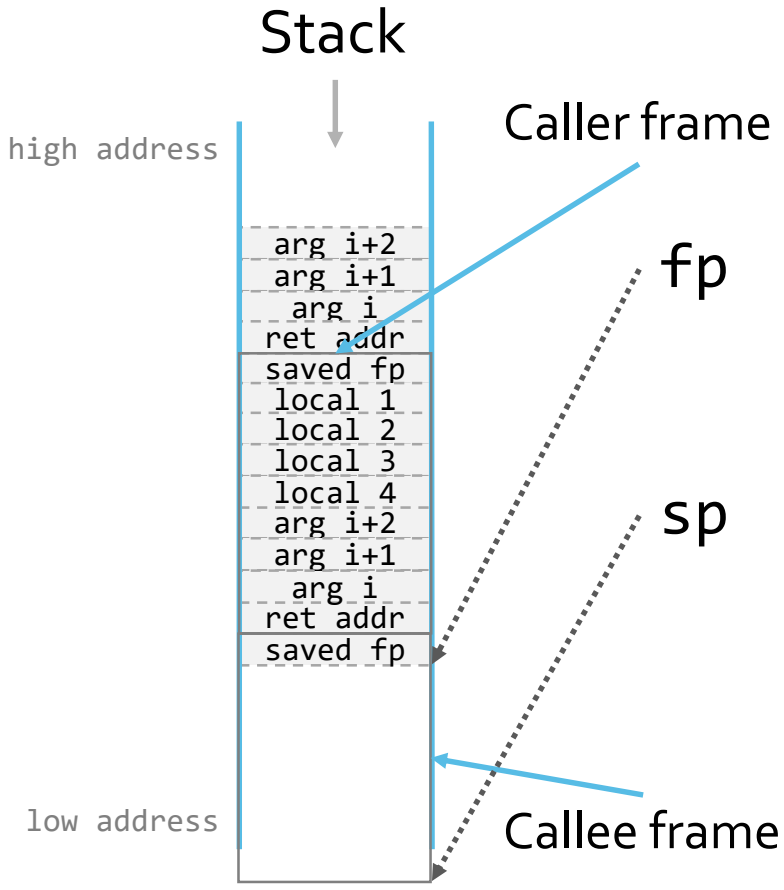
CSE 127 Computer Security

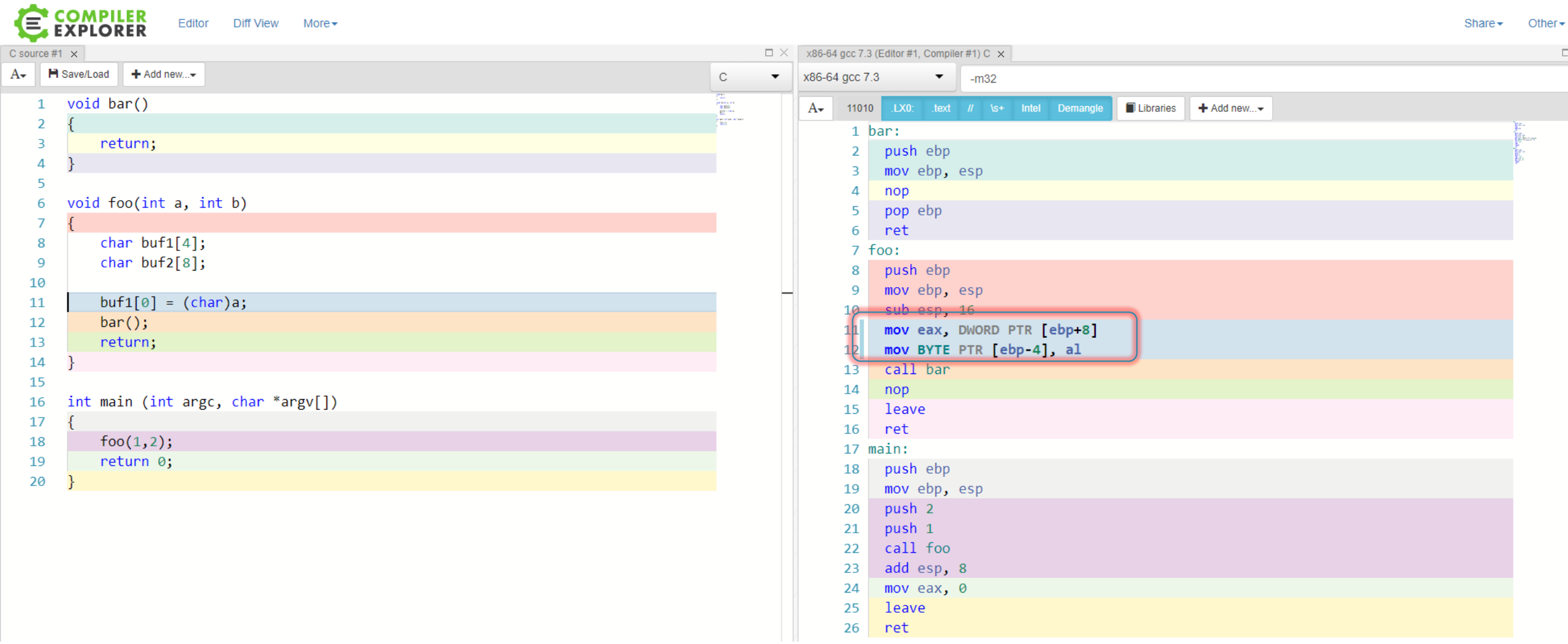
Alex Gantman, Spring 2018, Lecture 4

Low Level Software Security II: Format Strings, Shellcode, & Stack Protection

Review

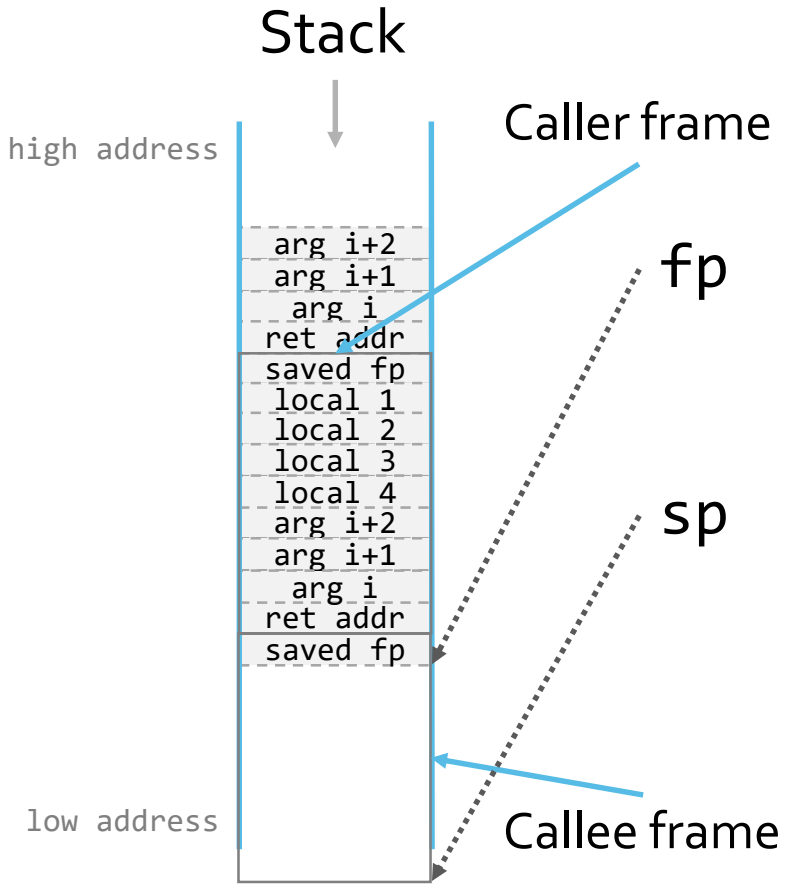
- Function arguments and local variables are stored on the stack
  - Next to control flow data like the return address and saved frame pointer
- Function arguments and local variables are accessed by providing relative to the frame pointer




COMPILER EXPLORER
Editor
Diff View
More
Share
Other
C source #1
Save/Load
Add new...
C
void bar()
{
return;
}
void foo(int a, int b)
{
char buf1[4];
char buf2[8];
buf1[0] = (char)a;
bar();
return;
}
int main (int argc, char *argv[])
{
foo(1,2);
return 0;
}
x86-64 gcc 7.3 (Editor #1, Compiler #1) C
x86-64 gcc 7.3
-m32
11010
.LX0:
.text
//
\s+
Intel
Demangle
Libraries
Add new...
bar:
push ebp
mov ebp, esp
nop
pop ebp
ret
foo:
push ebp
mov ebp, esp
sub esp, 16
mov eax, DWORD PTR [ebp+8]
mov BYTE PTR [ebp-4], al
call bar
nop
leave
ret
main:
push ebp
mov ebp, esp
push 2
push 1
call foo
add esp, 8
mov eax, 0
leave
ret

Review

- Implicit agreement between the caller and the callee about the number, size, and ordering of function arguments.
  - #include function declaration
- What happens if function declaration differs from actual implementation?
  - If the function is called with more or fewer arguments than expected?



Format String Vulnerabilities

printf()

- `printf("Diagnostic number: %d, message: %s\n", j, buf);`
  - "*If format includes format specifiers (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.*"
  - Also, `sprintf ( char * str,    const char * format, ... );`
    and `fprintf ( FILE * stream, const char * format, ... );`

- Format specifier: `%[flags][width][.precision][length]specifier`
- Specifier
  - Type and interpretation of the corresponding argument
  - Examples:
    - `%s`: string
    - `%d`: signed decimal integer
    - `%x`: unsigned hexadecimal integer
- Flags
  - Sign, padding, justification, …

printf()

- Format specifier:
  `%[flags][width][.precision][length]specifier`
- Width
  - Minimum width of printed argument in characters (can be indirect)
- Length
  - Size of argument
  - Examples:
    - `h: char`
    - `hh: short`
    - `l: long`
    - `ll: long long`

Variadic Functions

- So, how many arguments does `printf` take?
  - `int printf ( const char * format, ... );`
- C supports functions with a variable number of arguments.
- If the number of arguments is not pre-determined, how does the called function know how many were passed in?
  - Another argument explicitly specifies count
  - Another argument implicitly encodes count
    - Eg. format string
  - The last argument is a reserved terminator value
    - Eg. NULL

Format String Vulnerabilities

- What's the difference between the following two lines?
  - `printf("%s", buf);`
  - `printf(buf);`
- You can think of these functions as interpreting commands specified in the format string.
- Not a good idea to let an attacker feed arbitrary commands to your command interpreter.

Format String Vulnerabilities

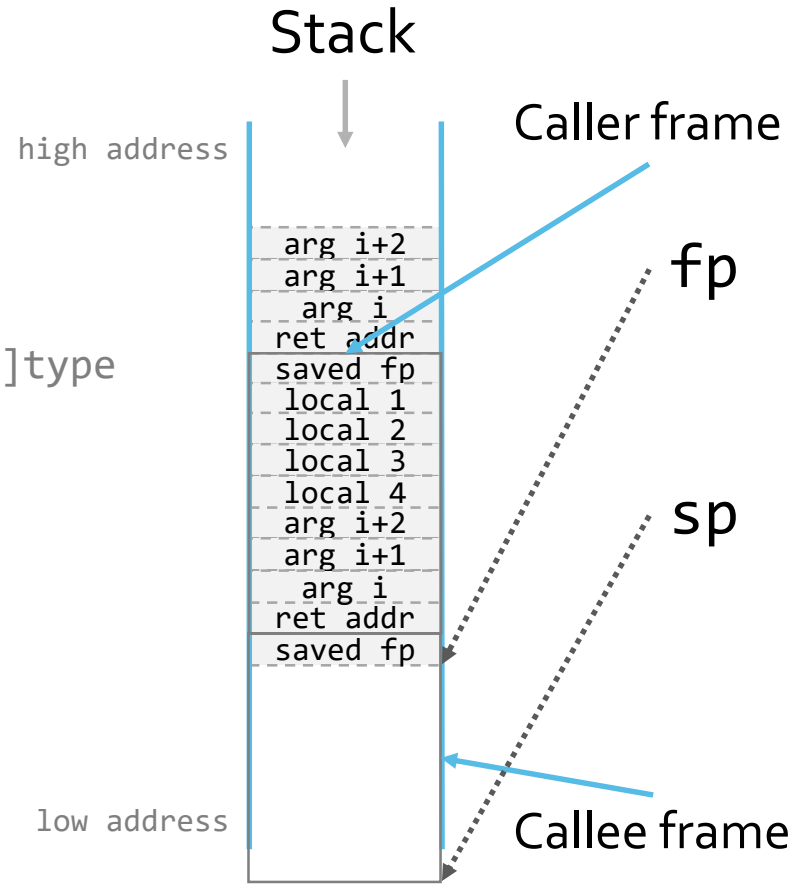
- Still, how bad could it be?
- What can an attacker do with just a format string?
  - Read arbitrary memory
  - Write arbitrary memory

Format String Vulnerabilities: Reading

`%[flags][width][.precision][length]specifier`

- Reading from the stack
  - `printf("%08x.%08x.%08x.%08x\n");`
  - `printf("%9$08x\n");`
    - POSIX extension to reference parameters by number
    - `%[parameter][flags][width][.precision][length]type`

- Reading via a pointer
  - `printf("%s\n");`



Format String Vulnerabilities: Writing

- How to use format strings for writing?
- Buffer overflow
  - `if (strlen(src) < sizeof(dst)) sprintf(dst, src);`
  - What if `src` contains format specifiers?

```
sprintf ( char * str,    const char * format, ... );
```

Format String Vulnerabilities: Writing

- Arbitrary write
  - `%n`
  - "*Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.*"
    - ```
      int x = 0;
      printf("Hello %n ", &x);  // after call x == 6
      ```
- Although it may have some valid uses, because of the danger it poses, this format specifier is disabled by default on many modern systems.
      ```

Format String Vulnerabilities: Writing

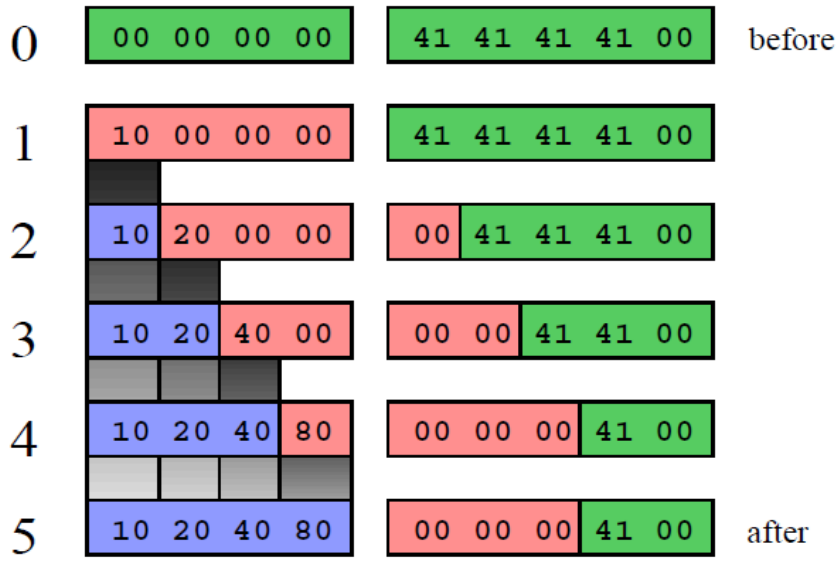
- Value that we really want to overwrite is likely a pointer (like the return address)
  - How to write a large 4-byte integer with %n?
- Use the width specifier to programmatically pad output to necessary size
  - May work with printf(), but not with sprintf()
  - Why?

Format String Vulnerabilities: Writing

- Unaligned writes FTW!
  - xxxxxx12345678
  - xxxxxx00000078
  - xxxx00000056
  - xx00000034
  - 00000012
- Can be done in a single pass with increasing counts!
  - xxxxxx12345678
  - xxxxxx00000078
  - xxxx00000156
  - xx00000234
  - 00000312

- Actually, little-endian

Figure 1: Four stage overwrite of an address



https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

Format String Vulnerabilities: Writing

```
%[flags][width][.precision][length]specifier
```

- Another trick: size modifiers.
  - %hn writes a half-word
  - %hhn writes a single byte
  - Not universally supported

Avoiding Format String Vulnerabilities

- gcc has optional warning levels that will alert on problematic usage of format specifiers:
  - `-Wformat`: warn if format specifiers match arguments
  - `-Wformat-overflow`: warn if destination might overflow
  - `-Wformat-security`: warn if format string is not a string literal
  - And many others

Additional Resources

- *Exploiting Format String Vulnerabilities* by scut / team teso
  - https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

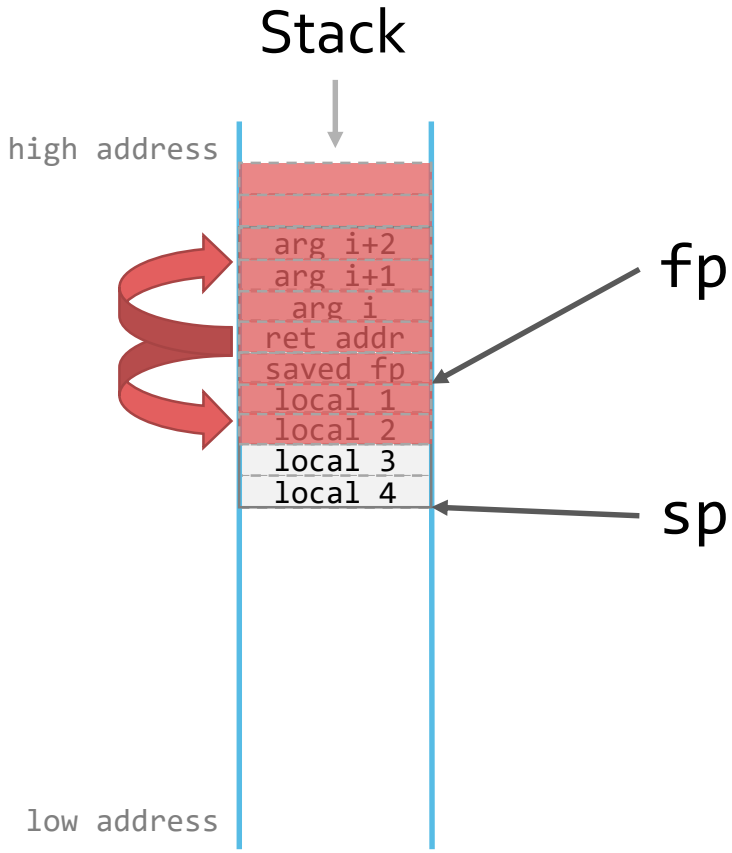
Review

- Don’t mix commands and data.
- Functions that take format strings act like command interpreters.
- Don’t let attackers decide which commands to pass to your command interpreters.

Shellcode

Review: Smashing The Stack

- Overwriting the return address
  - Upon function return, control is transferred to an attacker-chosen address
  - Arbitrary code execution
    - Attacker can re-direct to their own code



Shellcode

- What to do after we figure out how to seize control of the instruction pointer?
- Ideally, redirect to our own code!
- But what should that code be?
- Spawning a shell would provide us with full privileges of the victim process
  - Hence, "***shellcode***"

Shellcode

- How to spawn a shell?
- "*The exec family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the new process image file.*"
- Just need to call `execve` with the right arguments
  - `execve("/bin/sh", argv, NULL)`

Shellcode

- Note the tricks Aleph One uses:
  - Writing shellcode in C
    - Compile and run in debugger to review object code
    - Adjust references to strings, etc.
  - Using a `call` instruction to infer the address of payload on the stack
    - `call` will push the address of the next word onto the stack as a return address

```
void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

http://phrack.org/issues/49/14.html

Shellcode

- Note the tricks Aleph One uses:
  - Inline assembly to use gcc to translate from assembly to object code
    - Compile and run in debugger to review object code

```
void main() {
__asm__(“
        jmp    0x1f                    # 2 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        xorl   %eax,%eax               # 2 bytes
        movb   %eax,0x7(%esi)          # 3 bytes
        movl   %eax,0xc(%esi)          # 3 bytes
        movb   $0xb,%al                # 2 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        xorl   %ebx,%ebx               # 2 bytes
        movl   %ebx,%eax               # 2 bytes
        inc    %eax                    # 1 bytes
        int    $0x80                   # 2 bytes
        call   -0x24                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
                                       # 46 bytes total
");
}
```

http://phrack.org/issues/49/14.html

Shellcode

- Note the tricks Aleph One uses:
  - Testing shellcode standalone
    - Encode shellcode into a data buffer
    - Set the return address on the stack to point to your shellcode
  - Eliminating `0x00` from the shellcode
    - Find alternate instruction representations
  - Using a NOP sled
    - Relaxes constraints on guessing the exact location of the shellcode to put into the overwritten return address

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x0
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x0
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x8

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```

http://phrack.org/issues/49/14.html

Shellcode

- That works well for local attacks
  - When the victim is another process on the same machine
- What about remote attacks?
- Similar concept, just a few more system calls in the shellcode
  - Reverse
    - Connect back to your malicious server and present a remote shell
  - Bind
    - Open a port and wait for connections, present shell
  - Reuse
    - Re-use existing connection

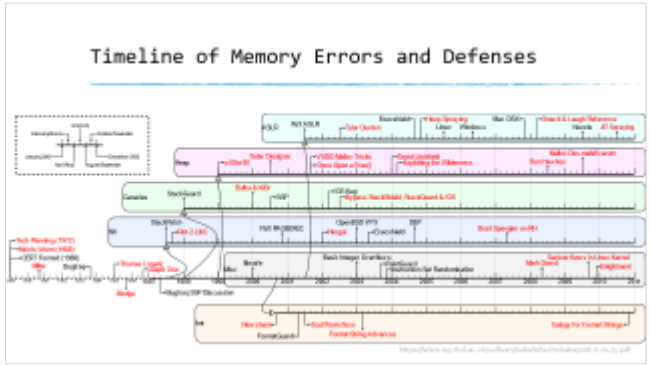
Stack Protection

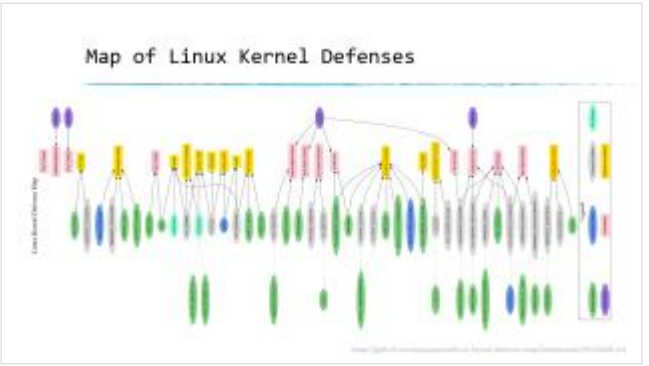
Countermeasures and Mitigations

- Asking developers to not insert vulnerabilities has not worked.
  - People will make mistakes and introduce vulnerabilities during development.
  - Not all of these vulnerabilities will be discovered prior to release.
- As the next line of defense, countermeasures are introduced that make reliable exploitation harder or mitigate the impact of remaining vulnerabilities.
  - Will not stop all exploits.
  - Make exploit development more difficult and costly.
  - Mitigations are important, but they should not be relied upon. It's much better to write code properly!
- Ongoing arms race between defenders and attackers
  - Co-evolution of defenses and exploitation techniques

Countermeasures and Mitigations

- As we consider different mitigations:
  - Challenge assumptions.
  - Keep thinking of how you can circumvent each "solution".
- Security is an ongoing arms race.
  - Developers introduce new features. Attackers devise ways to exploit these features. Defenders devise new countermeasures or mitigations. Attackers adapt to the new countermeasures, the defenders refine their approach, and the cycle continues with no end in sight…
    - … and full employment on all sides



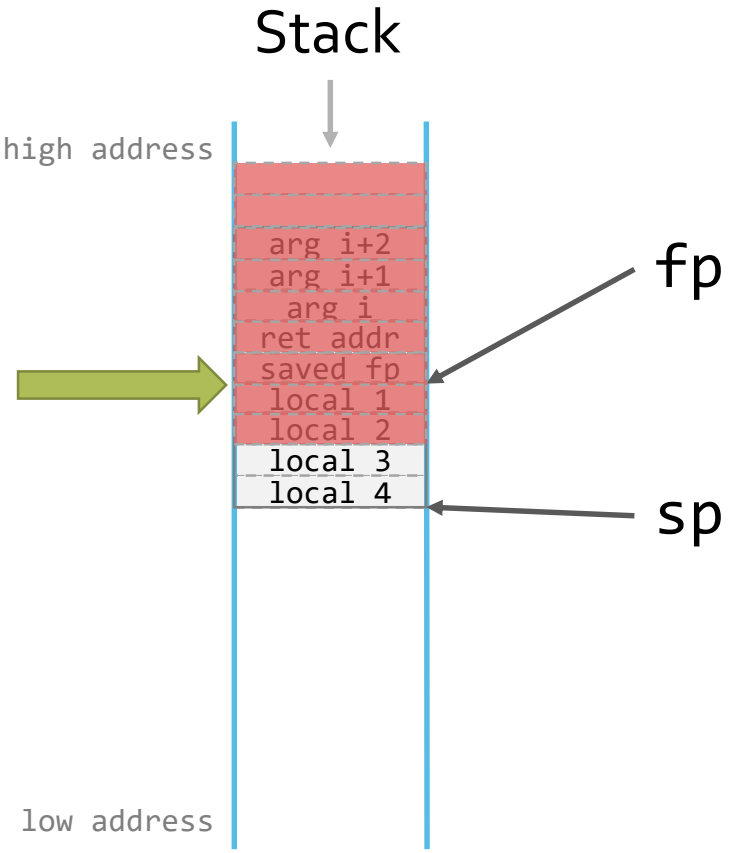


Countermeasures and Mitigations

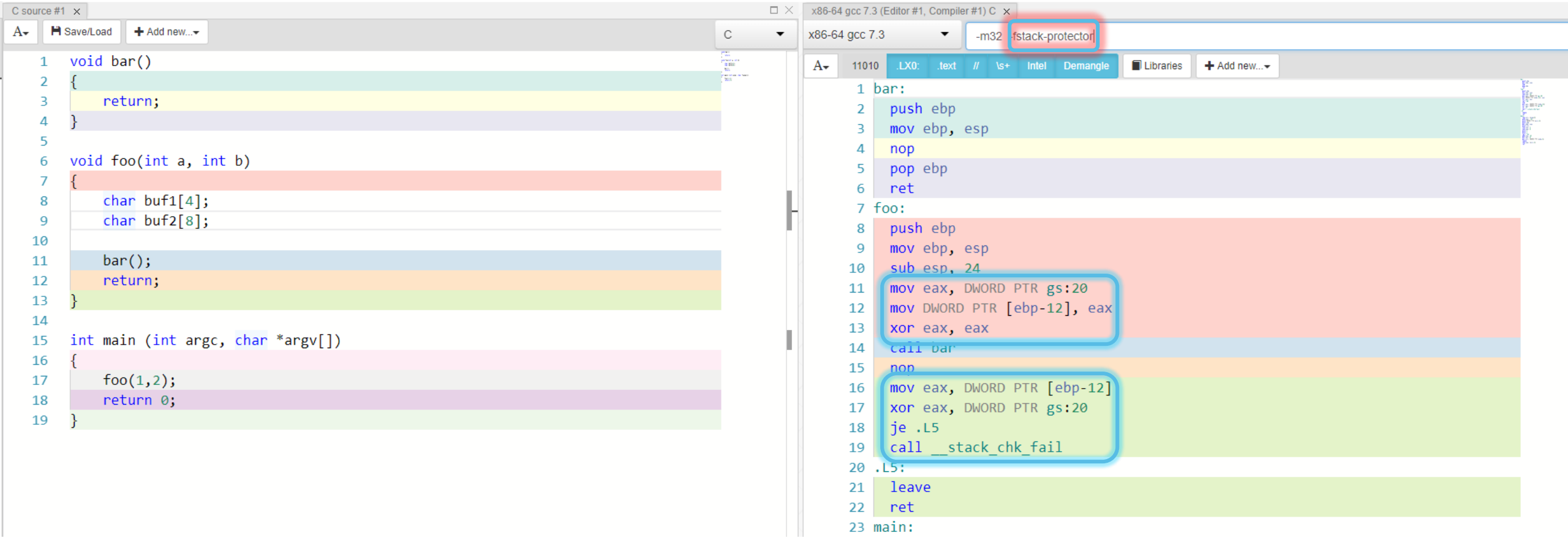
- What do we want to prevent?
  - Overwriting of the return address?
  - Hijacking of the control flow?
  - All out of bounds memory access?

Stack Canary

- Detect overwriting of the return address
  - Place a special value (aka ***canary*** or cookie) between local variables and saved frame pointer.
  - Check that value before popping saved frame pointer and return address from the stack.

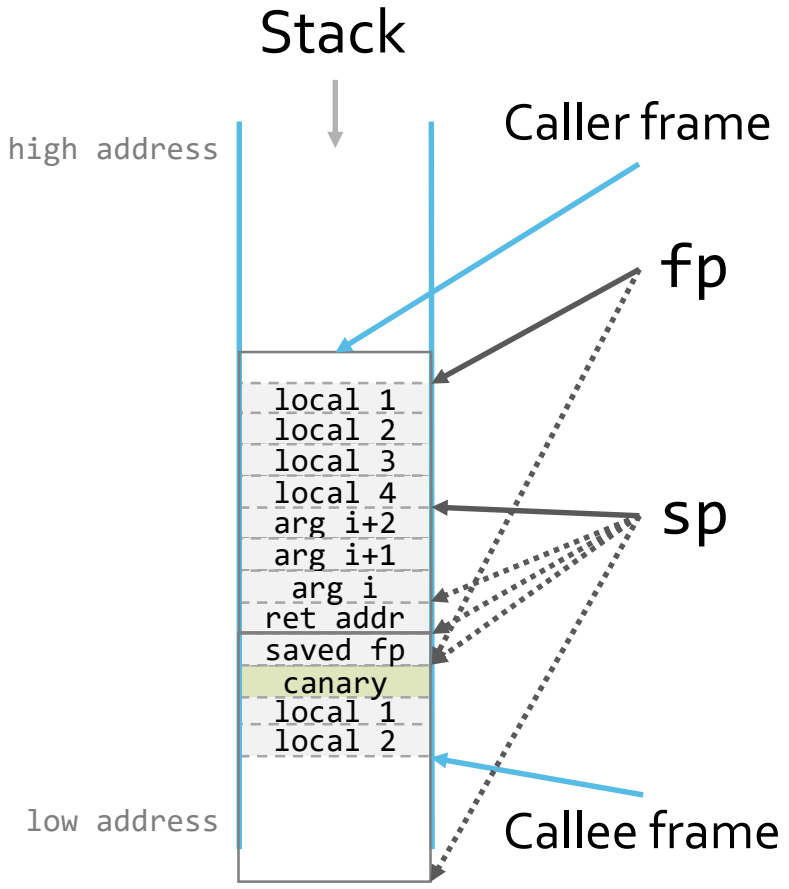


Stack Canary



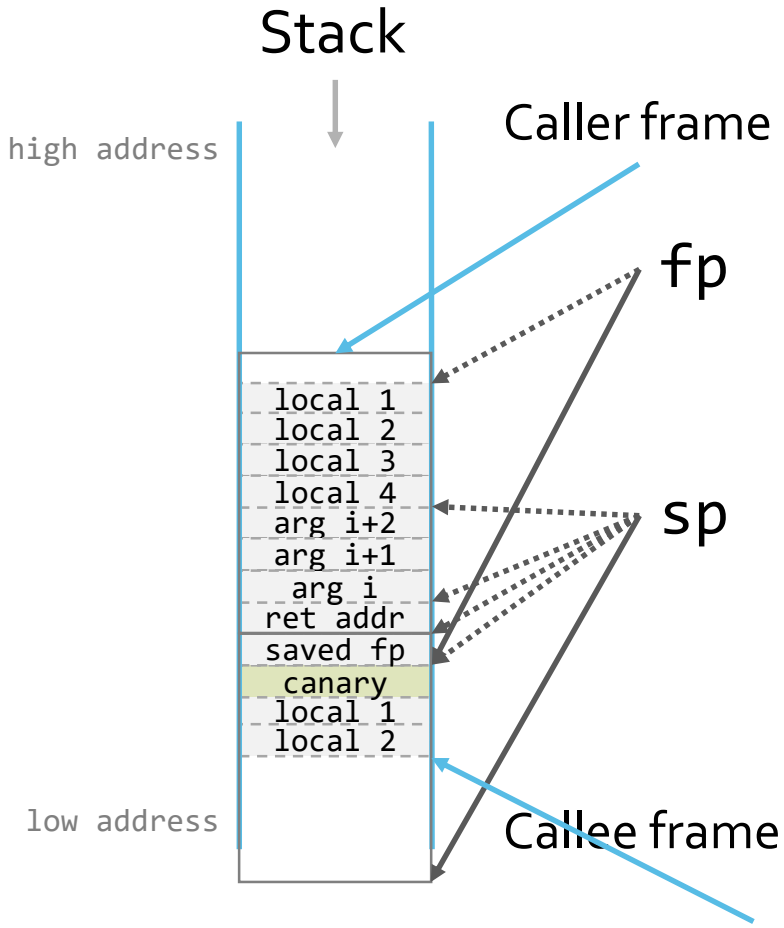
Stack Canary

- Calling a function
  - Caller
    - Pass arguments
    - Call and save return address
  - Callee
    - Save old frame pointer
    - Set frame pointer = stack pointer
    - Allocate stack space for local storage + space for the canary
    - Push canary



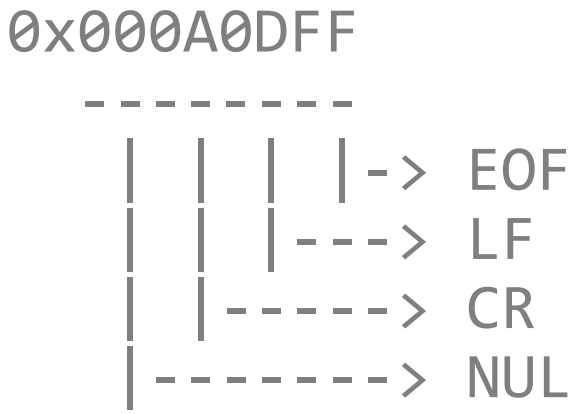
Stack Canary

- When returning
  - Callee
    - Check canary against a global gold copy
      - Jump to exception handler if !=
    - Pop local storage
      - Set stack pointer = frame pointer
    - Pop frame pointer
    - Pop return address and return
  - Caller
    - Pop arguments



Stack Canary

- What value should we use for the canary?
- What about `0x000A0DFF`?
  - ***Terminator Canary***
  - Hard to insert via string functions
- What's the problem with using a fixed value?
- Other options?
- Rather than making canaries hard-to-insert, we can make them hard-to-guess.
  - ***Random canaries*** are secure as long as they remain secret.



Stack Canary

C source #1

Save/Load | Add new... | C

```c
void bar()
{
    return;
}

void foo(int a, int b)
{
    char buf1[4];
    char buf2[8];

    bar();
    return;
}

int main (int argc, char *argv[])
{
    foo(1,2);
    return 0;
}
```

x86-64 gcc 7.3 (Editor #1, Compiler #1) C

x86-64 gcc 7.3 | -m32 -fstack-protector

11010 | .LX0: | .text | // | \s+ | Intel | Demangle | Libraries | Add new...

```asm
bar:
  push ebp
  mov ebp, esp
  nop
  pop ebp
  ret
foo:
  push ebp
  mov ebp, esp
  sub esp, 24
  mov eax, DWORD PTR gs:20
  mov DWORD PTR [ebp-12], eax
  xor eax, eax
  call bar
  nop
  mov eax, DWORD PTR [ebp-12]
  xor eax, DWORD PTR gs:20
  je .L5
  call __stack_chk_fail
.L5:
  leave
  ret
main:
```

Stack Canary Limitations

- How can stack canaries be bypassed?
  - Assumption: impossible to subvert control flow without corrupting the canary.
  - Challenge it!
  - Is it possible to overwrite the canary with a valid canary value?
  - Is it possible to overwrite non-protected data?
  - Is it possible to overwrite critical data without overwriting the canary?

Stack Canary Limitations

- Can an attacker overwrite the canary with the correct value?
  - Are terminator canaries impossible to insert?
    - Length-bound loops, `memcpy()`, etc.
  - How random are random canaries? Can an attacker guess them?
    - Network services that fork child processes to handle connections.
    - `fork()` and `execve()` in the child process to generate a new secret value for the canary.
  - How secret are random canaries? Can an attacker leak them?
    - An ***information leak*** might disclose the value of the canary.

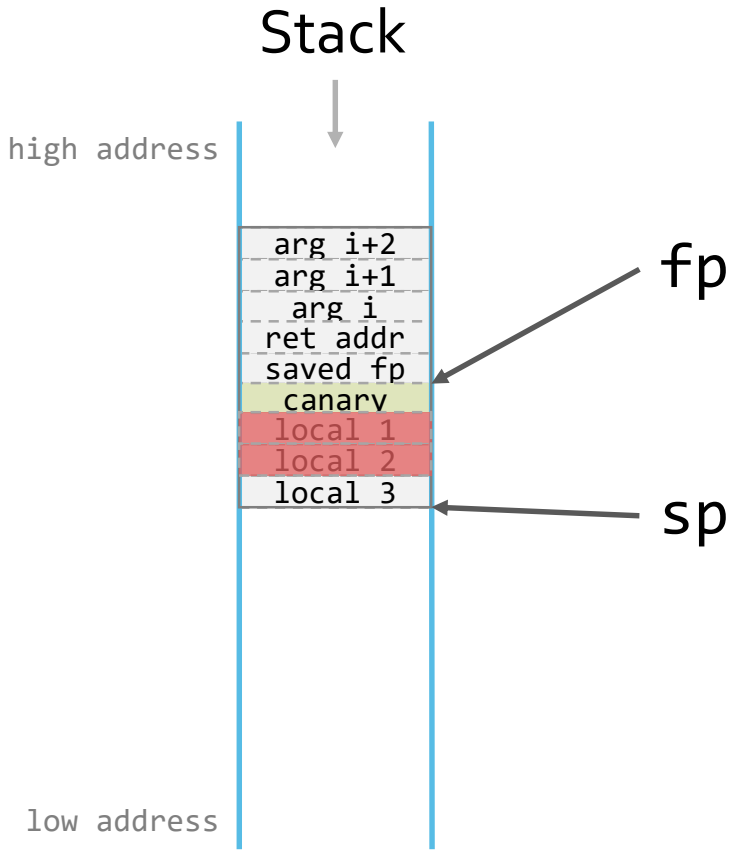
Information Leak

- Reading outside the bounds can also be a security issue.
- If the read data is returned to the attacker, it could disclose sensitive information and allow further exploitation.
  - Examples?
- "Chaining" multiple vulnerabilities is common for modern exploits.

http://heartbleed.com/

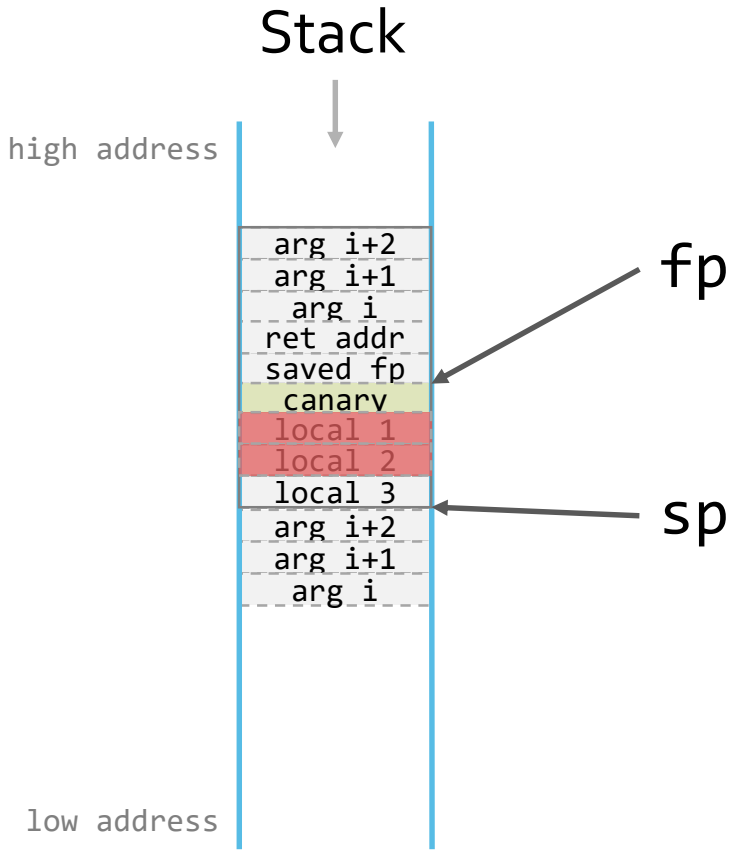
Stack Canary Limitations

- Can an attacker overwrite something that is not protected by canaries?
  - Local variables.
    - Variables that store result of a security check
      - Eg. isAutheticated, isValid, isAdmin, etc.
    - Variables used in security checks
      - Eg. buffer_size, etc.
    - Data pointers
      - Potential for further memory corruption
    - Function pointers
      - Direct transfer of control when function is called through overwritten pointer
  - Exception control data.



Stack Canary Limitations

- What can be done to protect local variables?
  - Some implementations reorder the local variables.
  - Buffers placed closest to the canary.
- What about function arguments?
  - When do you check the canary?
  - And when do you use the arguments?
  - Some implementations copy the arguments to the top of the stack to make overwriting them from local variables less likely.



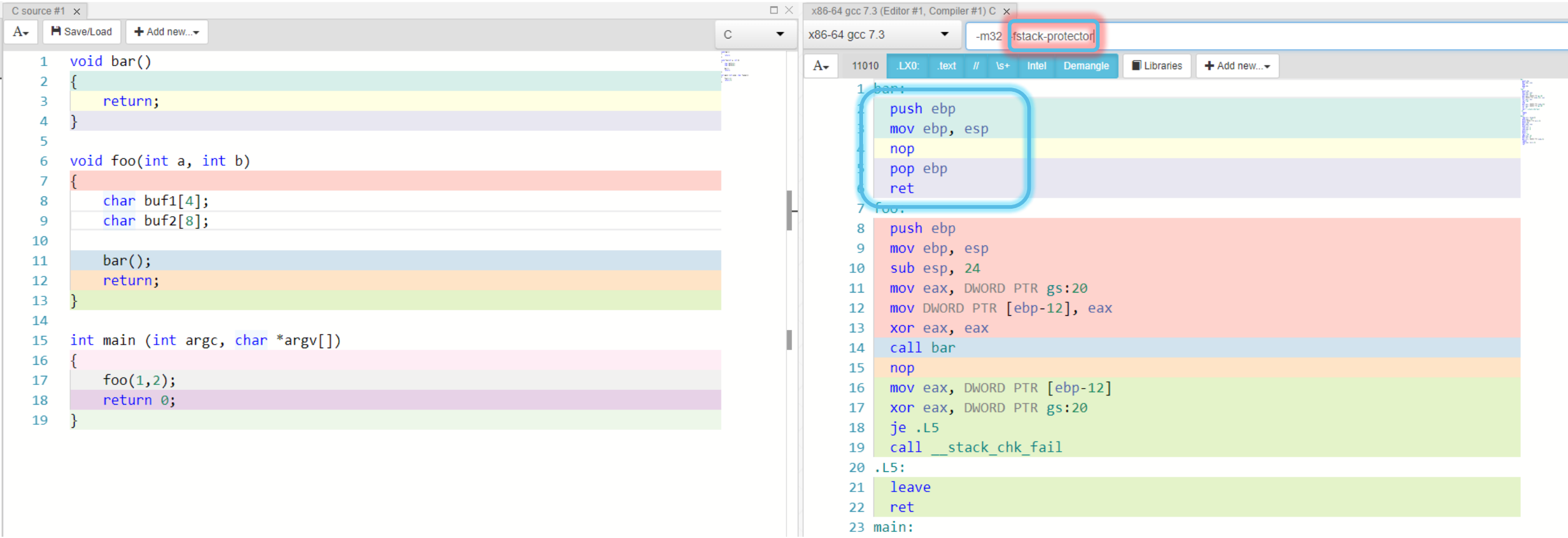
Stack Canary Limitations

- Stack canaries only protect against stack buffer overflows.
- Stack canaries do not protect from non-sequential overwrites.
  - Examples?
  - Indirect pointer overwriting
    - Overwrite a pointer (perhaps another local variable) that will later be used to for writing data and make it point to the saved return address on the stack (past the canary)
- Vulnerable to information disclosure attacks
  - Random canary fails if attacker can read memory
- Stack canaries do not prevent the overwrite. They only attempt to detect it once it happens.
  - How to recover?

Stack Canary Limitations

- Requires compiler support.
  - Supported by most modern compilers.
- Requires access to a good random number at runtime.
- Increases code size and adds performance overhead.
  - Small penalty on function entry and exit.

Stack Canary Limitations



Stack Canary Limitations

- Most implementations do not instrument every function.
  - Code size and performance overhead.
- Which functions get canaries?
  - `gcc -fstack-protector`
    - Functions with character buffers >= 8 bytes, functions that call `alloca()`
  - `gcc -fstack-protector-strong`
    - Above + functions with local arrays (of any type or size), functions that have references to local stack variables
  - `gcc -fstack-protector-all`
    - All functions

Stack Canaries

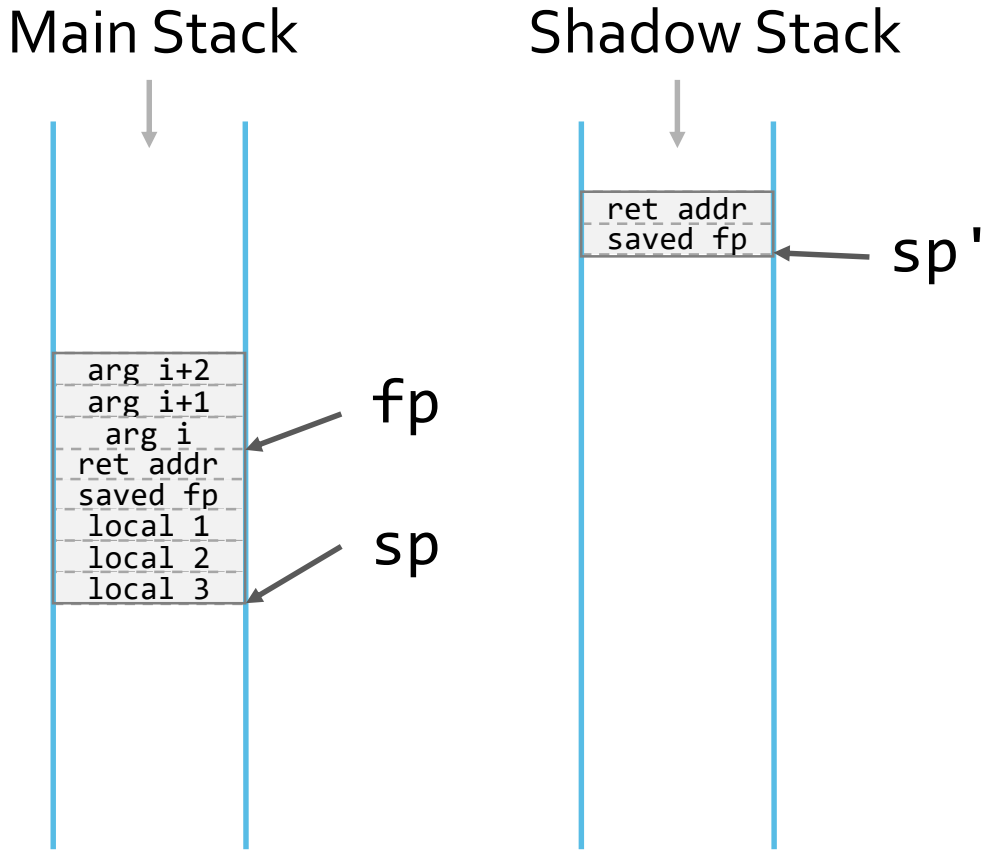
- In spite of the above limitations, stack canaries still offer significant value for relatively little cost.
- Considered essential mitigation on modern systems.

Shadow Stack/Split Stack

- Stack smashing attacks take advantage of the fact that control data is stored next to user data.
- Not a good idea™ from a security point of view.

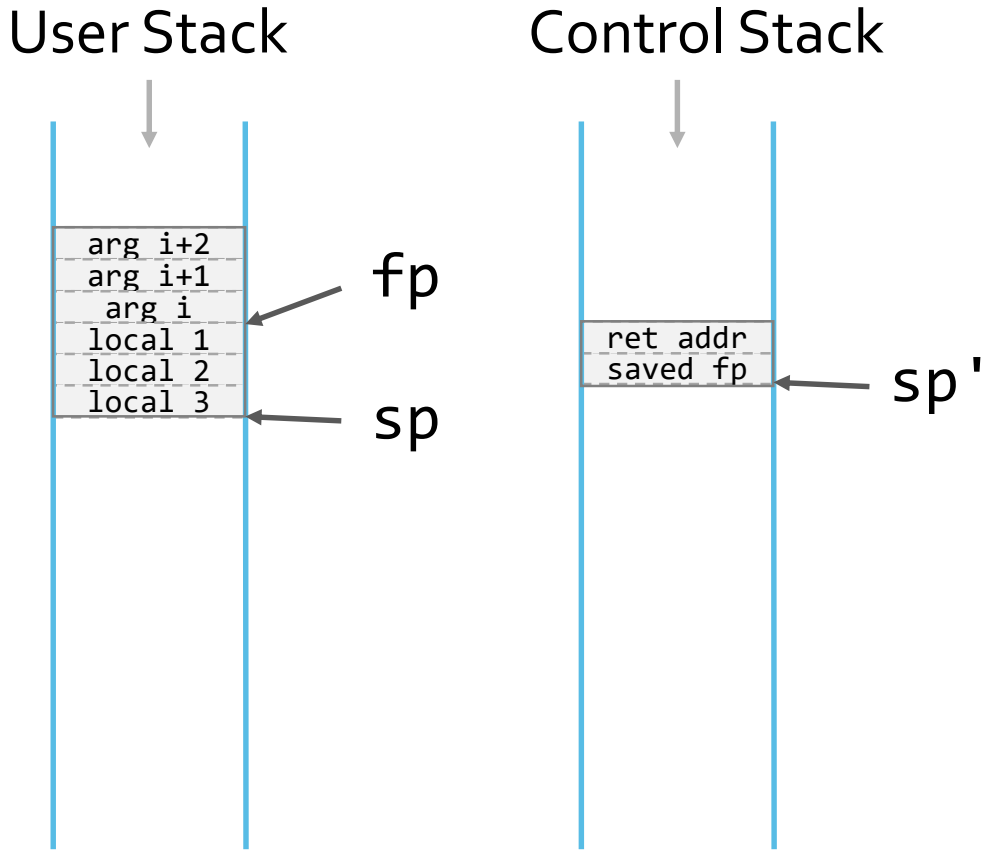
Shadow Stack/Split Stack

- Shadow Stack
  - On function entry, save a [shadow] copy of function call control flow data (return addresses and frame pointers) into another location
  - On function exit, compare the version on the stack to the shadow copy



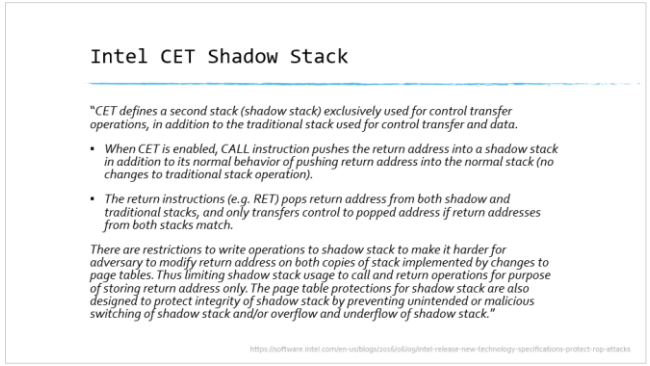
Shadow Stack/Split Stack

- Split Stack
  - Separate the storage of function call control flow data (return addresses and frame pointers) from that of user data (function arguments and local variables)



Shadow Stack/Split Stack Limitations

- How can this mitigation be bypassed?
  - Overwriting other data
  - No-sequential overwrites into the shadow stack
- Requires compiler and hardware support to be efficient.
  - Not widely deployed.



Intel CET Shadow Stack

"CET defines a second stack (shadow stack) exclusively used for control transfer operations, in addition to the traditional stack used for control transfer and data.

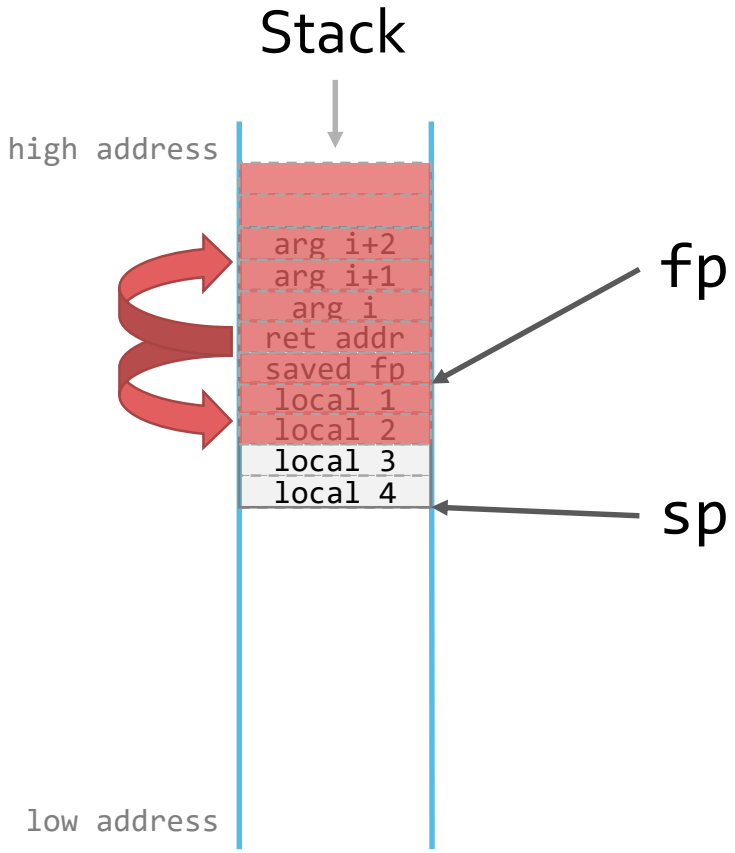
- *When CET is enabled, CALL instruction pushes the return address into a shadow stack in addition to its normal behavior of pushing return address into the normal stack (no changes to traditional stack operation).*
- *The return instructions (e.g. RET) pops return address from both shadow and traditional stacks, and only transfers control to popped address if return addresses from both stacks match.*

There are restrictions to write operations to shadow stack to make it harder for adversary to modify return address on both copies of stack implemented by changes to page tables. Thus limiting shadow stack usage to call and return operations for purpose of storing return address only. The page table protections for shadow stack are also designed to protect integrity of shadow stack by preventing unintended or malicious switching of shadow stack and/or overflow and underflow of shadow stack."

https://software.intel.com/en-us/blogs/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks

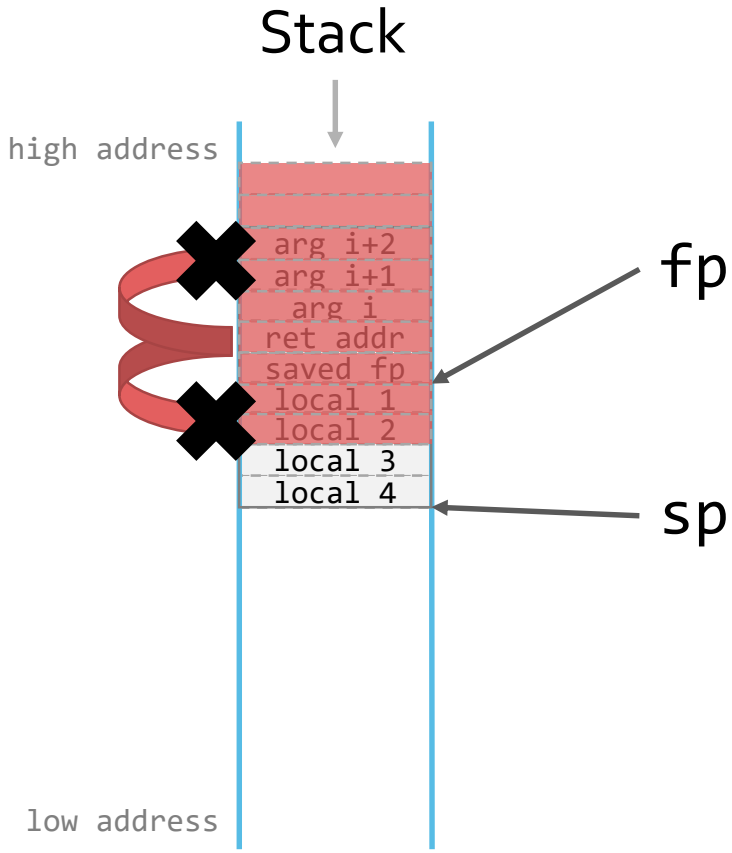
Nonexecutable Stack

- Other ideas?
- Virtual memory pages have permission settings for Read, Write, and eXecute access



Nonexecutable Stack

- Prevents execution of shellcode from the stack.
- Attempts to execute instructions from the stack trigger memory access violations.



Nonexecutable Stack

- Modern processors actually go further and support marking memory pages as readable and writable (for data), readable and executable (for code), but never both writeable and executable at the same time.
- Also known as
  - XN: eXecute Never
  - W^X: Write XOR eXecute
  - DEP: Data Execution Prevention

Data Execution Prevention (DEP)

- Mitigation extends beyond the stack.
- Make all pages either writeable or executable, but not both.
  - Stack and heap are writeable, but not executable.
  - Code is executable, but not writeable.

DEP Limitations

- Assumptions:
  - If we prevent attackers from injecting code, we deny them ability to execute arbitrary code.
  - All pages are either writeable or executable, but not both.

- We won!

  … right?

DEP Limitations

- What if some pages need to be both writeable and executable?
  - Examples?
  - Special handling needed for JIT code, memory overlays, and self-modifying code
- Is there another way for an attacker to execute arbitrary code even without the ability to inject it into the victim process?
  - Next time…
  - Teaser: Yes, yes there is.

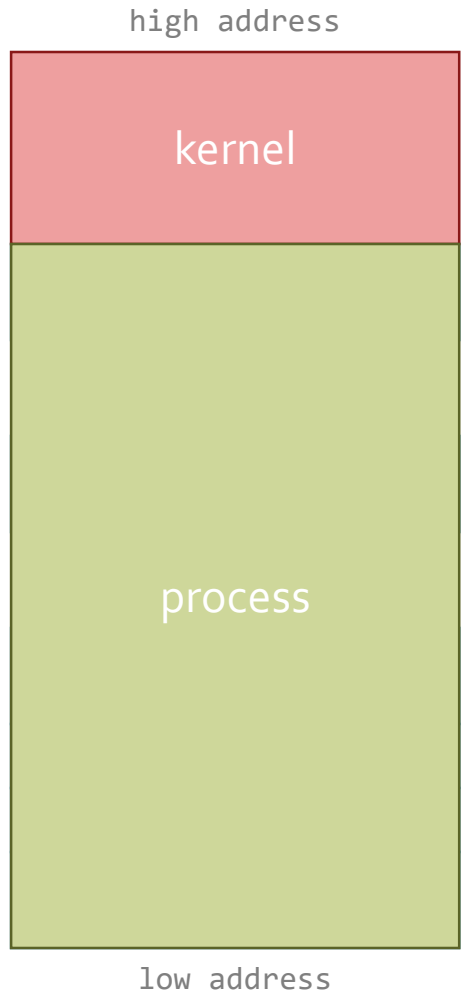
DEP Limitations

- What if the same physical page frame is mapped to two different virtual pages (with different access permissions)?
- An application can modify access permissions to it's own virtual memory pages. Can an attacker cause the victim application to unprotect its stack?
  - May be… next time

DEP Limitations

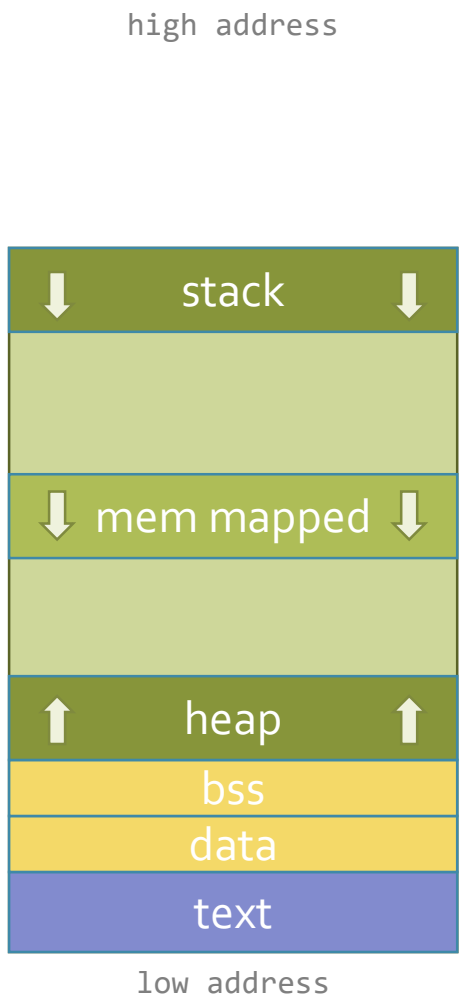
- Requires support from hardware (MMU) and operating system (page tables, loader)
  - Supported on most major platforms
    - May not extend into very low-tier devices
  - Compiler support makes it easy to enable by default for new applications
    - gcc marks code as read-only, stack and heap as non-executable by default
- Side Effects
  - Additional memory usage due to fragmentation

Process Memory Map



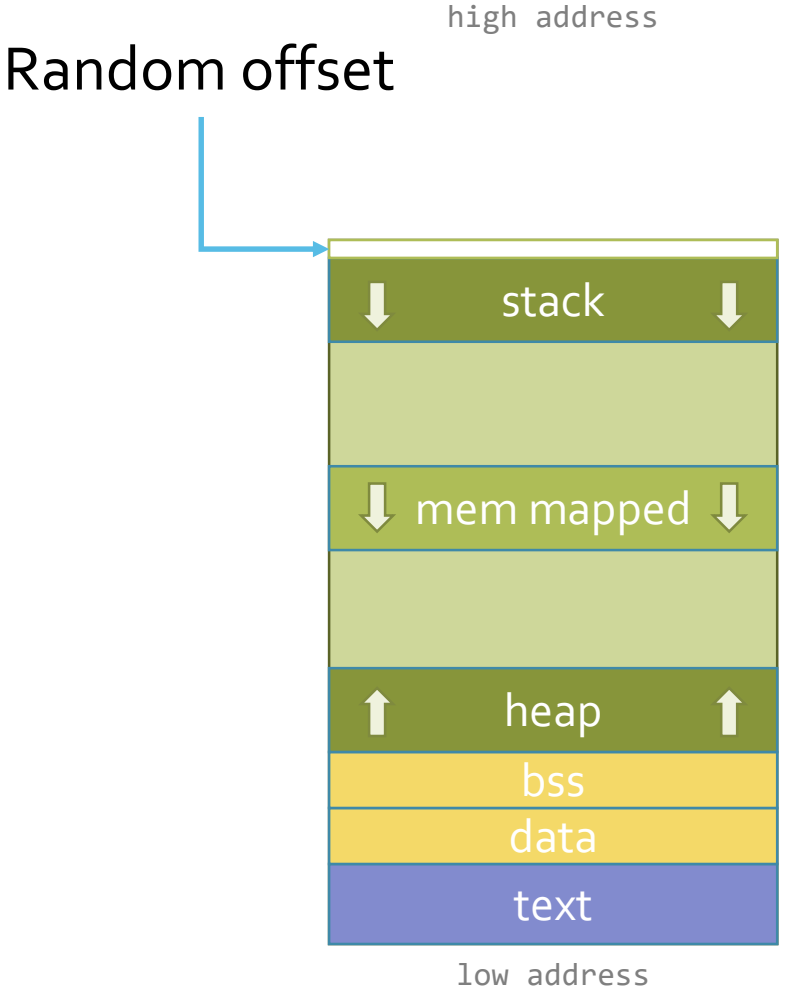
Process Memory Map

- Stack smashing exploits depend on being able to predict stack addresses.



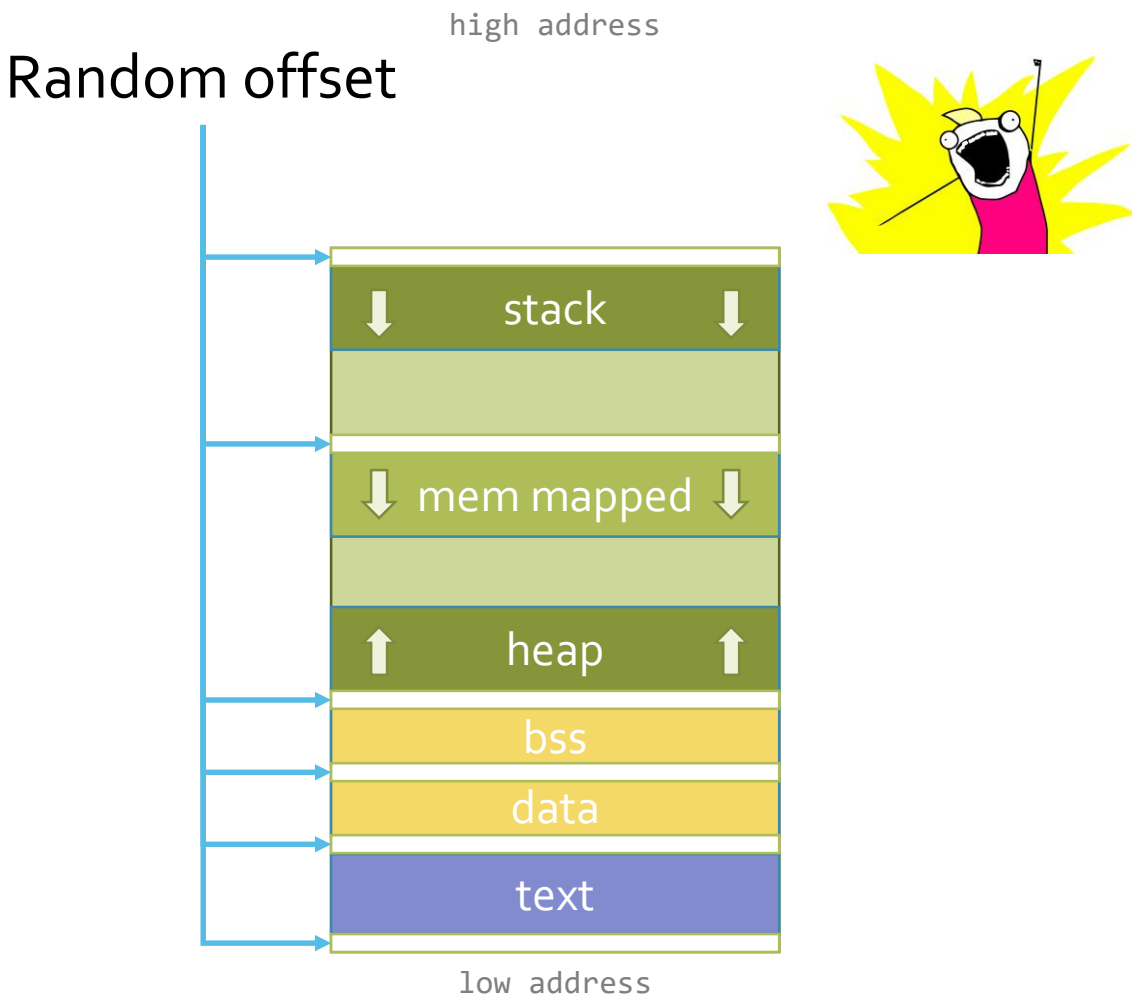
Stack Gap

- Add a random offset to stack base.
- Makes it harder for attackers to correctly predict necessary stack addresses.



Address Space Layout Randomization

- ASLR extends the concept to other sections of process memory.
  - More next time



https://hyperboleandahalf.blogspot.com/2010/06/this-is-why-ill-never-be-adult.html

Stack Gap Limitations

- Assumption: harder for attackers to guess the location of their shellcode on the stack.

- Guessing the offset
  - How random is it?
  - Is the possible range so small that it can it be brute forced?
  - Information leaks.
- Longer NOP sleds

Review

- Countermeasures can make reliable exploitation harder.
  - Will not stop all exploits.
  - Can make exploit development more difficult and costly.
  - Mitigations are important, but they should not be relied upon.
    - Can be bypassed.
- Ongoing arms race between defenders and attackers
  - Co-evolution of defenses and exploitation techniques

Homework

- (for 4/19) Read *Understanding glibc malloc* by sploitfun
  - https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
  - (skip to the part about chunks)

- Continue working on Project 2

Next Lecture…

Low Level Software Security III: Integers, ROP, & CFI