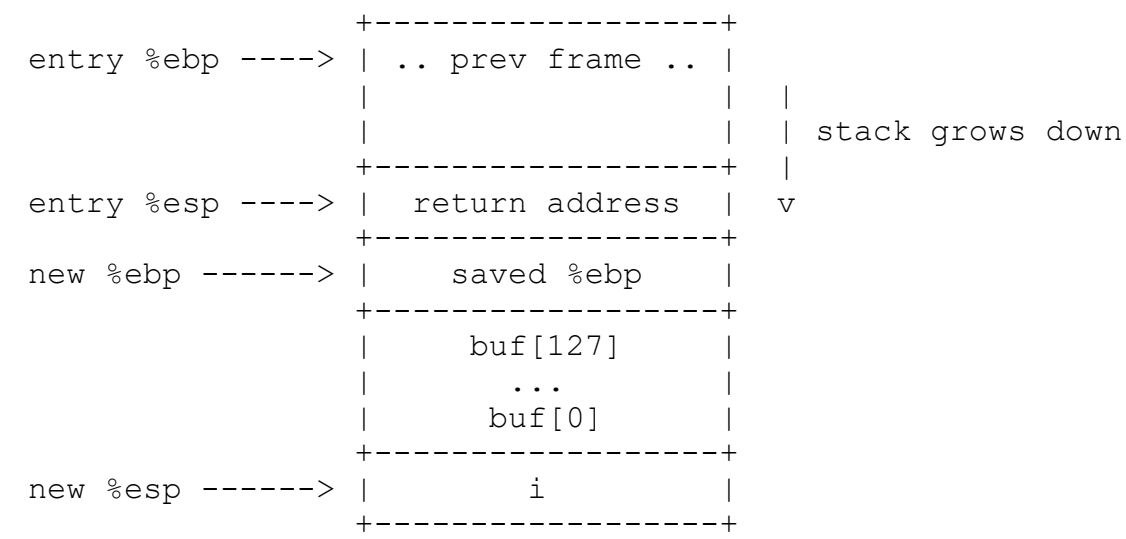
- Systems software is often written in C (operating systems, file systems, databases, compilers, network servers, command shells and console utilities)
- C is essentially high---level assembly, so . . .
    - Exposes raw pointers to memory
    - Does not perform bounds---checking on arrays (b/c the hardware 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.

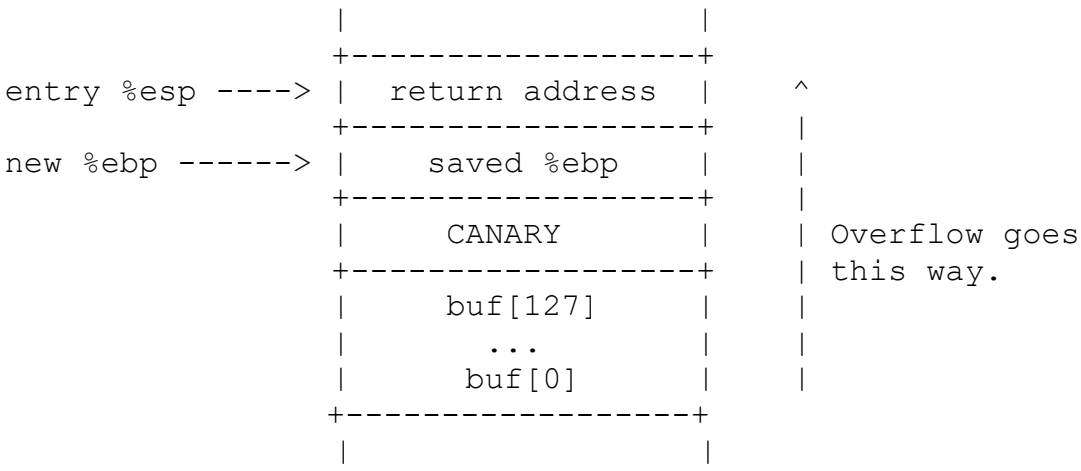
Good: Even partial analysis is useful, since programs should become strictly less buggy. For example, baggy bounds checking cannot catch all memory errors, but it can detect many important kinds.

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.

Mitigation approach 2: 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 3: Randomize memory address (ASLR, stack …)

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 not care exactly where to jump.
  - Ex: "Heap spraying": fill memory w/ shellcode so that a random jump is OK!

    Suppose the target code takes user supplied input in a heap allocated address.

```
read_user_string()
{
   char *str = malloc(128);
   gets(str); /* the target program reads string in a heap location */
   …..
   free(str);
}
```

    Suppose the implementation of free does not clear buffers and adversary somehow manage to supply a lot of shellcode as input, then it is highly likely that a large number of heap addresses in the process address space contain shellcode. Moreover, an attacker can pad the shellcode with a large number of nop instructions (think why?). Now an attacker can launch an attack by simply guessing an arbitrary heap location (which may contain nop that leads to shellcode), and overwrite the return address with it.

- 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.

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.

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 jump to the middle of the code and only execute a desired piece of that code.

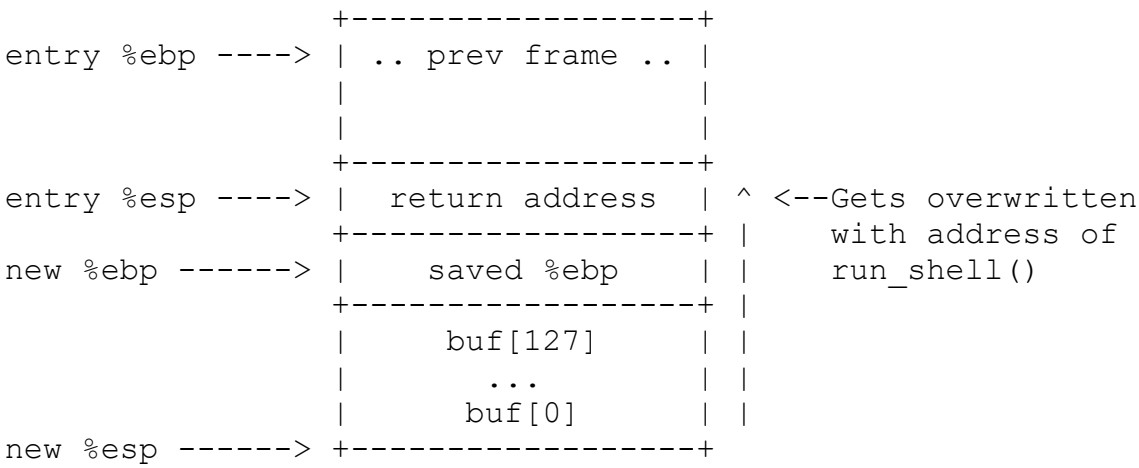
These kinds of attacks are called return---oriented programming, or ROP. To understand how ROP works, let's examine a simple C program that has a security vulnerability.

```
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 the 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.



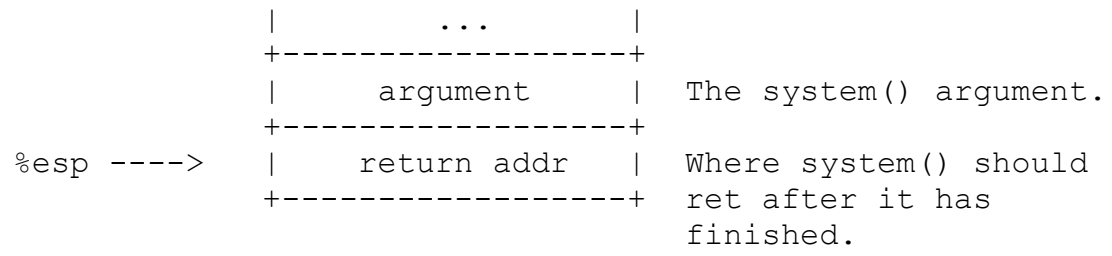
That's a straightforward extension of the buffer overflows that we've already looked at. But how can we pass arguments to the function that we're jumping to?

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

void run_cmd(){
    system("/something/boring");
}

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

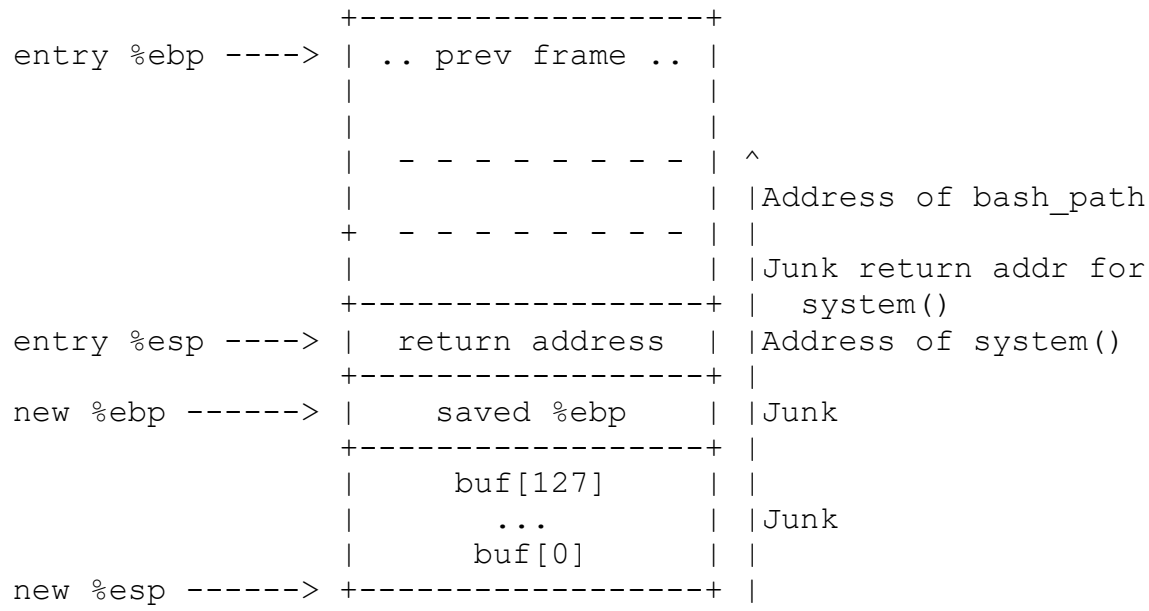
In this case, the argument that we want to pass to is already located in the program code. There's also a preexisting call to system(), but that call isn't passing the argument that we want.

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:

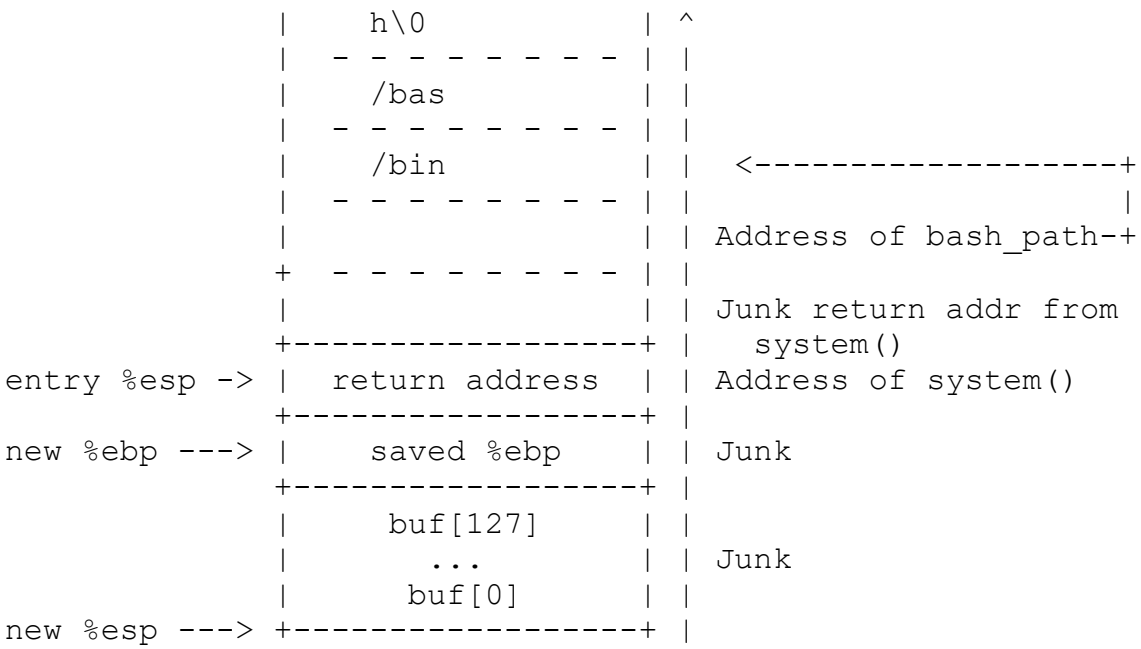


So, the buffer overflow needs to set up a stack that looks like this:



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.



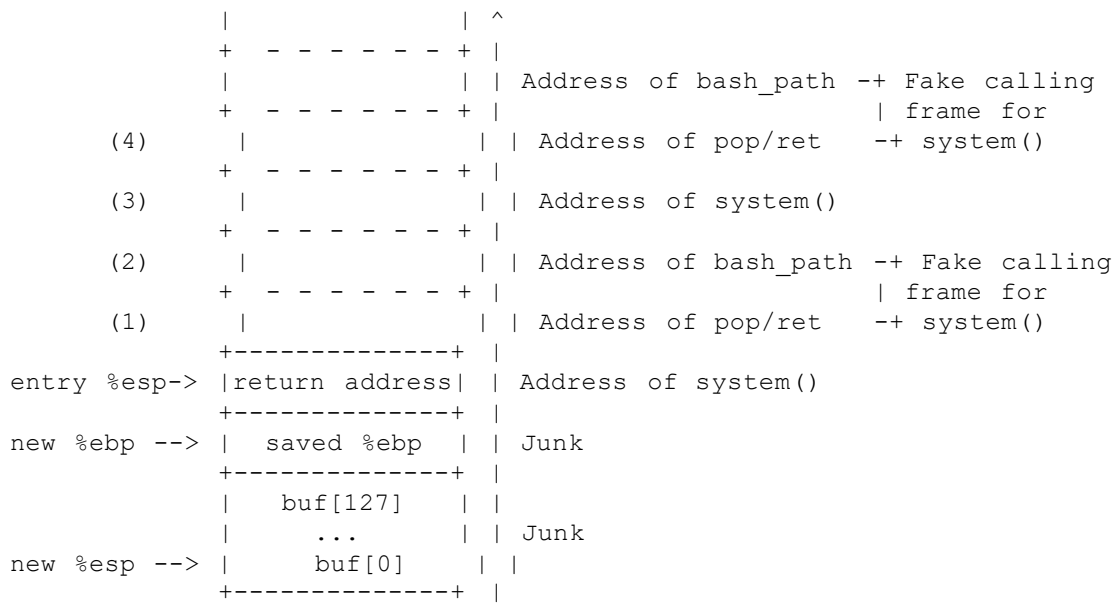
Note that, in these examples, I've been assuming that the attacker used a junk return address from system(). However, the attacker could set it to something useful. In fact, by setting it to something useful, the attacker can chain calls together!

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).