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 offsets relative to the frame pointer
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;
}
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?

![Diagram of stack and frames]

- Stack
- Caller frame
  - `fp`
  - `sp`
- Callee frame
  - Low address
  - High address
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

- 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
How is this implemented?

▪ Caller
  – Pushes arguments onto stack
  – Pushes pointer to format string onto stack

▪ Callee
  – Reads format string off stack
  – Uses format string to read arguments off of stack
    ▪ Reads one value off stack for each “%” parameter
    ▪ To be clear: printf runtime is looking up the stack, controlled by % parameters
Printf on the stack

```c
f() {
    printf("%d\n", 10);
}
```
Key problem

- User is responsible for enforcing one-to-one mapping between format specifiers and arguments

- What if there are too many arguments?

- What if there are too few arguments?
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 a well-crafted format string?
  - Read arbitrary memory
  - Write arbitrary memory
Format String Vulnerabilities: Reading

- printf(“%08x.%08x.%08x.%08x\n”);
  - What will this do?

- printf(“%s\n”);
  - What will this do?

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

- Reading from the stack
- Reading via a pointer
Format String Vulnerabilities: Reading

What if we want to read arbitrary memory?

- **Recall:** the format string itself is on the stack

Consider:

```c
f() {
    char localstring[80] = "\x10\x01\x48\x08_%08x.%08x.%08x.%08x.\|%s\|");
    printf (localstring);
}
```

- Uses "\%08x" specifiers to move printf's argument pointer up the stack... back into the format string itself!
- Will print whatever string is stored at address at 0x08480110
  - Note: "\x10\x01\x48\x08" = 0x08480110
  - Note: this assumes no intervening locals and four words of control data on stack before string pointer.
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?
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`

- Combine with the reading track to write to arbitrary addresses...
  - Can build up a full word write one byte at a time

- While it has uses, "\%n" is considered extremely dangerous and is frequently removed from libraries
Additional Resources

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

- 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.

```c
void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```
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
- 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

```c
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                  // 8 bytes
        .string "/bin/sh"           // 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

```c
char shellcode[] =
  "\x26\x5e\x89\x76\x08\x46\x07\x60\n  \x00\xb8\x0b\x00\x89\xf3\x8d\x4e\n  \xb8\x01\x00\x00\xb0\x00\x89\xf3\x8d\x4e\n  \xff\x2f\x62\x69\x2f\x73\x68\x00\x89\n
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
Mitigations
Countermeasures and Mitigations

- As 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 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?

- Approaches we’ll look at today
  - Try to detect overwrite of control data
    - Stack canaries/cookies
  - Try to make it difficult to redirect control flow to attacker code
    - Memory protection on stack (DEP)
    - Address Space Layout Randomization (ALSR)
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?  
  - 0x000A0DFF

- 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
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. isAuthenticated, 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 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)

- 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.
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.

![Diagram showing Main Stack and Shadow Stack with variables and stack pointers](Diagram.png)
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?

▪ Requires compiler and hardware support to be efficient.

▪ Not widely deployed.
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

- Requires hardware support (MPU, MMU, or SMMU)

- Separate code and data

- New allocations must also conform

- Side Effects
  - Additional memory usage due to fragmentation
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?
  – Special handling needed for JIT code, memory overlays, and self-modifying code
  – Is this common?

▪ 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.
Another idea...

- If the attacker doesn’t know where in memory their shellcode is stored, then it's hard to make the processor jump there...
Process Memory Map

- **Kernel**
- **Process**
Process Memory Map

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

- Add a random offset to stack base.
- Assumption: harder for attackers to guess the location of their shellcode on the stack.
- Bypass?
  - Information leak
  - Longer NOP sled
  - Putt shellcode on heap

![Diagram showing memory regions: stack, mem mapped, heap, bss, data, and text.]

Random offset
Address Space Layout Randomization

- ASLR extends the concept to other sections of process memory.
ASLR Limitations

- Requires compiler, linker, and loader support.

- Side Effects
  - Increases code size and performance overhead
  - Random number generator dependency
  - Boot time impact for relocation

- Bypass
  - Information leaks
    - Servers that fork(), limitations in layout options
  - Spraying
Advanced techniques: 
Heap spray

- Overflow can be used to cause control transfer into heap, but don’t know where shellcode is stored due to ASLR

- Basic idea: overwhelming force
  - Allocate jizillions of copies of the shellcode (with big NOP sleds) and then jump blindly into the heap

- Very common with today’s “drive-by download” attacks on vulnerability
  - Heap spray implemented using Javascript
Drive-By Heap Spraying

Program Heap

ASLR prevents the attack

bad

PC

Creates the malicious object

Triggers the jump

ok

ok

---

\[
\text{shellcode} = \text{unescape('%u4343%u4343%...');}\]

\[
\text{<html>}
\text{<script language="text/javascript">}
\text{shellcode = unescape('%u4343%u4343%...');}\]
\text{</script>}
\text{<iframe src="file://BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB..." name="CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC..." <!--[if gte mpr 9]><!--> &lt;3341;&amp;gt; &amp;lt;3341;&amp;gt; &lt;/iframe&gt;\]
\text{</html>}
\]
Drive-By Heap Spraying

Program Heap

Allocate 1000s of malicious objects

<SCRIPT language="text/javascript">
shellcode = unescape('%u4343%u4343');
oneblock = unescape('%u0C0C%u0C0C');
var fullblock = oneblock;
while (fullblock.length<0x40000) {
  fullblock += fullblock;
}
sprayContainer = new Array();
for (i=0; i<1000; i++) {
sprayContainer[i] = fullblock + shellcode;
}
</SCRIPT>
Review

- Run-time interpreters (e.g., printf) are risky

- Mitigations
  - Do not provide perfect security (all can be bypassed)
    - Makes reliable exploit more expensive to build
  - Stack canaries, DEP, ASLR
Next time

- Integer overflow, return-oriented program and control-flow integrity