6.858 Lecture 2
REVIEW OF BUFFER OVERFLOW ATTACKS

Last lecture, we looked at the basics of performing a buffer overflow attack. That attack leveraged several observations:

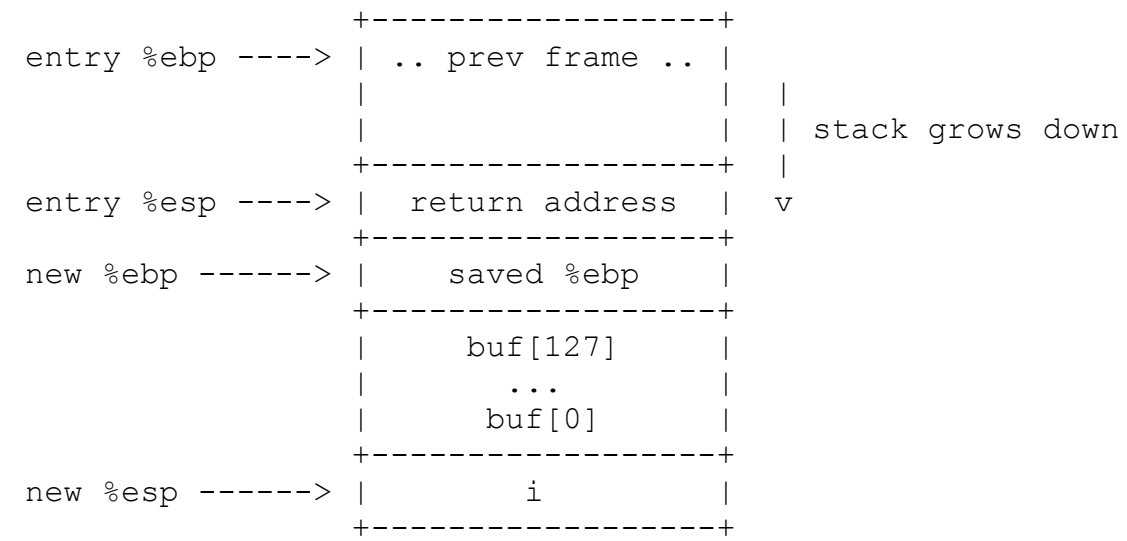
- Systems software is often written in C (operating systems, file systems, databases, compilers, network servers, command shells and console utilities)
- C is essentially high-level assembly, so . . .
  - Exposes raw pointers to memory
  - Does not perform bounds-checking on arrays (b/c the hardware doesn't do this, and C wants to get you as close to the hardware as possible)
- Attack also leveraged architectural knowledge about how x86 code works:
  - The direction that the stack grows
  - Layout of stack variables (esp. arrays and return addresses for functions)

```
void read_req() {
    char buf[128];
    int i;
    gets(buf);
    //. . . do stuff w/buf . . .
}
```

What does the compiler generate in terms of memory layout?

x86 stack looks like this:

- %esp points to the last (bottom-most) valid thing on the stack.
- %ebp points to the caller's %esp value.



How does the adversary take advantage of this code?

- Supply long input, overwrite data on stack past buffer.
- Key observation 1: attacker can overwrite the return address, make the program jump to a place of the attacker's choosing!
- Key observation 2: attacker can set return address to the buffer itself, include some x86 code in there!

What can the attackers do once they are executing code?

- Use any privileges of the process! If the process is running as root or Administrator, it can do whatever it wants on the system. Even if the process is not running as root, it can send spam, read files, and interestingly, attack or subvert other machines behind the firewall.
- Hmmm, but why didn't the OS notice that the buffer has been overrun?
    - As far as the OS is aware, nothing strange has happened! Remember that, to a first approximation, the OS only gets invoked by the web server when the server does IO or IPC. Other than that, the OS basically sits back and lets the program execute, relying on hardware page tables to prevent processes from tampering with each other's memory. However, page table protections don't prevent buffer overruns launched by a process "against itself," since the overflowed buffer and the return address and all of that stuff are inside the process's valid address space.
    - Later in this lecture, we'll talk about things that the OS *can* do to make buffer overflows more difficult.

FIXING BUFFER OVERFLOWS

Approach #1: Avoid bugs in C code.

Programmer should carefully check sizes of buffers, strings, arrays, etc. In particular, the programmer should use standard library functions that take buffer sizes into account (`strncpy()` instead of `strcpy()`, `fgets()` instead of `gets()`, etc.).

Modern versions of gcc and Visual Studio warn you when a program uses unsafe functions like gets(). In general, YOU SHOULD NOT IGNORE COMPILER WARNINGS. Treat warnings like errors!

Good: Avoid problems in the first place!

Bad: It's hard to ensure that code is bug-free, particularly if the code base is large. Also, the application itself may define buffer manipulation functions which do not use fgets() or strcpy() as primitives.

Approach #2: Build tools to help programmers find bugs.

For example, we can use static analysis to find problems in source code before it's compiled. Imagine that you had a function like this:

```
void foo(int *p){
     int offset;
     int *z = p + offset;
     if(offset > 7){
          bar(offset);
     }
}
```

By statically analyzing the control flow, we can tell that offset is used without being initialized. The if-statement also puts bounds on offset that we may be able to propogate to bar. We'll talk about static analysis more in later lectures.

“Fuzzers” that supply random inputs can be effective for finding bugs. Note that fuzzing can be combined with static analysis to maximize code coverage!

Bad: Difficult to prove the complete absence of bugs, esp. for unsafe code like C.

Good: Even partial analysis is useful, since programs should become strictly less buggy. For example, baggy bounds checking cannot catch all memory errors, but it can detect many important kinds.

Approach #3: Use a memory-safe language (JavaScript, C#, Python).

Good: Prevents memory corruption errors by not exposing raw memory addresses to the programmer, and by automatically handling garbage collection.

Bad: Low-level runtime code DOES use raw memory addresses. So, that runtime core still needs to be correct. For example, heap spray attacks:

- https://www.usenix.org/legacy/event/sec09/tech/full_papers/ratanaworabhan.pdf
- https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

Bad: Still have a lot of legacy code in unsafe languages (FORTRAN and COBOL oh noes).

Bad: Maybe you need access to low-level hardware features b/c, e.g., you're writing a device driver.

Bad: Perf is worse than a fine-tuned C application?

- Used to be a bigger problem, but hardware and high-level languages are getting better.
    - JIT compilation FTW!
    - asm.js is within 2x of native C++ perf! [http://asmjs.org/faq.html]
- Use careful coding to avoid garbage collection jitter in critical path.
- Maybe you're a bad person/language chauvinist who doesn't know how to pick the right tool for the job. If your task is I/O-bound, raw compute speed is much less important. Also, don't be the chump who writes text manipulation programs in C.

All 3 above approaches are effective and widely used, but buffer overflows are still a problem in practice.

- Large/complicated legacy code written in C is very prevalent.
- Even newly written code in C/C++ can have memory errors.

How can we mitigate buffer overflows despite buggy code?

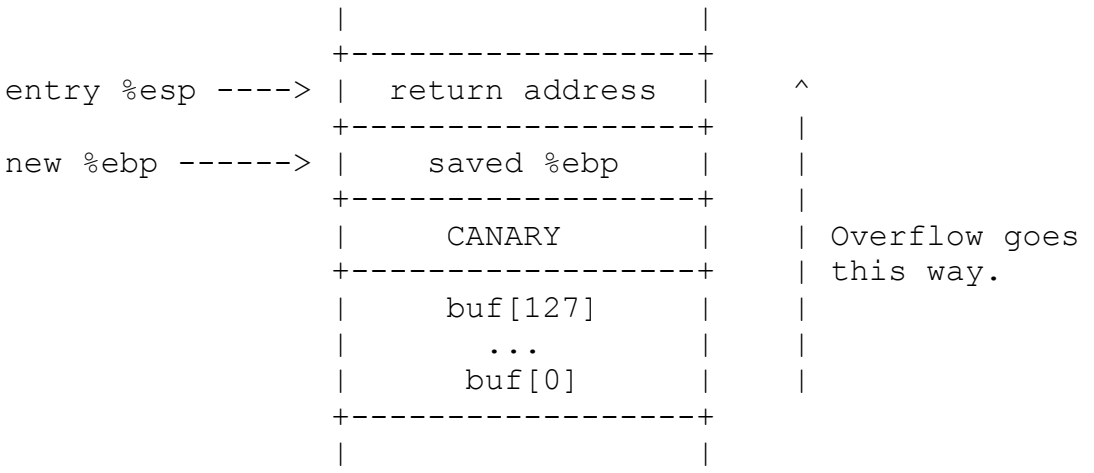
- Two things going on in a "traditional" buffer overflow:
    - Adversary gains control over execution (program counter).
    - Adversary executes some malicious code.
- What are the difficulties to these two steps?
    - Requires overwriting a code pointer (which is later invoked). Common target is a return address using a buffer on the stack. Any memory error could potentially work, in practice. Function pointers, C++ vtables, exception handlers, etc.
    - Requires some interesting code in process's memory. This is often easier than #1, because:
        - it's easy to put code in a buffer, and
        - the process already contains a lot of code that might be exploitable.
    - However, the attacker needs to put this code in a predictable location, so that the attacker can set the code pointer to point to the evil code!

Mitigation approach 1: canaries (e.g., StackGuard, gcc's SSP)

Idea: OK to overwrite code ptr, as long as we catch it before invocation.

One of the earlier systems: StackGuard

- Place a canary on the stack upon entry, check canary value before return.
- Usually requires source code; compiler inserts canary checks.
- Q: Where is the canary on the stack diagram?
    - A: Canary must go "in front of" return address on the stack, so that any overflow which rewrites return address will also rewrite canary.



Q: Suppose that the compiler always made the canary 4 bytes of the 'a' character. What's wrong with this?

- A: Adversary can include the appropriate canary value in the buffer overflow!

So, the canary must be either hard to guess, or it can be easy to guess but still resilient against buffer overflows. Here are examples of these approaches.

- "Terminator canary": four bytes (0, CR, LF, -1)
  - Idea: Many C functions treat these characters as terminators(e.g., gets(), sprintf()). As a result, if the canary matches one of these terminators, then further writes won't happen.
- Random canary generated at program init time: Much more common today (but, you need good randomness!).

What kinds of vulnerabilities will a stack canary not catch?

- Overwrites of function pointer variables before the canary.
- Attacker can overwrite a data pointer, then leverage it to do arbitrary mem writes.

```
int *ptr = ...;
char buf[128];
gets(buf);  //Buffer is overflowed, and overwrites ptr.
*ptr = 5;   //Writes to an attacker-controlled address!
            //Canaries can't stop this kind of thing.
```

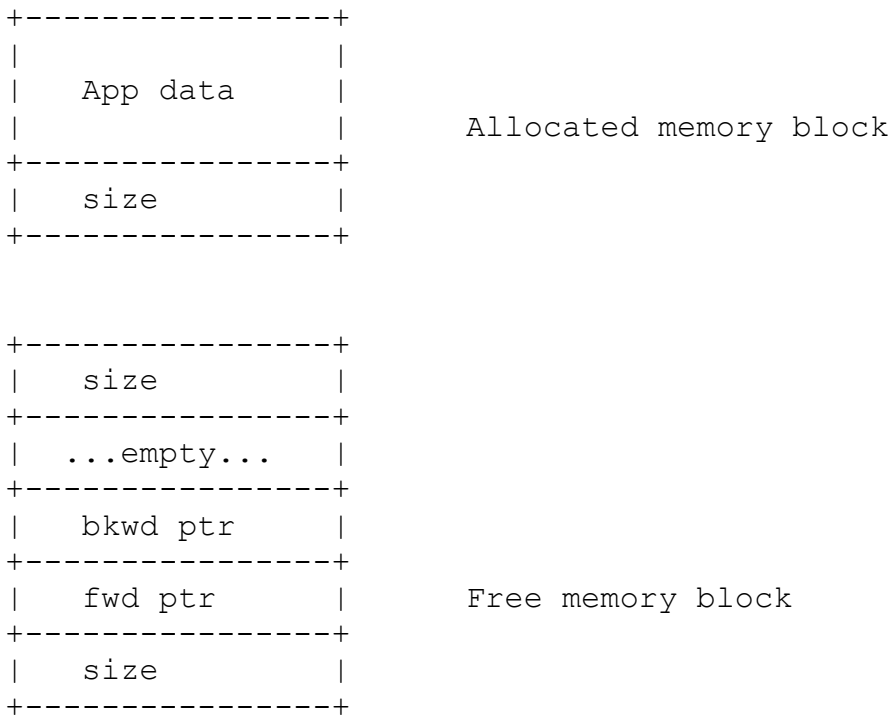
- Heap object overflows (function pointers, C++ vtables).
- malloc/free overflows

```
int main(int argc, char **argv) {
     char *p, *q;

     p = malloc(1024);
     q = malloc(1024);
```

```
      if(argc >= 2)
           strcpy(p, argv[1]);
      free(q);
      free(p);
      return 0;
}
```

Assume that the two blocks of memory belonging to p and q are adjacent/nearby in memory.
Assume that malloc and free represent memory blocks like this:



So, the buffer overrun in p will overwrite the size value in q's memory block! Why is this a problem?

When free() merges two adjacent free blocks, it needs to manipulate bkwd and fwd pointers, and the pointer calculation uses size to determine where the free memory block structure lives!

```
p = get_free_block_struct(size);
bck = p->bk;
fwd = p->fd;
fwd->bk = bck; //Writes memory!
bck->fd = fwd;   //Writes memory!
```

The free memory block is represented as a C struct; by corrupting the size value, the attacker can force free() to operate on a fake struct that resides in attacker-

controlled memory and has attacker-controlled values for the forward and backwards pointers.

If the attacker knows how free() updates the pointers, he can use that update code to write an arbitrary value to an arbitrary place. For example, the attacker can overwrite a return address.

Actual details are a bit more complicated; if you're interested in gory details, go here: http://www.win.tue.nl/~aeb/linux/hh/hh-11.html

The high-level point is that stack canaries won't prevent this attack, because the attacker is "skipping over" the canary and writing directly to the return address!

So, stack canaries are one approach for mitigating buffer overflows in buggy code.

Mitigation approach 2: bounds checking.

Overall goal: prevent pointer misuse by checking if pointers are in range.

Challenge: In C, it can be hard to differentiate between a valid pointer and an invalid pointer. For example, suppose that a program allocates an array of characters …

```
char x[1024];
```

… as well as a pointer to some place in that array, e.g.,

```
char *y = &x[107];
```

Is it OK to increment y to access subsequent elements?

- If x represents a string buffer, maybe yes.
- If x represents a network message, maybe no.

Life is even more complicated if the program uses unions.

```
union u{
    int i;
    struct s{
        int j;
        int k;
    };
};
int *ptr = &(u.s.k); //Does this point to valid data?
```

The problem is that, in C, a pointer does not encode information about the intended usage semantics for that pointer. So, a lot of tools don't try to guess those semantics. Instead, the tools have a less lofty goal than "totally correct" pointer semantics: the

tools just enforce the memory bounds on heap objects and stack objects. At a high level, here's the goal: For a pointer p' that's derived from p, p' should only be dereferenced to access the valid memory region that belongs to p.

Enforcing memory bounds is a weaker goal than enforcing "totally correct" pointer semantics. Programs can still shoot themselves in the foot by trampling on their memory in nasty ways (e.g., in the union example, the application may write to ptr even though it's not defined).

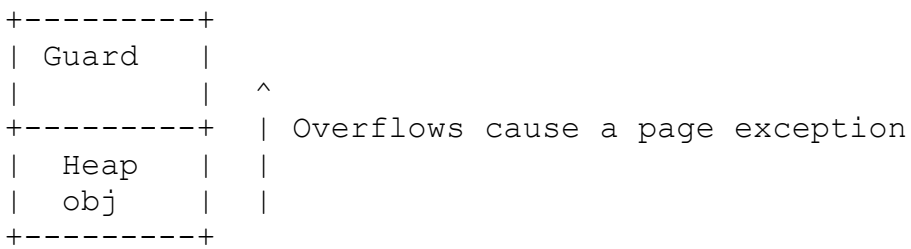
However, bounds checking is still useful because it prevents *arbitrary* memory overwrites. The program can only trample its memory if that memory is actually allocated! THIS IS CONSIDERED PROGRESS IN THE WORLD OF C.

A drawback of bounds checking is that it typically requires changes to the compiler, and programs must be recompiled with the new compiler. This is a problem if you only have access to binaries.

What are some approaches for implementing bounds checking?

Bounds checking approach #1: Electric fences

- This is an old approach that had the virtue of being simple.
- Idea: Align each heap object with a guard page, and use page tables to ensure that accesses to the guard page cause a fault.

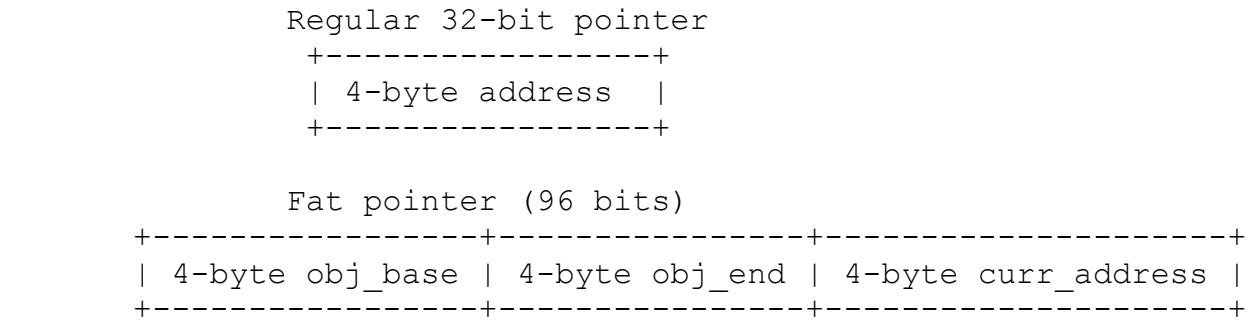


- This is a convenient debugging technique, since a heap overflow will immediately cause a crash, as opposed to silently corrupting the heap and causing a failure at some indeterminate time in the future.
- Big advantage: Works without source code---don't need to change compilers or recompile programs! [You *do* need to relink them so that they use a new version of malloc which implements electric fences.]
- Big disadvantage: Huge overhead! There's only one object per page, and you have the overhead of a dummy page which isn't used for "real" data.
- Summary: Electric fences can be useful as debugging technique, and they can prevent some buffer overflows for heap objects. However, electric fences can't protect the stack, and the memory overhead is too high to use in production systems.

Bounds checking approach #2: Fat pointer

Idea: Modify the pointer representation to include bounds information. Now, a pointer includes a memory address and bounds information about an object that lives in that memory region.

Ex:



You need to modify the compiler and recompile the programs to use the fat pointers. The compiler generates code to abort the program if it dereferences a pointer whose address is outside of its own base...end range.

```
int *ptr = malloc(sizeof(int) * 2);
while(1){
     *ptr = 42;    <----------|
     ptr++;                   |
}                             |
    _____________________________|
   |
```

This line checks the current address of the pointer and ensures that it's in-bounds. Thus, this line will fail during the third iteration of the loop.

Problem #1: It can be expensive to check all pointer dereferences. The C community hates things that are expensive, because C is all about SPEED SPEED SPEED.

Problem #2: Fat pointers are incompatible with a lot of existing software.

- You can't pass a fat pointer to an unmodified library.
- You can't use fat pointers in fixed-size data structures. For example, `sizeof(that_struct)` will change!
- Updates to fat pointers are not atomic, because they span multiple words. Some programs assume that pointer writes are atomic.

Bounds checking approach #3: Use shadow data structures to keep track of bounds information (Jones and Kelly, Baggy).

Basic idea: For each allocated object, store how big the object is. For example: Record the value passed to malloc:

```
char *p = malloc(mem_size);
```

For static variables, the values are determined by the compiler:

```
char p[256];
```

For each pointer, we need to interpose on two operations:

- pointer arithmetic: `char *q = p + 256;`
- pointer dereferencing: `char ch = *q;`

Q: Why do we need to interpose on dereference? Can't we do just arithmetic?

- A: An invalid pointer isn't always a bug! For example, a pointer to one element past the last item of an array might be used as a stopping test in a loop. Applications can also do goofy stuff like:
    - Simulating 1-indexed arrays
    - Computing p+(a-b) as (p+a)-b
    - Generating OOB pointers that are later checked for validity

So, the mere creation of invalid pointer shouldn't cause program to fail.

Q: Why do we need to interpose on arithmetic? Can't we do just dereference?

- A: Interposing on arithmetic is what allows us to track the provenance of pointers and set the OOB bit. Without the OOB, we won't be able to tell when a derived pointer goes outside of the bounds of its base object.

Challenge 1: How do we find the bounds information for a regular pointer, i.e., a pointer that's in-bounds?

Naive: Use a hash table or interval tree to map addresses to bounds.
Good: Space efficient (only store info for in-use pointers, not all possible addresses).
Bad: Slow lookup (multiple memory accesses per look-up).

Naive: Use an array to store bounds info for *every* memory address.
Good: Fast!
Bad: Really high memory overhead.

Challenge 2: How do we force out-of-bounds pointer dereferences to fail?

Naive: Instrument every pointer dereference.
Good: Uh, it works.
Bad: Expensive---we have to execute extra code for every dereference!

The baggy bounds approach: 5 tricks

- Round up each allocation to a power of 2, and align the start of the allocation to that power of 2.
- Express each range limit as log_2(alloc_size). For 32-bit pointers, only need 5 bits to express the possible ranges.
- Store limit info in a linear array: fast lookup with one byte per entry. Also, we can use virtual memory to allocate the array on-demand!
- Allocate memory at slot granularity (e.g., 16 bytes): fewer array entries.

Ex:

```
slot_size = 16
p = malloc(16);              table[p/slot_size] = 4;

p = malloc(32);              table[p/slot_size] = 5;
                             table[(p/slot_size) + 1] = 5;
```

Now, given a known good pointer p, and a derived pointer p', we can test whether p' is valid by checking whether both pointers have the same prefix in their address bits, and they only differ in their e least significant bits, where e is equal to the logarithm of the allocation size.

C code

```
------
p' = p + i;
```

Bounds check

```
------------
size = 1 << table[p >> log_of_slot_size];
base = p & ~(size - 1);
 (p' >= base) && ((p' - base) < size)
```

Optimized bounds check

```
----------------------
 (p^p') >> table[p >> log_of_slot_size] == 0
```

- Use virtual memory system to prevent out-of-bound derefs: set most significant bit in an OOB pointer, and then mark pages in the upper half of the address space as inaccessible. So, we don't have to instrument pointer dereferences to prevent bad memory accesses!

Example code (assume that slot_size=16):

```
char *p = malloc(44);
//Note that the nearest power of 2 (i.e.,
//64 bytes) are allocated. So, there are
//64/(slot_size) = 4 bounds table entries
//that are set to log_2(64) = 6.

char *q = p + 60;
//This access is ok: It's past p's object
//size of 44, but still within the baggy
//bounds of 64.

char *r = q + 16;
//r is now at an offset of 60+16=76 from
```

```
//p. This means that r is (76-64)=12 bytes
//beyond the end of p. This is more than
//half a slot away, so baggy bounds will
//raise an error.

char *s = q + 8;
//s is now at an offset of 60+8=68 from p.
//So, s is only 4 bytes beyond the baggy
//bounds, which is les than half a slot
//away. No error is raised, but the OOB
//high-order bit is set in s, so that s
//cannot be dereferenced.

char *t = s - 32;
//t is now back inside the bounds, so
//the OOB bit is cleared.
```

For OOB pointers, the high bit is set (if OOB within half a slot).

- Typically, OS kernel lives in upper half, protects itself via paging hardware.
- Q: Why half a slot for out-of-bounds?

So what's the answer to the homework problem?

```
char *p = malloc(256);
char *q = p + 256;
char ch = *q;  //Does this raise an exception?
               //Hint: How big is the baggy bound for p?
```

ADDITIONAL/SUPPLEMENTAL INFO

==========

Some bugs in the baggy bounds paper:

Figure 3, explicit bounds check should generate the size like this:

```
size = 1 << table[p >> log_of_slot_size]
```

Figure 3, optimized bounds check should be

```
(p^p') >> table[p >> log_of_slot_size] == 0
```

Figures 5 and 18, pointer arithmetic code should be

```
char *p = &buf[i];
```

or

```
char *p = buf + i;
```

6.858 Computer Systems Security

Fall 2014