CSCI 331: Introduction to Computer Security

Lecture 15: Shellcode

Instructor: Dan Barowy

Williams

Announcements

Williams Computer Science Presents:
Class of '60s Speaker, Jon Kleinberg, Cornell University

Thursday, November 04 @ 7:00pm in Wege
"Designing Algorithms for High-Stakes Decisions"

As algorithms are deployed to make sensitive decisions about people, recent work has focused on several challenges that can result. First, concerns have been raised about the effects of algorithmic monoculture, in which multiple decision-makers all rely on the same algorithm. In a set of models drawing on minimal assumptions, we show that when competing decision-makers converge on the use of the same algorithm as part of a decision pipeline, the result can potentially be harmful for social welfare even when the algorithm is more accurate than any decision-maker acting on their own. Second, we consider some of the canonical ways in which data is simplified over the course of these decision-making pipelines, showing how this process of simplification can introduce effects that connect to principles from the psychology of stereotype formation. (This talk will be based on joint work with Sendhil Mullainathan and Manish Raghavan.)

Announcements

Williams Computer Science Presents:
Class of '60s Speaker, Jon Kleinberg, Cornell University

Friday, November 05 @ 2:35pm in Wege
"Monoculture and Simplicity in Algorithmic Decision-Making"

Topics

Paper discussion

Defensive programming

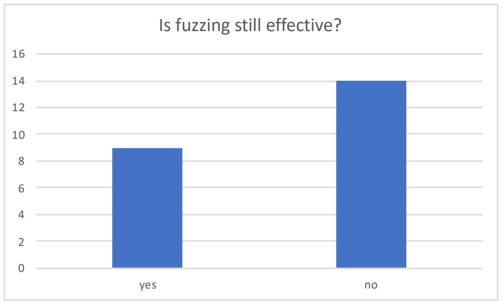
Shellcode: the big idea

Crafting shellcode

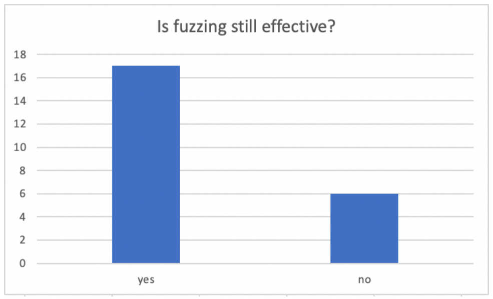
Your to-dos

1. Lab 5, **due Sunday 11/7**.
2. Reading response (Wang), **due Wed 11/10**.
3. Project part 2, **due Sunday 11/14**.

Paper discussion

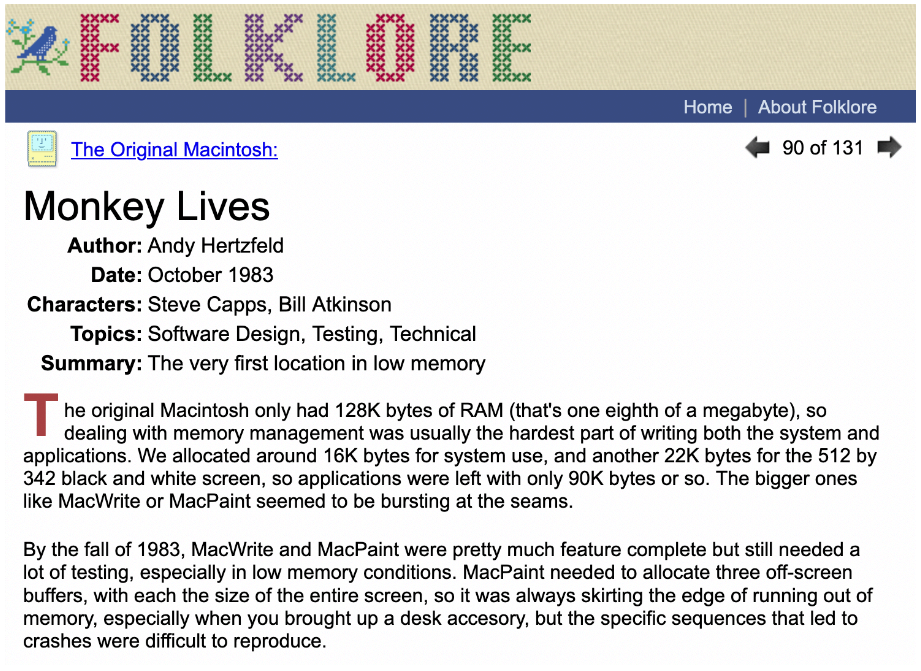


2019



2021

Steve Capps



Monkey Lives

Author: Andy Hertzfeld
Date: October 1983
Characters: Steve Capps, Bill Atkinson
Topics: Software Design, Testing, Technical
Summary: The very first location in low memory

The original Macintosh only had 128K bytes of RAM (that's one eighth of a megabyte), so dealing with memory management was usually the hardest part of writing both the system and applications. We allocated around 16K bytes for system use, and another 22K bytes for the 512 by 342 black and white screen, so applications were left with only 90K bytes or so. The bigger ones like MacWrite or MacPaint seemed to be bursting at the seams.

By the fall of 1983, MacWrite and MacPaint were pretty much feature complete but still needed a lot of testing, especially in low memory conditions. MacPaint needed to allocate three off-screen buffers, with each the size of the entire screen, so it was always skirting the edge of running out of memory, especially when you brought up a desk accesory, but the specific sequences that led to crashes were difficult to reproduce.

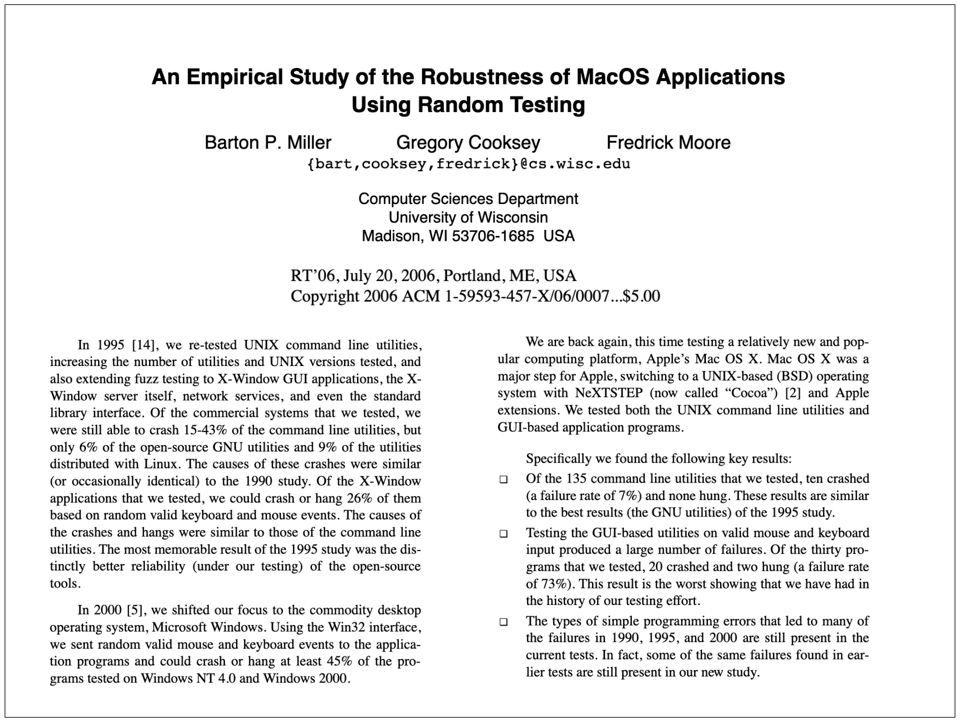
Steve Capps had been working on a "journaling" feature for the "Guided Tour" tutorial disc, where the Macintosh could demo itself by replaying back events that were recorded in a prior session. He realized that the so-called "journaling hooks" that were used to feed pre-recorded events to the system could also be the basis of a testing tool he called "The Monkey".

The Monkey was a small desk accessory that used the journaling hooks to feed random events to the current application, so the Macintosh seemed to be operated by an incredibly fast, somewhat angry monkey, banging away at the mouse and keyboard, generating clicks and drags at random positions with wild abandon. It had great potential as a testing tool, so Capps refined it to generate more semantically rich events, with a certain percentage of the events as menu commands, a certain percentage as window drags, etc.

The Monkey proved to be an excellent testing tool, and a great amusement, as well. Its manic activity was sort of hypnotic, and it was interesting to see what kind of MacPaint pictures the Monkey could draw, or if it would ever produce something intelligible in MacWrite. At first it could crash the system fairly easily, but soon we fixed the more obvious bugs. We thought it would be a good test for an application to see if it could run the Monkey all night, but usually it didn't run for more than 20 minutes, even if it didn't crash, because the Monkey would invariably select the quit command.

Bill Atkinson came up with the idea of defining a system flag called "MonkeyLives" (pronounced with a short "i" but often mispronounced with a long one), that indicated when the Monkey was running. The flag allowed MacPaint and other applications to test for the presence of the Monkey and disable the quit command while it was running, as well as other areas they wanted the Monkey to avoid. This allowed the Monkey to run all night, or even longer, driving the application through every possible situation.

afl-fuzz demo



An Empirical Study of the Robustness of MacOS Applications Using Random Testing

Barton P. Miller Gregory Cooksey Fredrick Moore
{bart,cooksey,fredrick}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685 USA

RT'06, July 20, 2006, Portland, ME, USA
Copyright 2006 ACM 1-59593-457-X/06/0007...$5.00

In 1995 [14], we re-tested UNIX command line utilities, increasing the number of utilities and UNIX versions tested, and also extending fuzz testing to X-Window GUI applications, the X-Window server itself, network services, and even the standard library interface. Of the commercial systems that we tested, we were still able to crash 15-43% of the command line utilities, but only 6% of the open-source GNU utilities and 9% of the utilities distributed with Linux. The causes of these crashes were similar (or occasionally identical) to the 1990 study. Of the X-Window applications that we tested, we could crash or hang 26% of them based on random valid keyboard and mouse events. The causes of the crashes and hangs were similar to those of the command line utilities. The most memorable result of the 1995 study was the distinctly better reliability (under our testing) of the open-source tools.

In 2000 [5], we shifted our focus to the commodity desktop operating system, Microsoft Windows. Using the Win32 interface, we sent random valid mouse and keyboard events to the application programs and could crash or hang at least 45% of the programs tested on Windows NT 4.0 and Windows 2000.

We are back again, this time testing a relatively new and popular computing platform, Apple's Mac OS X. Mac OS X was a major step for Apple, switching to a UNIX-based (BSD) operating system with NeXTSTEP (now called "Cocoa") [2] and Apple extensions. We tested both the UNIX command line utilities and GUI-based application programs.

Specifically we found the following key results:

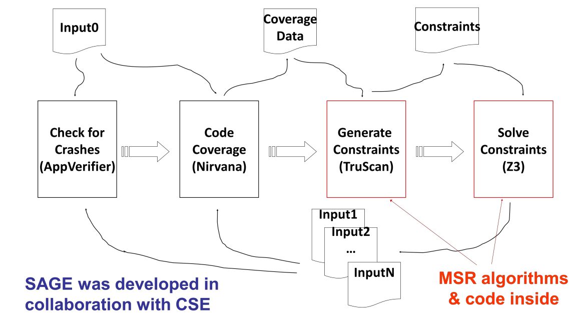
- ❑ Of the 135 command line utilities that we tested, ten crashed (a failure rate of 7%) and none hung. These results are similar to the best results (the GNU utilities) of the 1995 study.
- ❑ Testing the GUI-based utilities on valid mouse and keyboard input produced a large number of failures. Of the thirty programs that we tested, 20 crashed and two hung (a failure rate of 73%). This result is the worst showing that we have had in the history of our testing effort.
- ❑ The types of simple programming errors that led to many of the failures in 1990, 1995, and 2000 are still present in the current tests. In fact, some of the same failures found in earlier tests are still present in our new study.

PLDI 2013 Seattle
Microsoft Research Open House Event

SAGE: Whitebox Fuzzing for Security Testing

Ella Bounimova Patrice Godefroid David Molnar

Basic idea:
1. Run the program with first inputs,
2. gather constraints on inputs at conditional statements,
3. use a constraint solver to generate new test inputs,
4. repeat - possibly forever!



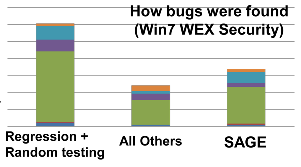
SAGE was developed in collaboration with CSE

MSR algorithms & code inside

Research Challenges:
- How to recover from imprecision ? PLDI'05, PLDI'11
- How to scale to billions of x86 instructions? NDSS'08
- How to check many properties together? EMSOFT'08
- How to leverage grammar specifications? PLDI'08
- How to deal with path explosion ? POPL'07,TACAS'08
- How to reason precisely about pointers? ISSTA'09
- How to deal with floating-point instr.? ISSTA'10
- How to deal with input-dependent loops? ISSTA'11
- How to synthesize x86 circuits automatically? PLDI'12
- How to run 24/7/365 for months at a time? ICSE'2013

+ research on constraint solvers

Impact: since 2007
- 500+ machine years (in largest fuzzing lab in the world)
- 3.4 Billion+ constraints (largest SMT solver usage ever!)
- 100s of apps, 100s of bugs (missed by everything else…)
- Ex: 1/3 of all Win7 WEX security bugs found by SAGE →
- Bug fixes shipped quietly (no MSRCs) to 1 Billion+ PCs
- Millions of dollars saved (for Microsoft and the world)
- SAGE is now used daily in Windows, Office, etc.



Common C bugs

```
gets
strcat
strcpy
printf
sprintf
vsprintf
scanf
strncpy (!!!)
strncat (!!!)
```

format string vulnerability

Arsenal of tools

- Memory safety: Address Sanitizer / Valgrind
- Fuzz testing: AFL-fuzz
- Unit testing: junit

```
 @Test
public void testAdditionExampleBased() {
   Calculator calculator = new Calculator();
   calculator.add(2);
   assertEquals(calculator.getResult(), 2);
}
```

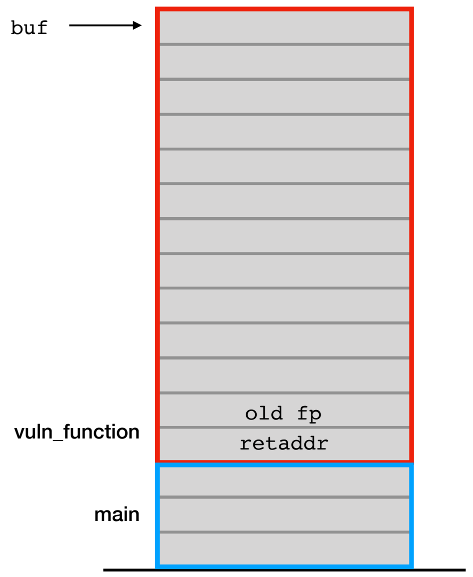
- Property-based testing: Quickcheck

```
 @Property(trials = 5)
 public void testAddition(int number) {
     Calculator calculator = new Calculator();
     calculator.add(number);
     assertEquals(calculator.getResult(), number);
 }
```

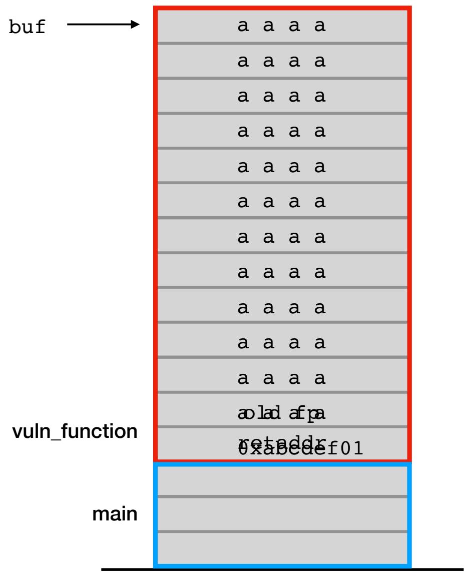
- Formal verification: Agda, Coq, etc.
  (remains very difficult— automated approaches remain open research problems)

Shellcode—the big idea

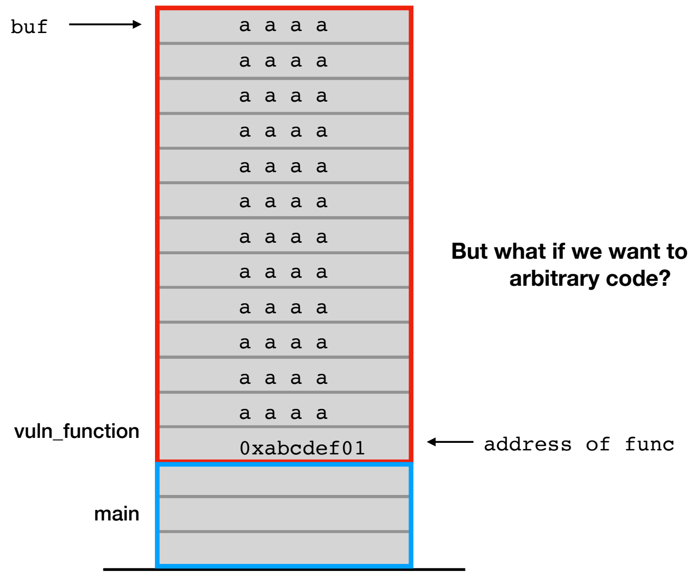
Make an overflow happen



Make an overflow happen

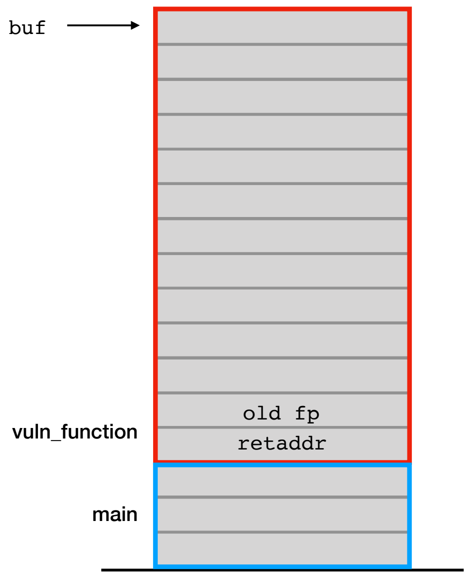


Arbitrary code



But what if we want to run arbitrary code?

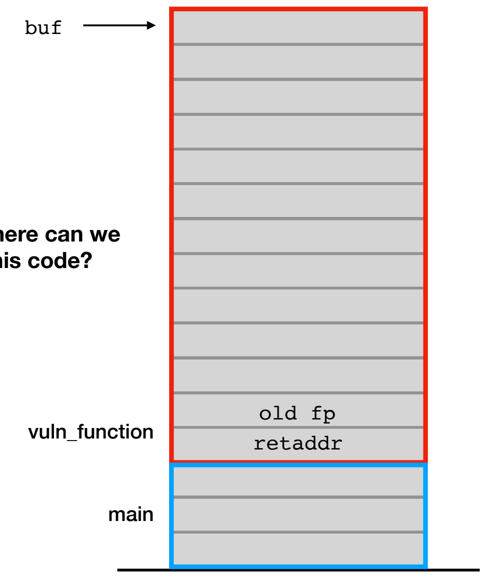
Arbitrary code



E.g.,

```
eor r2, r2
adr r1, shell
push {r1, fp, lr}
pop {r0, fp, lr}
strb r2, [r1, #7]
push {r1, fp, lr}
add fp, sp, #4
mov r7, #11
…
```

Arbitrary code

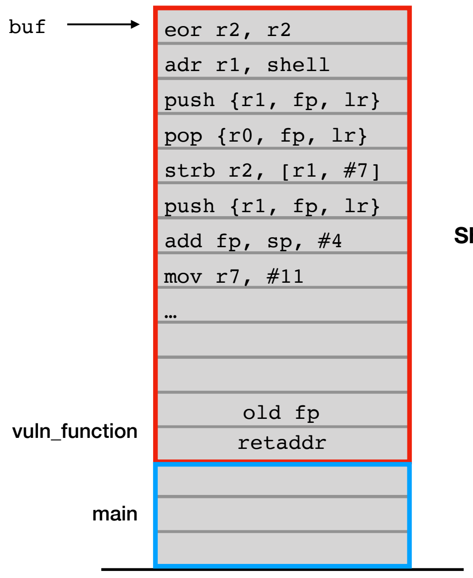


First, where can we put this code?

E.g.,

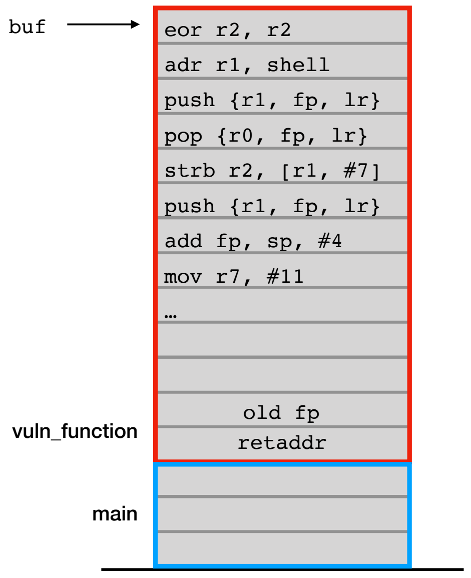
```
eor r2, r2
adr r1, shell
push {r1, fp, lr}
pop {r0, fp, lr}
strb r2, [r1, #7]
push {r1, fp, lr}
add fp, sp, #4
mov r7, #11
…
```

Put arbitrary code in the buffer



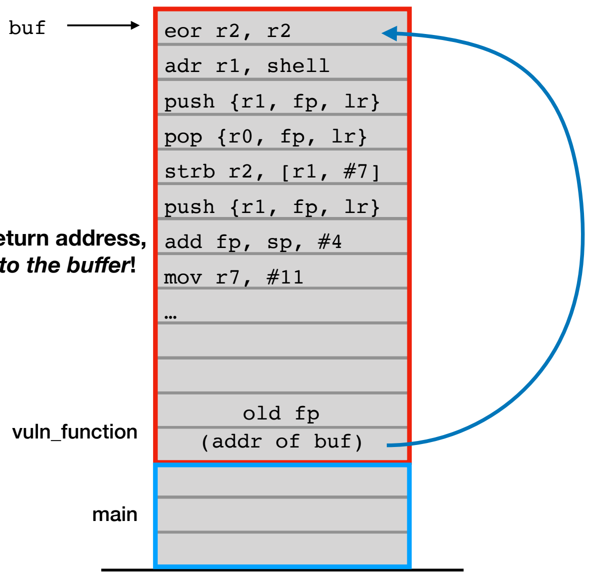
Short answer: in the buffer

How to call arbitrary code?



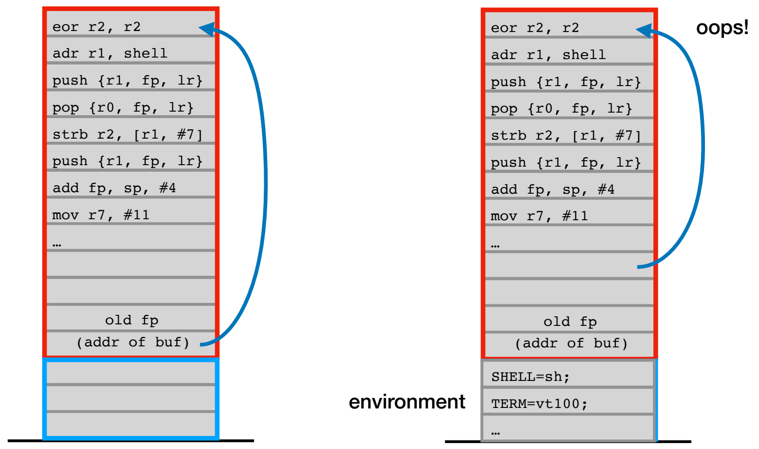
How do we call it?

How to call arbitrary code?

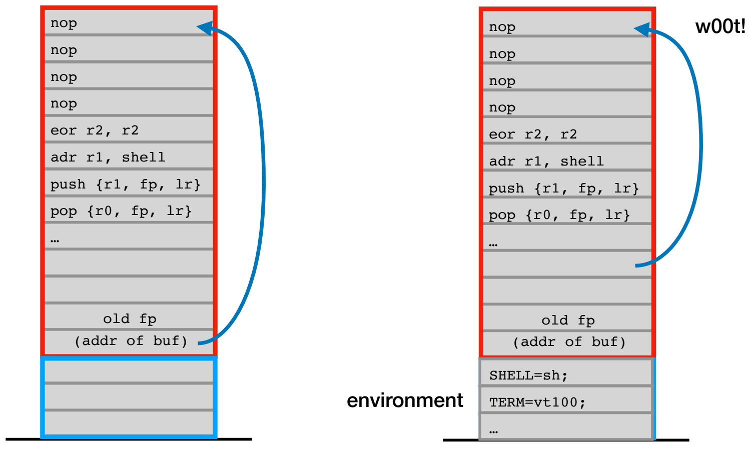


Overflow the return address, and point it *into the buffer*!

Caveat #1: environment variables



Fix #1: NOP sled



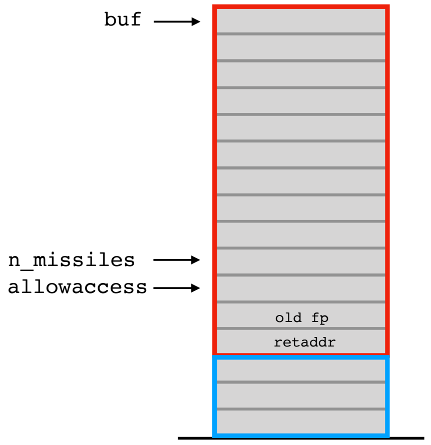
```
nop
nop
nop
nop
eor r2, r2
adr r1, shell
push {r1, fp, lr}
pop {r0, fp, lr}
…


old fp
(addr of buf)
```

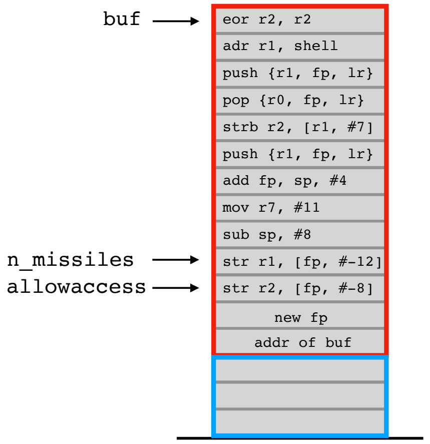
```
nop
nop
nop
nop
eor r2, r2
adr r1, shell
push {r1, fp, lr}
pop {r0, fp, lr}
…


old fp
(addr of buf)
SHELL=sh;
TERM=vt100;
…
```

Caveat #2: program does stuff after overflow

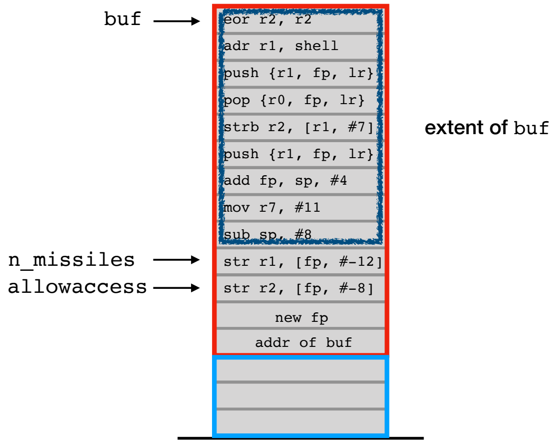


Caveat #2: program does stuff after overflow



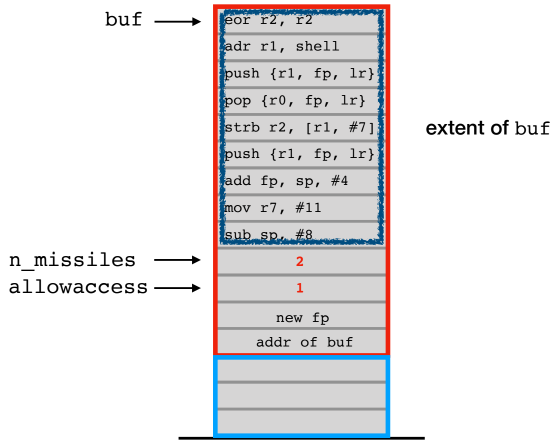
```
eor r2, r2
adr r1, shell
push {r1, fp, lr}
pop {r0, fp, lr}
strb r2, [r1, #7]
push {r1, fp, lr}
add fp, sp, #4
mov r7, #11
sub sp, #8
str r1, [fp, #-12]
str r2, [fp, #-8]
new fp
addr of buf
```

Caveat #2: program does stuff after overflow

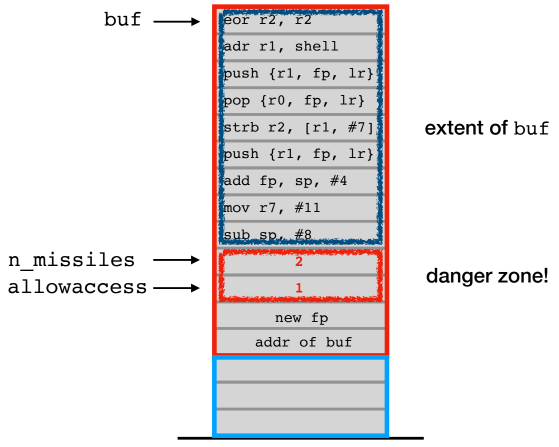


```
eor r2, r2
adr r1, shell
push {r1, fp, lr}
pop {r0, fp, lr}
strb r2, [r1, #7]
push {r1, fp, lr}
add fp, sp, #4
mov r7, #11
sub sp, #8
str r1, [fp, #-12]
str r2, [fp, #-8]
new fp
addr of buf
```

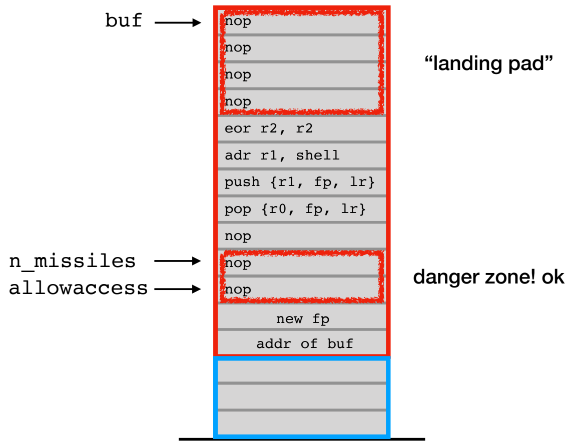
Caveat #2: program does stuff after overflow



Caveat #2: program does stuff after overflow



Fix #2: pad program on *both* sides



Recap & Next Class

Today we learned:

Shellcode: the big idea

Next class:

NULL byte removal