6.858 Lecture 3

Baggy bounds continued:
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;
//ERROR: r is now at an offset of 60+16=76
//from p. This means that r is (76-64)=12
//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 less than half a slot
//away. No error is raised, but the OOB
//high-order bit is set in s, so that s
//cannot be derefernced.
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?
```

Does baggy bounds checking have to instrument *every* memory address computation and access? No: static analysis can prove that some addresses are always safe to use. However, some address calculations are "unsafe" in the sense

that there's no way to statically determine bounds on their values. Such unsafe variables need checks.

Handling function call arguments is a bit tricky, because the x86 calling convention is fixed, i.e., the hardware expects certain things to be in certain places on the stack.

However, we can copy unsafe arguments to a separate area, and make sure that the copied arguments are aligned and protected.

Q: Do we have to overwrite the original arguments with the copies values upon function return?

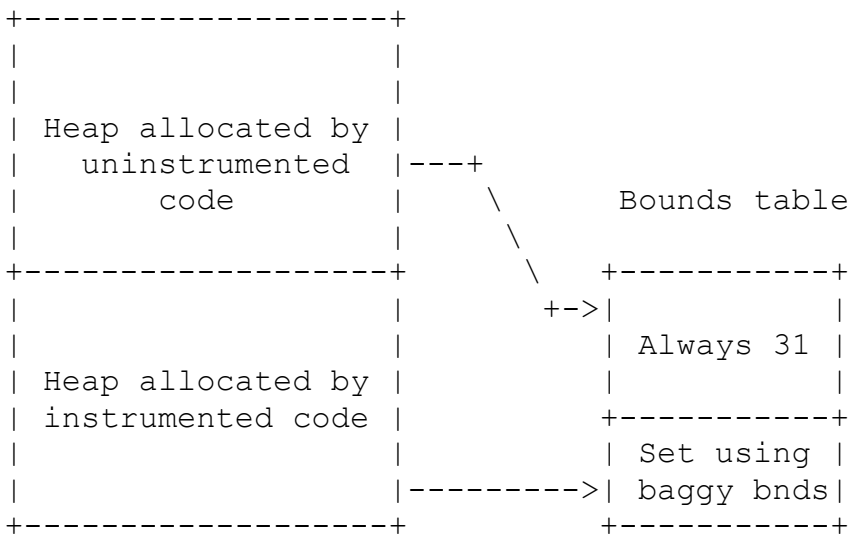
- A: No, because everything is pass-by-value in C!

How does baggy bounds checking ensure binary compatibility with existing libraries? In particular, how does baggy bounds code interact with pointers to memory that was allocated by uninstrumented code?

Solution: Each entry in the bounds table is initialized to the value 31, meaning that the corresponding pointer has a memory bound of 2^31 (which is all of the addressable memory). On memory allocation in *instrumented* code, bounds entries are set as previously discussed, and reset to 31 when the memory is deallocated. Memory allocated to uninstrumented code will never change bounds table entries from their default values of 31; so, when instrumented code interacts with those pointers, bound errors will never happen.

Example:

Contiguous range of memory used for the heap



What does this all mean?

- Can't detect out-of-bounds pointers generated in uninstrumented code.
- Can't detect when OOB pointer passed into library goes in-bounds again.
    - Q: Why?
    - A: Because there is no pointer inspection in the uninstrumented code which could clear the high-order OOB bit!
    - Q: Why do they instrument strcpy() and memcpy()?
    - A: Because otherwise, those functions are uninstrumented code, and suffer from the same problems that we just discussed. For example off-the-shelf strcpy() does not ensure that dest has enough space to store src!

How can baggy bits leverage 64-bit address spaces?

- Can get rid of the table storing bounds information, and put it in the pointer.

```
Regular pointer
  +---------------+-------+-----------------------+
  |      zero     |  size |   supported addr space |
  +---------------+-------+-----------------------+
          21          5             38

OOB pointer
  +--------+------+-------+-----------------------+
  | offset | size |  zero |   supported addr space |
  +--------+------+-------+-----------------------+
      13      5       8             38
```

This is similar to a fat pointer, but has the advantages that:

1) tagged pointers are the same size as regular pointers
2) writes to them are atomic

so programmer expectations are not broken, and data layouts stay the same.
Also note that, using tagged pointers, we can now keep track of OOB pointers that go much further out-of-bounds. This is because now we can tag pointers with an offset indicating how far they are from their base pointer. In the 32-bit world, we couldn't track OOB offsets without having an additional data structure!

Can you still launch a buffer overflow attack in a baggy bounds system? Yes, because the world is filled with sadness.

- Could exploit a vulnerability in uninstrumented libraries.
- Could exploit temporal vulnerabilities (use-after-free).
- Mixed buffers and code pointers:

```
struct {
    void (*f) (void);
    char buf[256];
} my_type;
```

Note that *f is not an allocated type, so there are no bounds checks associated with its dereference during invocation. Thus, if s.buf is overflowed (e.g., by a bug in an uninstrumented library) and s.f is corrupted, the invocation of f will not cause a bounds error!

Would re-ordering f and buf help?

- Might break applications that depend on struct layout.
- Might not help if this is an array of (struct my_type)'s

In general, what are the costs of bounds checking?

- Space overhead for bounds information (fat pointer or baggy bounds table).
- Baggy bounds also has space overhead for extra padding memory used by buddy allocator (although some amount of overhead is intrinsic to all popular algorithms for dynamic memory allocation).
- CPU overheads for pointer arithmetic, dereferencing.
- False alarms!
    - Unused out-of-bounds pointers.
    - Temporary out-of-bounds pointers by more than slot_size/2.
    - Conversion from pointer to integers and back.
    - Passing out-of-bounds pointer into unchecked code (the high address bit is set, so if the unchecked code does arithmetic using that pointer, insanity may ensue).
- Requires a significant amount of compiler support

So, baggy bounds checking is an approach for mitigating buffer overflows in buggy code.

Mitigation approach 3: non-executable memory (AMD's NX bit, Windows DEP, W^X, ...)

- Modern hardware allows specifying read, write, and execute perms for memory (R, W permissions were there a long time ago; execute is recent.)
- Can mark the stack non-executable, so that adversary cannot run their code.
- More generally, some systems enforce "W^X", meaning all memory is either writable, or executable, but not both. (Of course, it's OK to be neither.)
    - Advantage: Potentially works without any application changes.
    - Advantage: The hardware is watching you all of the time, unlike the OS.
    - Disadvantage: Harder to dynamically generate code (esp. with W^X).
        - JITs like Java runtimes, Javascript engines, generate x86 on the fly.
        - Can work around it, by first writing, then changing to executable.

Mitigation approach 4: randomized memory addresses (ASLR, stack randomization, ...

Observation: Many attacks use hardcoded addresses in shellcode! [The attacker grabs a binary and uses gdb to figure out where stuff lives.]

- So, we can make it difficult for the attacker to guess a valid code pointer.
    - Stack randomization: Move stack to random locations, and/or place padding between stack variables. This makes it more difficult for attackers to determine:
        - Where the return address for the current frame is located
        - Where the attacker's shellcode buffer will be located
    - Randomize entire address space (Address Space Layout Randomization): randomize the stack, the heap, location of DLLs, etc.
        - Rely on the fact that a lot of code is relocatable.
        - Dynamic loader can choose random address for each library, program.
        - Adversary doesn't know address of system(), etc.
    - Can this still be exploited?
        - Adversary might guess randomness. Especially on 32-bit machines, there aren't many random bits (e.g., 1 bit belongs to kernel/user mode divide, 12 bits can't be randomized because memory-mapped pages need to be aligned with page boundaries, etc.).
        - For example, attacker could buffer overflow and try to overwrite the return address with the address of usleep(16), and then seeing if the connection hangs for 16 seconds, or if it crashes (in which case the server forks a new ASLR process with the same ASLR offsets). usleep() could be in one of 2^16 or 2^28 places. [Mor details: https://cseweb.ucsd.edu/~hovav/dist/asrandom.pdf]
    - ASLR is more practical on 64-bit machines (easily 32 bits of randomness).
- -Adversary might extract randomness.
    - Programs might generate a stack trace or error message which contains a pointer.
    - If adversaries can run some code, they might be able to extract real addresses (JIT'd code?).
    - Cute address leak in Flash's Dictionary (hash table):
        1) Get victim to visit your Flash-enabled page (e.g., buy an ad).
        2) Hash table internally computes hash value of keys.
        3) Hash value of integers is the integer.
        4) Hash value of object is its memory address.
        5) Iterating over a hash table is done from lowest hash key to highest hash key.
        6) So, the attacker creates a Dictionary, inserts a string object which has shellcode, and then inserts a bunch of numbers into the Dictionary.

7) By iterating through the Dictionary, the attacker can determine where the string object lives by seeing which integers the object reference falls between!
8) Now, overwrite a code pointer with the shellcode address and bypass ASLR!

- Adversary might not care exactly where to jump.
  - Ex: "Heap spraying": fill memory w/ shellcode so that a random jump is OK!
- Adversary might exploit some code that's not randomized (if such code exists).
- Some other interesting uses of randomization:
  - System call randomization (each process has its own system call numbers).
  - Instruction set randomization so that attacker cannot easily determine what "shellcode" looks like for a particular program instantiation.
  - *Ex: Imagine that the processor had a special register to hold a "decoding key." Each installation of a particular application is associated with a random key. Each machine instruction in the application is XOR'ed with this key. When the OS launches the process, it sets the decoding ke register, and the processor uses this key to decode instructions before executing them.

Which buffer overflow defenses are used in practice?

- gcc and MSVC enable stack canaries by default.
- Linux and Windows include ASLR and NX by default.
- Bounds checking is not as common, due to:
  1) Performance overheads
  2) Need to recompile program
  3) False alarms: Common theme in security tools: false alarms preven adoption of tools! Often, zero false alarms with some misses better than zero misses but false alarms.

RETURN-ORIENTED PROGRAMMING (ROP)

ASLR and DEP are very powerful defensive techniques.

- DEP prevents the attacker from executing stack code of his or her choosing
- ASLR prevents the attacker from determining where shellcode or return addresses are located.
- However, what if the attacker could find PREEXISTING CODE with KNOWN FUNCTIONALITY that was located at a KNOWN LOCATION? Then, the attacker could invoke that code to do evil.
  - Of course, the preexisting code isn't *intentionally* evil, since it is a normal part of the application.
  - However, the attacker can pass that code unexpected arguments, or jum to the middle of the code and only execute a desired piece of that code.

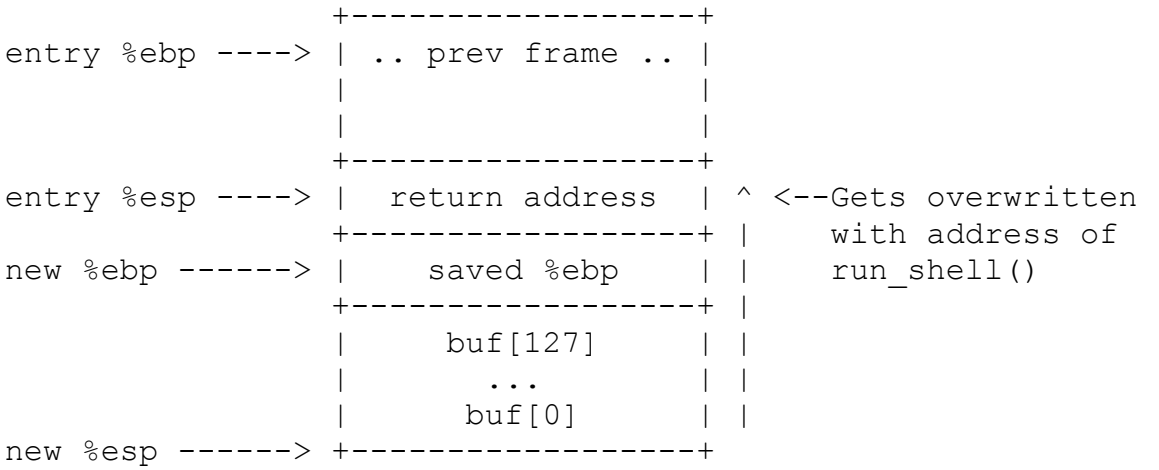
These kinds of attacks are called return-oriented programming, or ROP. To understand how ROP works, let's examine a simple C program that has a securit vulnerability.

```
void run_shell(){
    system("/bin/bash");
}

void process_msg(){
    char buf[128];
    gets(buf);
}
```

Let's imagine that the system does not use ASLR or stack canaries, but it does use DEP. process_msg() has an obvious buffer overflow, but the attacker can't use this overflow to execute shellcode in buf, since DEP makes the stack non-executable. However, that run_shell() function looks tempting . . . how can the attacker execute it?

1) Attacker disassembles the program and figures out where the starting address of run_shell().
2) The attacker launches th buffer overflow, and overwrites the return address of process_msg() with the address of run_shell(). Boom! The attacker now has access to a shell which runs with the privileges of the application.



That's a straightforward extension of the buffer overflows that we've already looked at. But how can we pass arguments to the function that we're jumping to?

```
char *bash_path = "/bin/bash";

void run_cmd(){
```

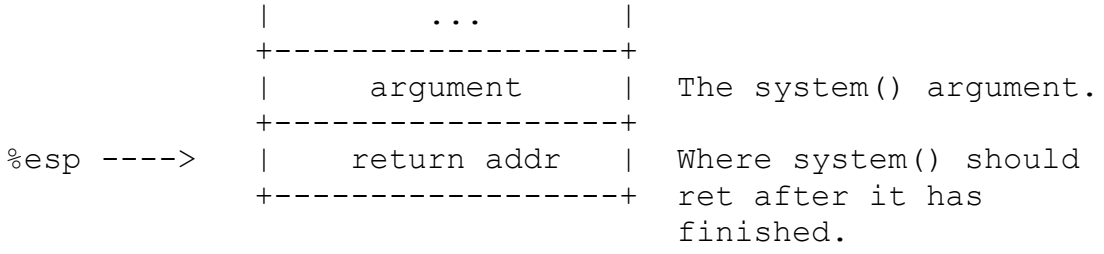
```
        system("/something/boring");
    }

    void process_msg(){
        char buf[128];
        gets(buf);
    }
```

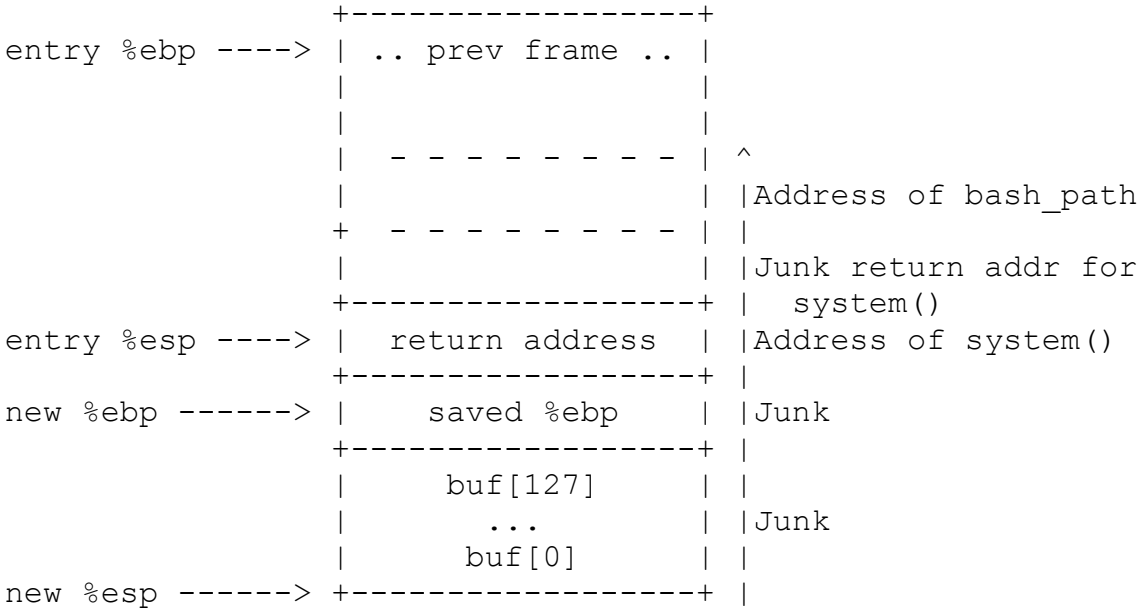
In this case, the argument that we want to pass to is already located in the progra code. There's also a preexisting call to system(), but that call isn't passing the argument that we want.

We know that system() must be getting linked to our program. So, using our trust friend gdb, we can find where the system() function is located, and where bash_path is located.

To call system() with the bash_path argument, we have to set up the stack in the way that system() expects when we jump to it. Right after we jump to system() system() expects this to be on the stack:

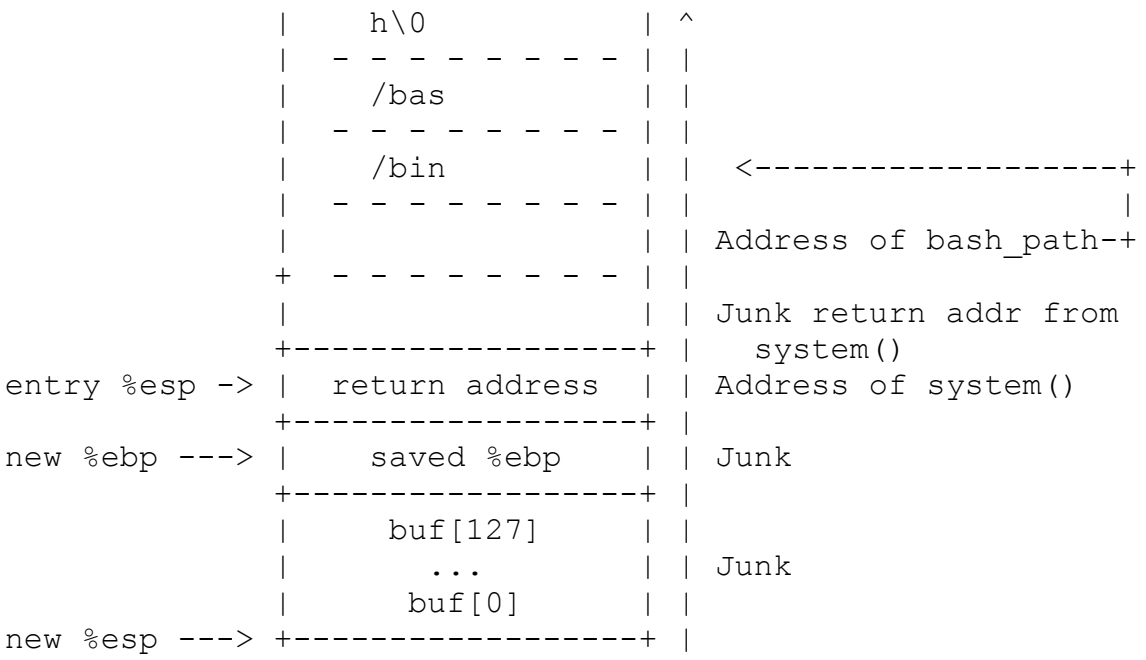


So, the buffer overflow needs to set up a stack that looks like this:



In essence, what we've done is set up a fake calling frame for the system() call! In other words, we've simulated what the compiler would do if it actually wanted to setup a call to system().

What if the string "/bin/bash" was not in the program
We could include that string in the buffer overflow, and then have the argument to system() point to the string.



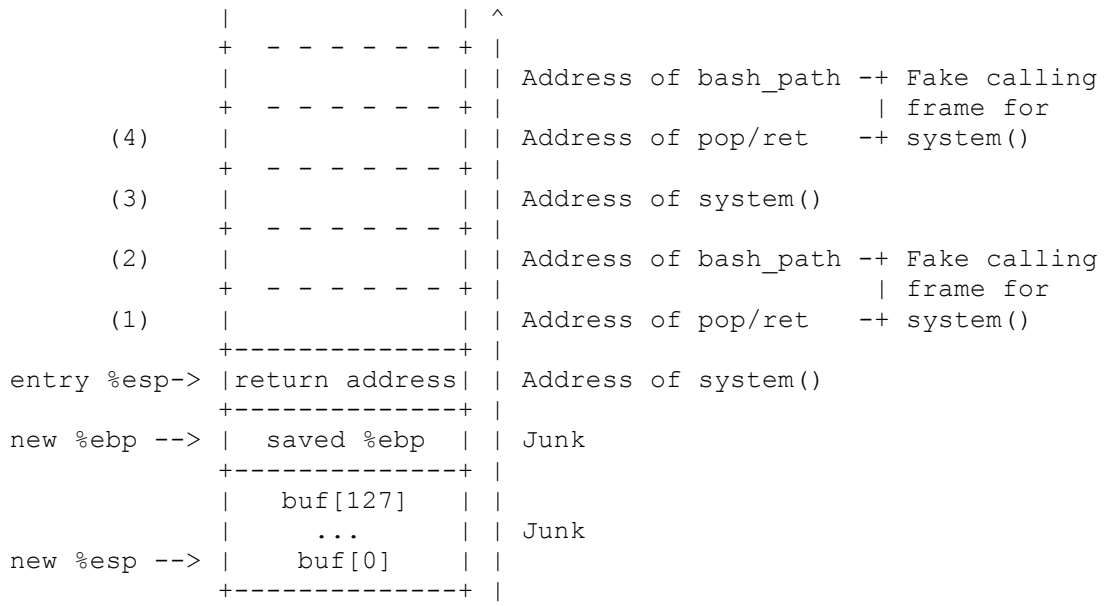
Note that, in these examples, I've been assuming that the attacker used a junk return address from system(). However, the attacker could set it to something useful. In fact, by setting it to something useful, the attacker can chain calls together!

GOAL: We want to call system("/bin/bash") multiple times. Assume that we've found three addresses:

1) The address of system()
2) The address of the string "/bin/bash"
3) The address of these x86 opcodes:

- pop %eax //Pops the top-of-stack and puts it in %eax
- ret //Pops the top-of-stack and puts it in %eip

These opcodes are an example of a "gadget." Gadgets are preexisting instruction sequences that can be strung together to create an exploit. Note that there are user-friendly tools to help you extract gadgets from preexisting binaries (e.g. msfelfscan).



So, how does this work? Remember that the return instruction pops the top of the stack and puts it into %eip.

1) The overflowed function terminates by issuing ret. Ret pops off the top-of-the-stack (the address of system()) and sets %eip to it. system() starts executing, and %esp is now at (1), and points to the pop/ret gadget.
2) system() finishes execution and calls ret. %esp goes from (1)-->(2) as the ret instruction pops the top of the stack and assigns it to %eip. %eip is now the start of the pop/ret gadget.
3) The pop instruction in the pop/ret gadget discards the bash_path variable from the stack. %esp is now at (3). We are still in the pop/ret gadget!
4) The ret instruction in the pop/ret gadget pops the top-of-the-stack and puts it into %eip. Now we're in system() again, and %esp is (4).

And so on and so forth. Basically, we've created a new type of machine that is driven by the stack pointer instead of the regular instruction pointer! As the stack pointe moves down the stack, it executes gadgets whose code comes from preexisting program code, and whose data comes from stack data created by the buffer overflow. This attack evades DEP protections--we're not generating any new code, just invoking preexisting code!

Stack reading: defeating canaries
Assumptions

1) The remote server has a buffer overflow vulnerability.
2) Server crashes and restarts if a canary value is set to an incorrect value.
3) When the server respawns, the canary is NOT re-randomized, and the ASLR is NOT re-randomized, e.g., because the server uses Linux's PIE mechanism, and fork() is used to make new workers and not execve().

So, to determine an 8-byte canary value:

```
char canary[8];
for(int i = 1; i <= 8; i++){  //For each canary byte…
     for(char c = 0; c < 256; c++){  //…guess the value.
     canary[i-1] = c;
     server_crashed = try_i_byte_overflow(i, canary);
          if(!server_crashed){
               //We've discovered i-th byte of the
               //the canary!
               break;
          }
     }
}
//At this point we have the canary, but remember that the
//attack assumes that the server uses the same canary after
//a crash.
```

Guessing the correct value for a byte takes 128 guesses on average, so on a 32-bit system, we only need 4*128=512 guesses to determine the canary (on a 64-bit system, we need 8*128=1024).

- Much faster than brute force attacks on the canary (2^15 or 2^27 expected guesses on 32/64 bit systems with 16/28 bits of ASLR randomness).
- Brute force attacks can use the usleep(16) probe that we discussed earlier.

Canary reading can be extended to reading arbitrary values that the buffer overflow can overwrite!

So, we've discussed how we can defeat randomized canaries if canaries are not changed when a server regenerates. We've also shown how to use gdb and gadgets to execute preexisting functions in the program using arguments that the attacker controls. But what if the server DOES use ASLR? This prevents you from usin offline analysis to find where the preexisting functions are?

This is what the paper for today's lecture discussed. That paper assumed that we're using a 64-bit machine, so that's what we'll assume in this lecture from now on. For the purposes of this discussion, the main change is that function arguments are now passed in registers instead of on the stack.

Blind Return-oriented Programming

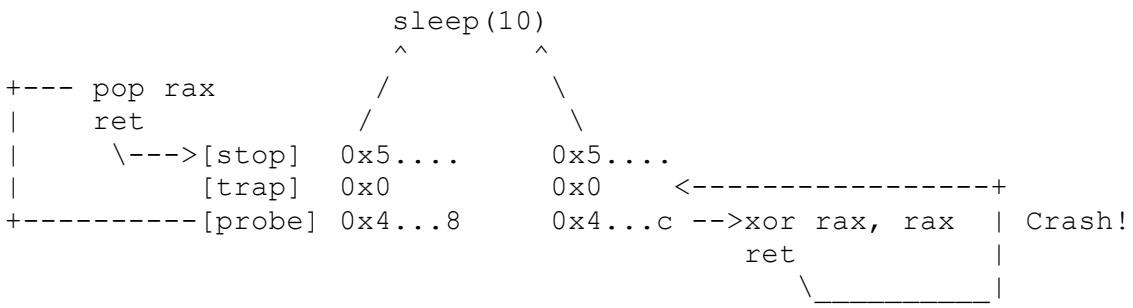
STEP 1: Find a stop gadget
A stop gadget is a return address that points to code that will hang the program, but not crash it. Once the attacker can defeat canaries, he can overwrite the overflown function's return address and start guessing locations for a stop gadget. If the client network connection suddenly closes, the guessed address was not a stop gadget. If the connection stays open, the gadget is a stop gadget.

STEP 2: Find gadgets that pop stack entries.

Once you have a stop gadget, you can use it to find other gadgets that pop entries off of the stack and into registers. There are three building blocks to locate stack popping gadgets:

- probe: Address of a potential stack popping gadget
- stop: Address of a stop gadget
- crash: Address of non-executable code (0x0)

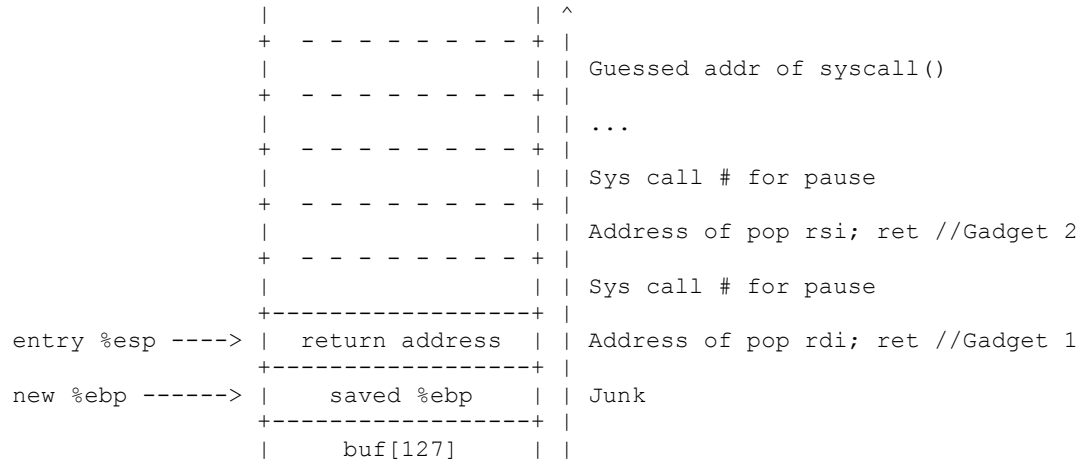
Example: Find a gadget that pops one thing off the stack.

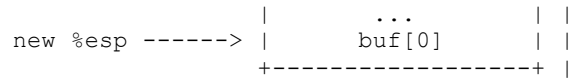


After you do this a bunch of times, you'll have a collection of gadgets that pop one thing from the stack and then return. However, you won't know which *register* those gadgets store the popped value in. You need to know which registers are used to store data so that you can issue a system call. Each system call expects its arguments to be in a specific set of registers.

Note that we also don't know the location of the syscall() library function.

STEP 3: Find syscall() and determine which registers the pop gadgets use
pause() is a system call that takes no arguments (and thus ignores everything in the registers). To find pause(), the attacker chains all of the "pop x; ret" gadgets on the stack, pushing the system call number for pause() as the "argument" for each gadget. At the bottom of the chain, the attacker places the guessed address for syscall().





So, at the end of this chain, the pop gadgets have placed the syscall number for pause() in a bunch of registers, hopefully including rax, which is the one that syscall() looks in to find the the syscall number.

Once this mega-gadget induces a pause, we know that we've determined the location of syscall(). Now we need to determine which gadget pops the top-of-the stack into rax. The attacker can figure this out by process-of-elimination: iterativel try just one gadget and see if you can invoke pause().

To identify arbitrary "pop x; ret" gadgets, you can use tricks with other system calls that use the x register that you're trying to find.

So, the outcome of this phase is knowledge of "pop x; ret" gadgets, location of syscall().

STEP 4: Invoke write()
Now we want to invoke the write call on the network socket that the server has with the attacker's client. We need the following gadgets:

```
pop rdi; ret (socket)
pop rsi; ret (buffer)
pop rdx; ret (length)
pop rax; ret (write syscall number)
syscall
```

We have to guess the socket value, but that's fairly easy to do, since Linux restricts processes to 1024 simultaneously open file descriptors, and new file descriptors have to be the lowest one available (so guessing a small file descriptor works well in practice).

To test whether we've guessed the correct file descriptor, simply try the write and see if we receive anything!

Once we have the socket number, we issue a write, and for the data to send . . . we send a pointer to the program's .text segment! This allows the attacker to read the program's code (which was randomized but now totally known to the attacker!) Now the attacker can find more powerful gadgets directly, and leverage those gadgets to open a shell.

Defenses against BROP

- Re-randomize the canaries and the address space after each crash!
    - Use exec() instead of fork() to create processes, since fork() copies the address space of the parent to the child.

    - Interesting, Windows is vulnerable to BROP because Windows has no fork() equivalent.
- Sleep-on-crash?
    - Now a BROP attack is a denial-of-service!
- Bounds-checking?
    - Up to 2x performance overhead . . .

More info on ROP and x86 calling conventions:

- http://www.slideshare.net/saumilshah/dive-into-rop-a-quick-introduction-to return-oriented-programming
- https://cseweb.ucsd.edu/~hovav/dist/rop.pdf

6.858 Computer Systems Security

Fall 2014