- Store old frame pointer (%ebp) – so that context can be returned to callee function after this call
- Set current frame pointer (%ebp) to where the end of the stack is right now (%esp)
- Push local variables onto the stack

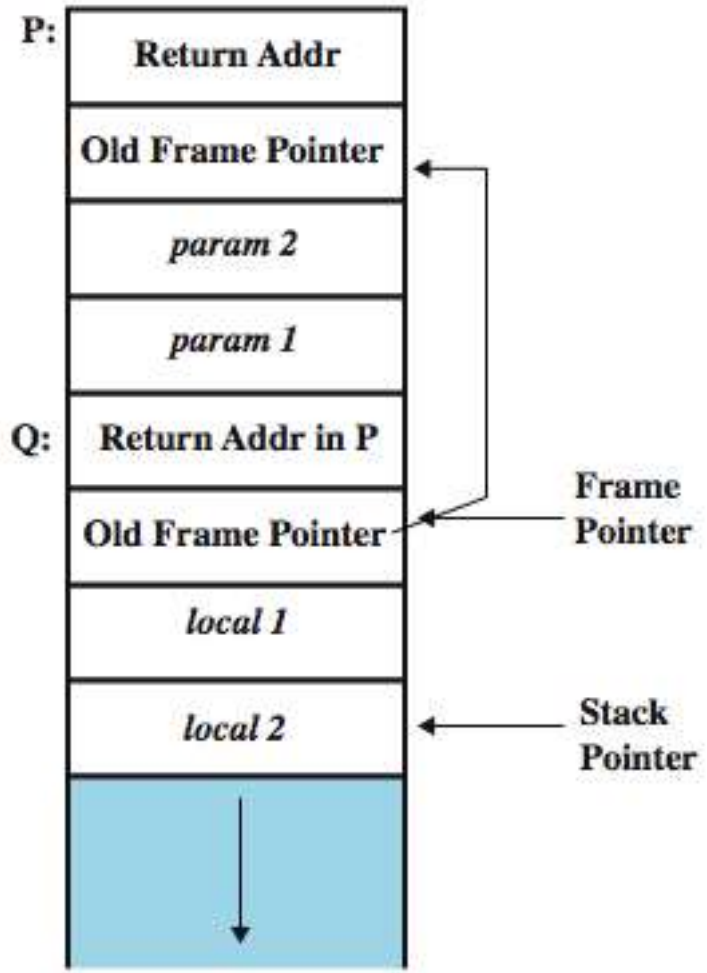
Removal of Activation Frame

- Called function (to return):
  - Deallocate local variables: %esp = %ebp
  - Restore base pointer: pop %ebp
  - Jump back to where they wanted us to: %eip = (%esp)
- Calling function (on return):
  - Remove arguments from stack

Activation Frames (contd).

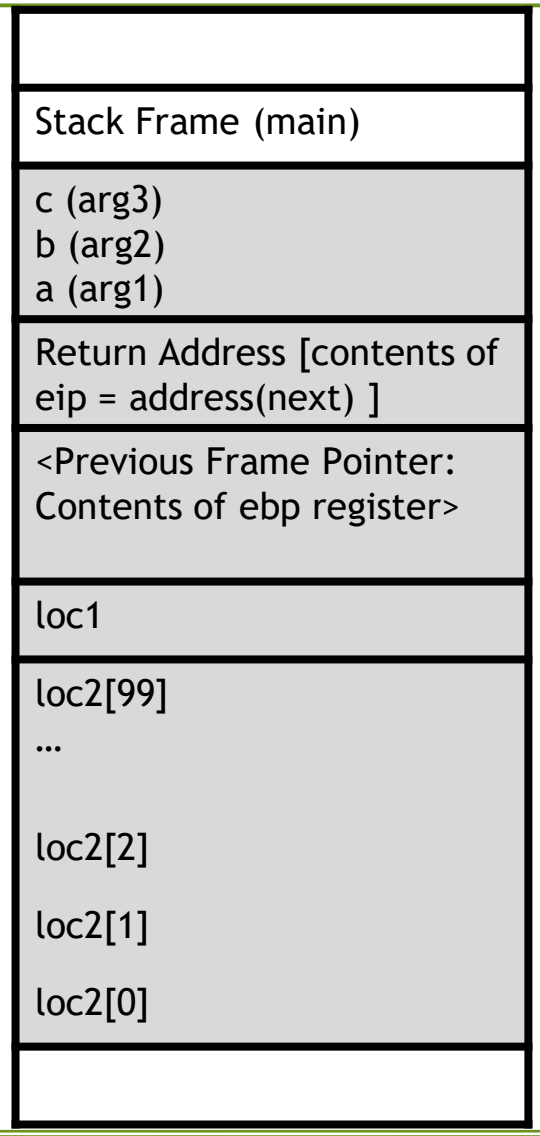
Calling function: needs a data structure to store the “return” address and parameters to be passed

Called function: needs a place to store its local variables (different for every call)



Buffer Overflow: An example

```
void func(char *arg1, int arg2, int
arg3)
{
    int loc1, loc2[100];
        ...
    loc2++;
    ...
}
void main(int argc, char **argv) {
    …
    func(a,b,c);
    next
    …
}
```



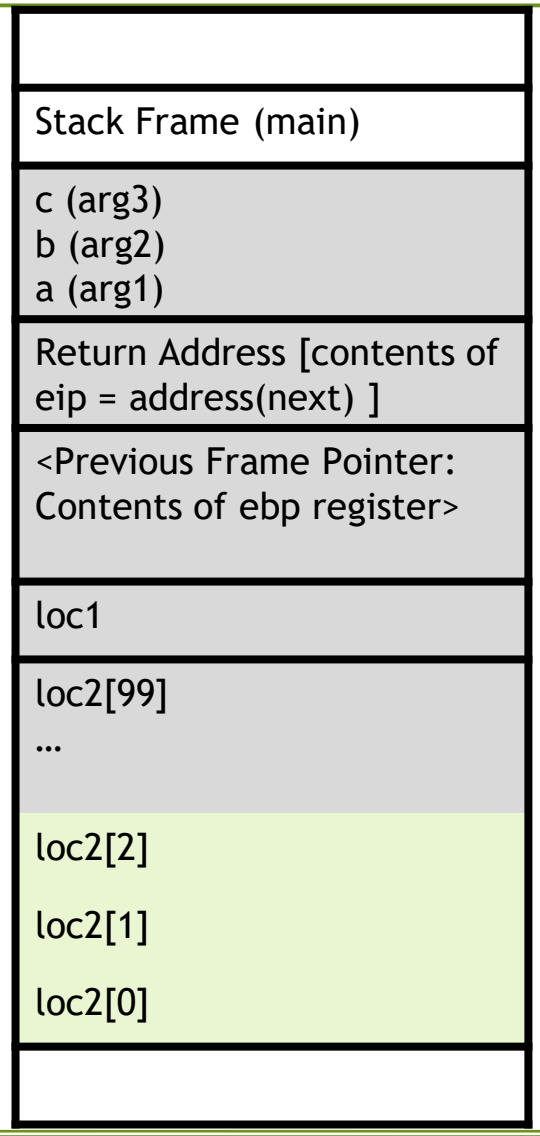
Every variable address is relative to stack pointer

loc2 is buffer here

ebp (new)

Buffer Overflow: An example

```
void func(char *arg1, int arg2, int
arg3)
{
    int loc1, loc2[100];
        ...
    loc2++;
    ...
}
void main(int argc, char **argv) {
    …
    func(a,b,c);
    next
    …
}
```



High address

Low address

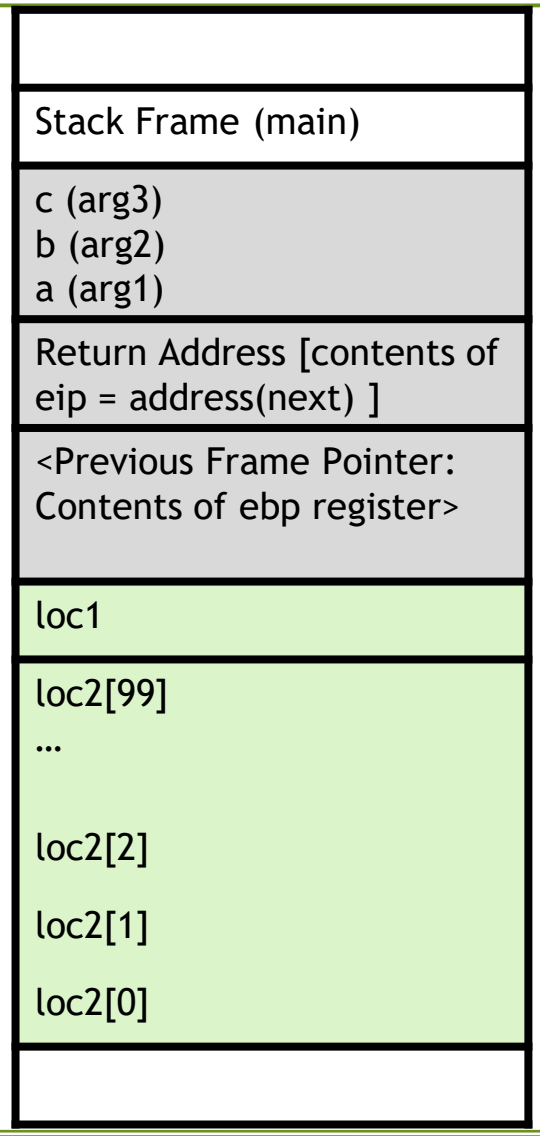
User input (less than 100) can be fit in buffer

No Overflow

ebp (new)

Buffer Overflow: An example

```
void func(char *arg1, int arg2, int
arg3)
{
    int loc1, loc2[100];
        ...
    loc2++;
    ...
}
void main(int argc, char **argv) {
    …
    func(a,b,c);
    next
    …
}
```

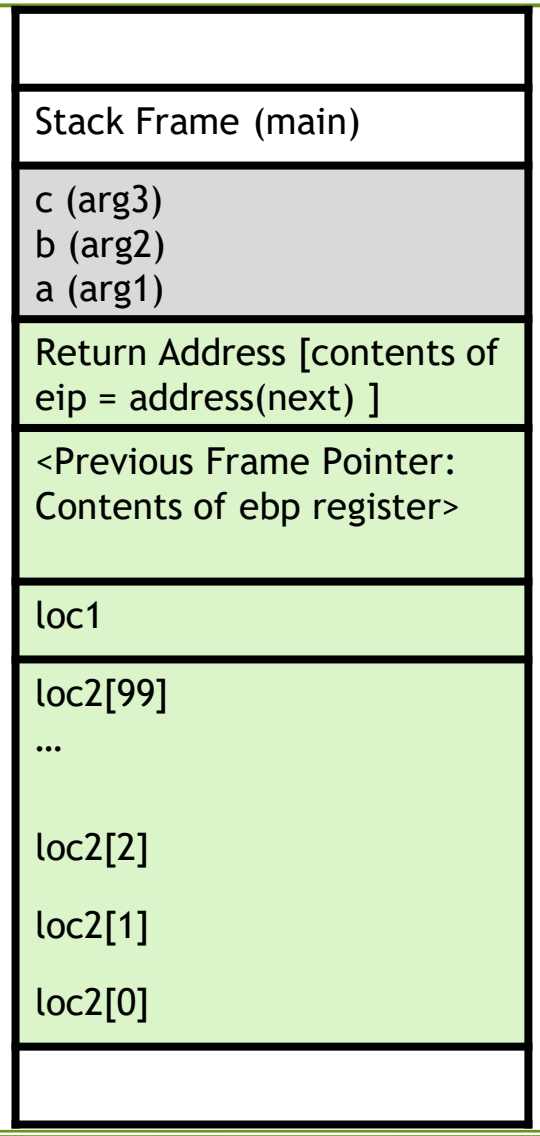


What if user input exceeds buffer capacity?
Overflow: other data on stack overwritten

ebp (new)

Buffer Overflow: An example

```
void func(char *arg1, int arg2, int
arg3)
{
    int loc1, loc2[100];
        ...
    loc2++;
    ...
}
void main(int argc, char **argv) {
    …
    func(a,b,c);
    next
    …
}
```

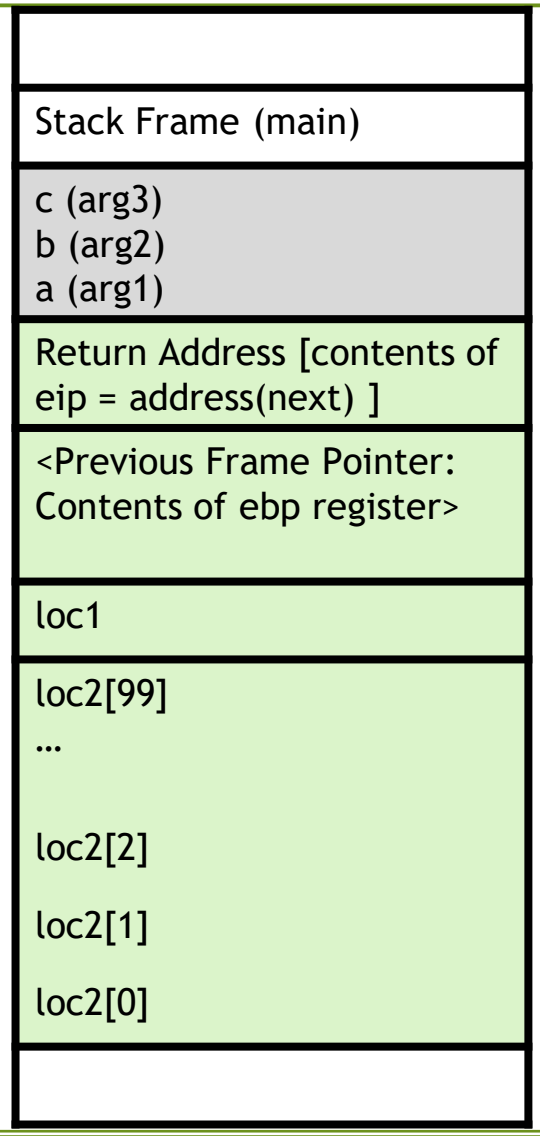


What if user input exceeds buffer capacity? Overflow: return address corrupted

ebp (new)

Buffer Overflow: An example

```
void func(char *arg1, int arg2, int
arg3)
{
    int loc1, loc2[100];
        ...
    loc2++;
    ...
}
void main(int argc, char **argv) {
    …
    func(a,b,c);
    next
    …
}
```



Overflow: return address corrupted
Control flow affected
How can malicious user exploit this?

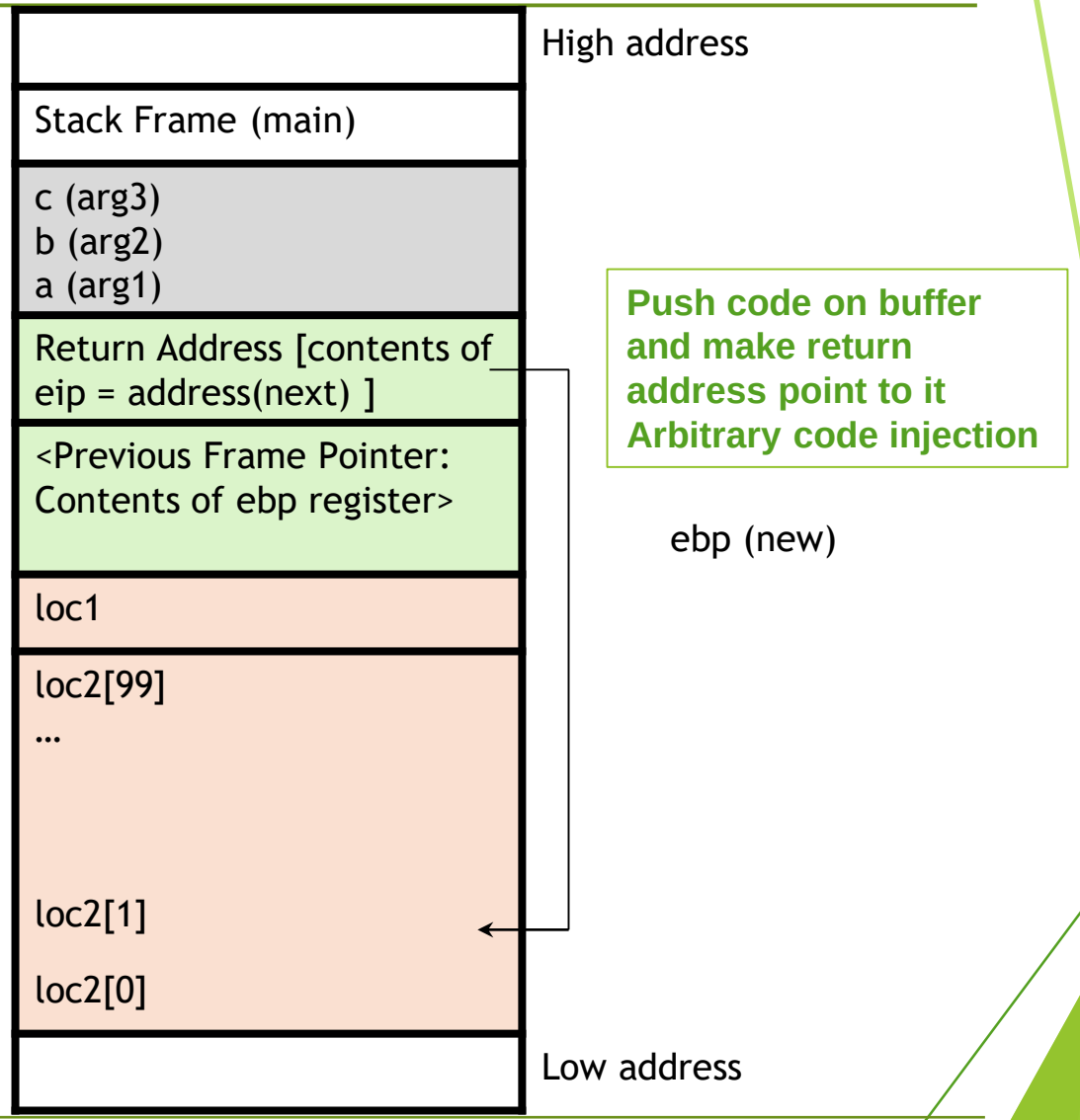
ebp (new)

Shellcode

- place the code in the buffer and overwrite return address so it points back into the buffer
  - code supplied as user input (aim is to cause buffer overflow and transfer control to this code)
  - machine code - specific to processor and operating system
- good assembly language skills no longer necessary as automated sites/tools available
- Most common usage: transfer control to a shell
- Use /bin/sh to open shell command on victim computer.
  - Same privileges as user program under attack
  - If system call has buffer overflow vulnerability, root privileges to shell code (privilege escalation)

Buffer Overflow: An example

```
void func(char *arg1, int arg2, int
arg3)
{
    int loc1, loc2[100];
        ...
    loc2++;
    ...
}
void main(int argc, char **argv) {
    …
    func(a,b,c);
    next
    …
}
```



Buffer Overflow

- To exploit a buffer overflow an attacker must identify a buffer overflow vulnerability in some program
  - inspection,
  - tracing execution,
  - fuzzing tools
- understand how buffer is stored in memory and determine potential for corruption

Shellcode

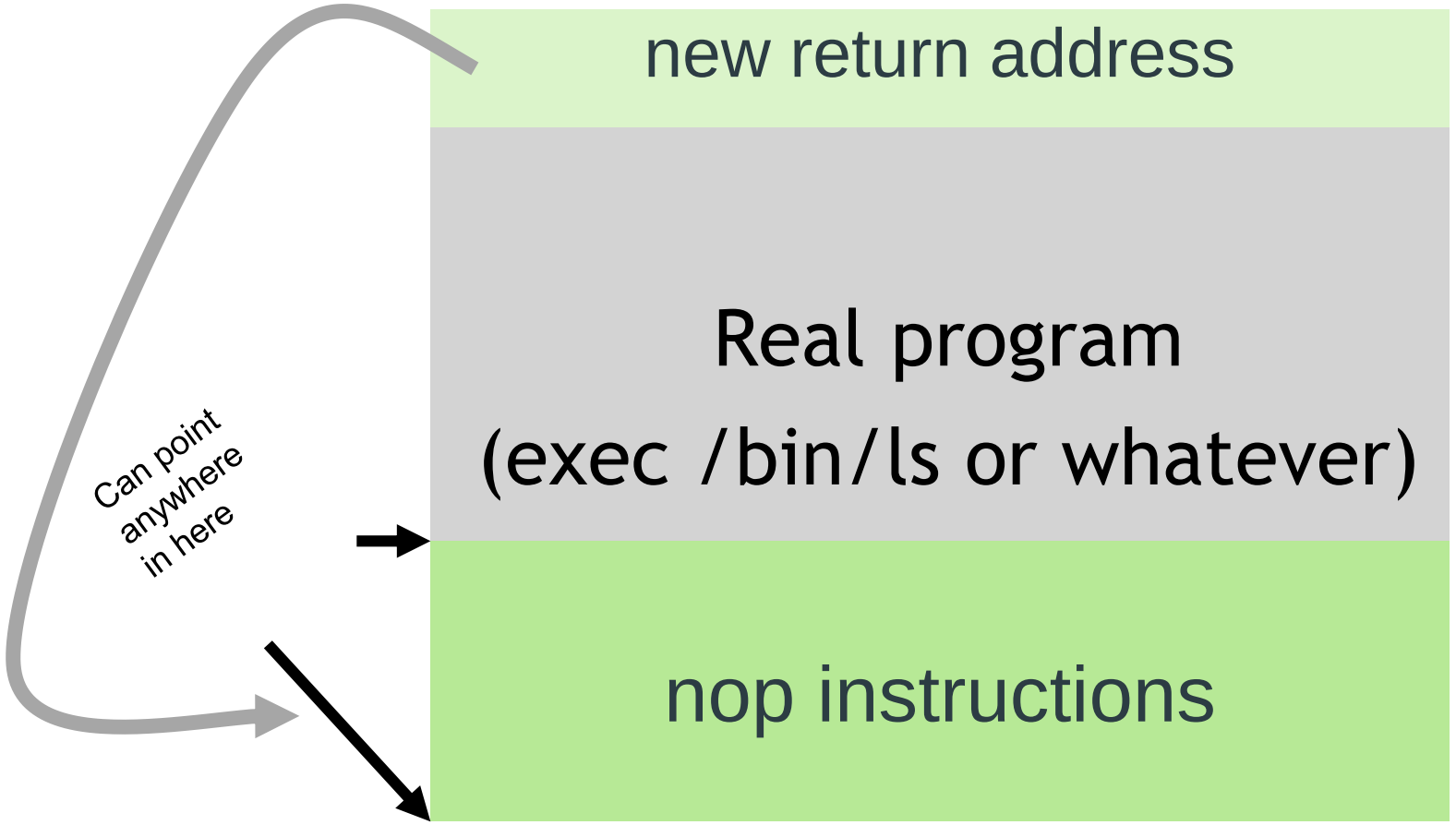
```
#include<stdio.h>
void main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0],name,NU
   LL);
}
```

- Compile the following to assembly language code
- Extract necessary code (sans activation frame; absolute addresses) and make necessary changes
- Convert to hex format (as per processor - little/big endian) and store as a string
- No null bytes as this is end of string (code may not be completely copied)
- String supplied as input

Locating Return Address

- No information on internals of the program: number and size of local variables.
  - No access to the code,
  - No information on how far the buffer is from the saved `%ebp`
- One approach: just try a lot of different values!
- Worst case scenario: it's a 32 (or 64) bit memory space, which means 2^{32} (2^{64}) possible answers

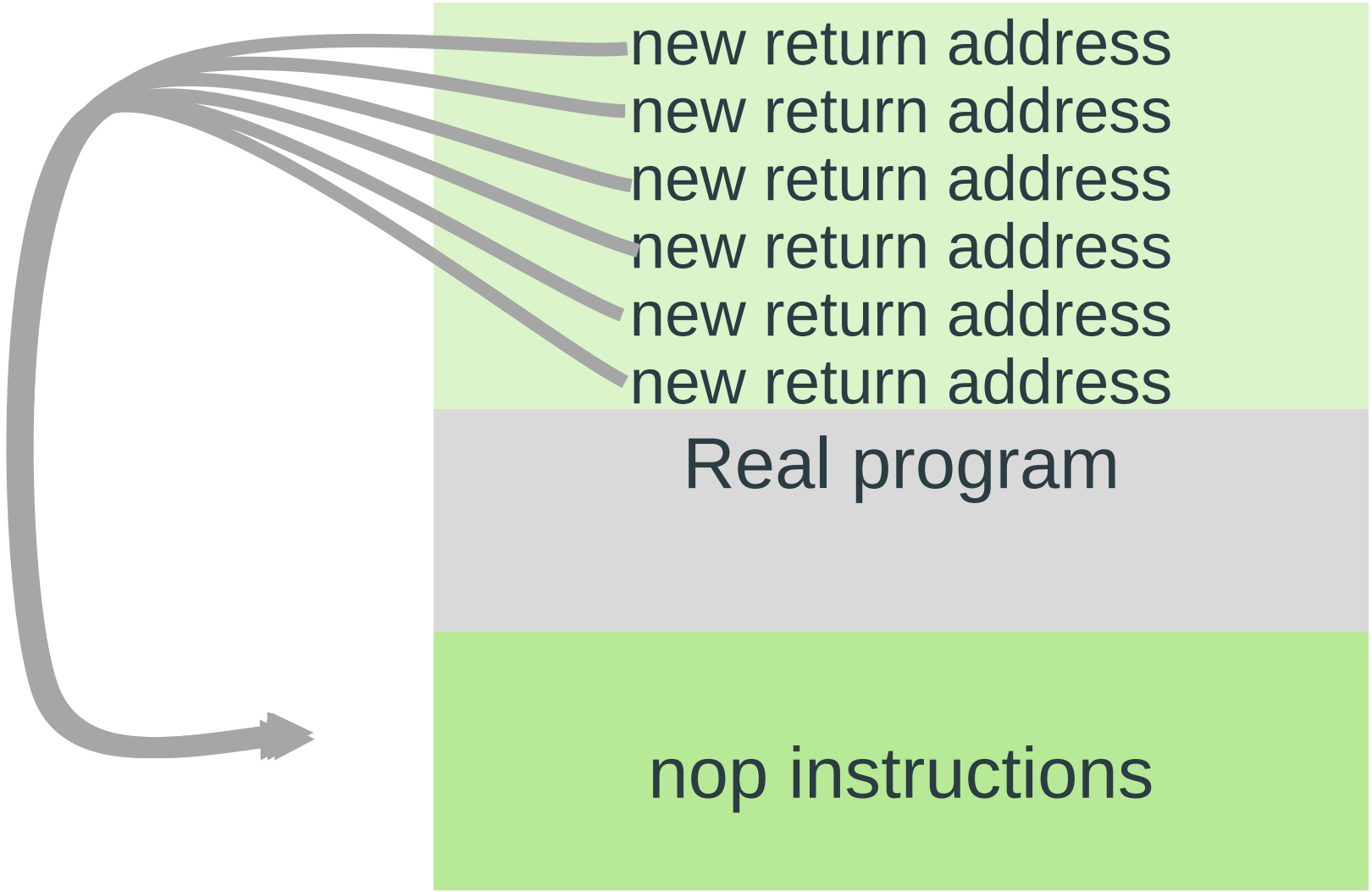
NOP Sleds


new return address
Real program
(exec /bin/ls or whatever)
nop instructions
Can point anywhere in here

NOP Sled

- Most CPUs have a No-Operation instruction – it does nothing but advance the instruction pointer.
- Usually we can put a bunch of these ahead of our program (in the string).
- As long as the new return-address points to a NOP we are OK.

Estimating the Location



Countermeasures

- compile-time - harden new programs
- run-time - handle attacks on existing programs

Compile Time Countermeasures

- strong typing; safe programming
- compiler enforces range checks and permissible operations on variables
- Canary/Stackgaurd: add function entry and exit code to check stack for signs of corruption
  - issues: recompilation, debugger support
- Shadow stack: Or save/check safe copy of return address (in a safe, non-corruptible memory area), e.g. Stackshield, RAD

Run-time Countermeasures

- Non-executable stack space
- Address Space Layout Randomization
  - shift location of key data structures (stack, heap, global data) randomly for each process
  - Randomize location of heap buffers and location of standard library functions
- Place guard pages
  - between critical regions of memory (between stack frames; between stack frames and heap buffers)
  - flagged in MMU (mem mgmt unit) as illegal addresses
  - any access aborts process

Famous Stack Overflow Attacks

- Morris Worm (1988): fingerd on VAX
- Codered (2001) : MS-IIS Server
- SQL Slammer (2003): MS SQL Server

Off by one

- Copying source string (length= buffer length) shall copy one null byte after buffer
- If this is part of saved frame pointer EBP, its LSB is set to zero (an address within buffer)
- This modified value shall be treated as frame when function returns
- Return address within this dummy buffer is in control of attacker
  - Arbitrary code execution
  - Useful technique in limited buffer overflow

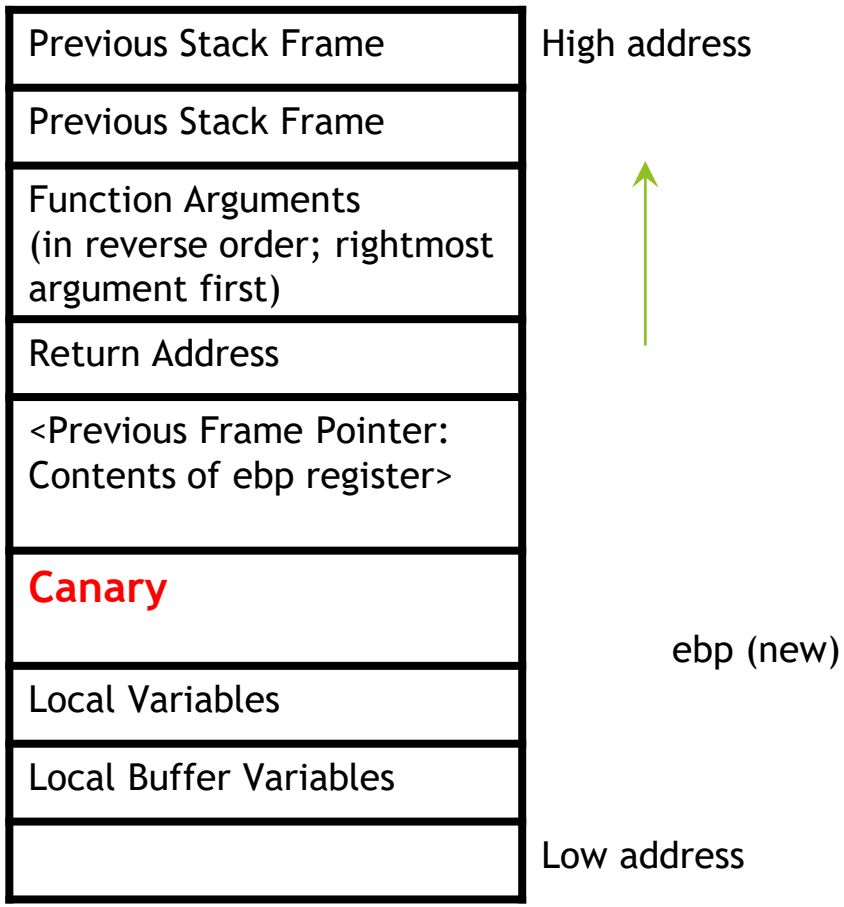
Replacement Stack Frame

- Input string creates a dummy frame within buffer;
- On overflow, saved (previous) frame pointer address is changed to point to this dummy stack frame; return address is not changed
- On return from called function, control goes to correct instruction but different stack frame
- In dummy frame, return address points to code within buffer
- On return from calling function, control is transferred to the shellcode in the overwritten buffer
- Need to protect both previous frame pointer as well as return address

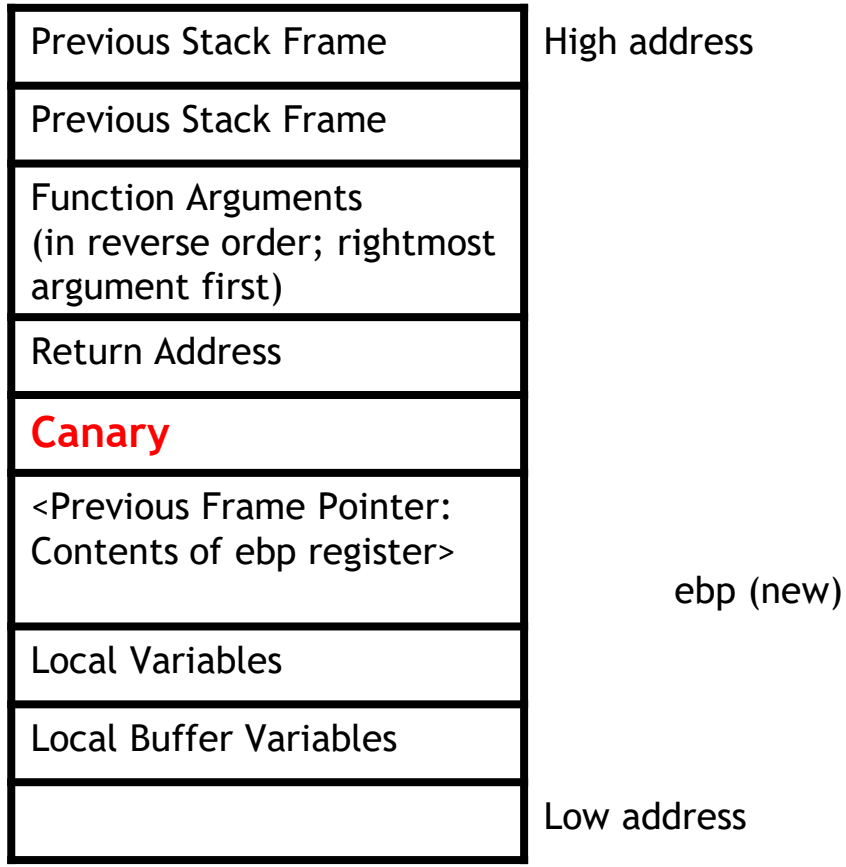
Stack Canary

- Like the legendary canary-in-the-mine, it detects stack smash attacks.
- Insert a “Canary value” in path from buffer to return address
- If return address is modified, canary is changed too
- At return from function, check if canary is changed.
- Canary can be
  - just below the return address (Stack Guard) or
  - just below the previous frame pointer (Stack Smashing Protector).

Stack Canary



Stack Smashing Protector: "Canary value" just below the previous frame pointer



StackGuard: "Canary value" just below the return address

Canary values

- Terminator canaries (CR, LF, NULL, -1)
  - Leverages the fact that scanf etc. don't allow these
- Random canaries
  - Write a new random value @ each process start
  - Save the real value somewhere in memory and write-protect this value
- Random XOR canaries
  - Same as random canaries
  - But store canary XOR some control info, instead
- Challenges:
  - protection depends on secrecy of canary value
  - No protection against stack data other than return address (memory corruption)
  - What if non-contiguous overwrites take place (bypassing canary)?

Shadow Stack

- a second stack that "shadows" stack
- At call, return address stored to both stacks
- At return, the return address from both stacks are compared
- do not protect stack data other than return addresses (memory safety errors)
- Shadow stacks themselves can be protected with guard pages

Non Executable Stack

- Mark stack addresses as non-executable: return to any address on stack shall fail to execute and result in error
- W XOR X: every page in address space (process/ kernel) is either writable or executable, but not both.
- Windows used DEP (data execution protection)
- NX (no-execute) bit supported by hardware for memory protection
  - XD (execute disable) bit in Intel
  - XN (execute never) in ARM
  - EVP (enhanced virus protection) in AMD
- Malware writers bypass this through “return to libc” attack

Return to libc

- return address is replaced with with address of a standard library function
- response to non-executable stack defences
- attacker constructs suitable parameters on stack above return address
- function returns and library function executes e.g. system(“shell commands”)
- attacker may need exact buffer address

Address Space Layout Randomization (ASLR)

- Basic idea: change the layout of the stack
- Patch at the kernel level, changing the memory mapping
- Small performance penalty, by extra memory lookups (actually, extra cache lookups)
- Makes it very difficult to perform a useful buffer overflow

ASLR

Run #1	Run #2	Run #3
Stack	Stack	Heap
Libraries (libc)		
	Heap	Stack
Heap		
	.rodata segment	.rodata segment
.rodata segment	Libraries (libc)	
		.text segment
.text segment		
	.text segment	ELF executable
		Libraries (libc)
ELF executable	ELF executable	
Runtime Memory	Runtime Memory	Runtime Memory

Safe Libraries

- Many vulnerabilities in code are due to unsafe use of system libraries
- An alternative is to install a kernel patch that dynamically substitutes calls to unsafe library functions for safe versions of those
- Not possible for closed-source systems such as MS operating systems

Defenses

- Putting code into the memory (no zeroes)
  - Option: Make this detectable with canaries
- Getting %eip to point to our code (dist but to eip)
  - Non-executable stack doesn't work so well
- Finding the return address (guess the raw addr) for example through return to libc
  - Address Space Layout Randomization

- Good coding practices

Thank you.