August 27: Course Overview and Intro to Memory Safety

Scribe: Grant Ho

1 Course Logistics

1. **Website:** `http://www.eecs.berkeley.edu/~raluca/cs261-f15/`
2. **Office Hours:** Tuesday 2-3pm in Soda 729
3. **Assignments:** Each class will have 1-2 required readings, posted on the course schedule. For each paper, write up a one to two paragraph summary per paper and email them to the homework email address (cs261.f15@gmail.com). Some student responses will be selected as examples for the benefit of the class; please indicate, in the first line of your email response, if you would like to opt out of this.
4. **Final Project:** A major component of this class is the final project. Students should form groups of two to three people and develop a concrete solution to a systems security problem. Project proposals are due on October 6, presentations will be held on December 1 and 3, and a project writeup will be due by the end of the course (date TBA). There are three important aspects that your project should address: first, your project should target a problem of practical importance/impact in the real world (purely theoretical works or projects focusing on implementation bugs in single commodity systems won't be suitable). Second, your project should assess existing work on the project and highlight novel contributions that your project makes. Finally, your project should develop a feasible, concrete solution; namely, if you propose a novel design/defense, you should at least develop a simple proof-of-concept implementation.

2 Systems Security Overview

What is security? From a scientific standpoint, systems do not have an intrinsic "security" property; there is no objective or definitive notion of security. Instead, we consider "security" to be how well a system can achieve a set of goals in the presence of defined adversaries. Varying from system to system, these goals might include things such as ensuring the availability of its service, providing certain levels of privacy and confidentiality, or guaranteeing the integrity of data or computations.

In our increasingly digitalized world, security has a litany of important applications. For example, securing cyber-physical systems such as cars, airplanes, and implanted medical devices is critical to protecting the physical safety of people. Beyond these immediate, tangible harms, secure

systems can protect sensitive medical, financial, and behavior data collected from users, as well as sensitive corporate and government data.

How do you build a secure system? There are three steps to building a secure system:

1. Identify the **security goals**/policies of your system
2. Identify your system's **threat models**: a threat model (also called an "attacker model") describes the types of adversaries your system can provide protection against, in terms of each adversary's power/capabilities.
   For example, an "active network attacker" is often described as an adversary who can view and actively modify all network traffic to/from your system (or user) in real time; however, we typically consider a network attacker to be unable to break correctly implemented secure channels (e.g., TLS). In contrast, a "web attacker" is an adversary who can only deliver malicious web content to a user's browser; however, this type of adversary cannot view or modify network traffic (e.g., consider an attacker who has the ability to trick a user into visiting arbitrary websites).
3. Build a mechanism/system that concretely achieves your goals in the face of these threat models

What goes wrong? Of course this high level depiction omits many of the practical challenges to building a secure system. System designers can construct their systems with bad security policies; an online service might offer an password reset operation that allows anyone to reset a user's password upon answering a few basic personal questions (potentially leading to account compromise), or the service might allow anyone to query whether or not a given person uses the online service (potentially leaking private information about a person's behavior). Even if developers come up with excellent security goals, they might design their system with the incorrect threat models in mind. For example, email service providers typically assume that data stored on their servers is safe from an attacker; however, this threat model overlooks threats by insider attackers and powerful adversaries who manage to compromise the service provider's servers. Other examples of overlooked/insufficient threat models include overly ambitious computational assumptions (creating encryption protocols with short key lengths) and Tor's inability to protect against a global passive adversary. Finally, attackers can still compromise the security of systems with apt security goals and threat models if bugs exist in the design or implementation of defensive mechanisms.

```
1  void login() {
2      char buf[15];
3      int authenticated = 0;
4      gets(buf);
5      if !strcmp(buf, ``secretpwd") {
6          printf(``%s", ``Invalid password.\n")
7      }
8      else {
9          authenticated = 1
10     }
11
12     // do some more stuff
13
14     if authenticated != 0 { // give root priv to user }
15 }
```

Listing 1: Example of code with a buffer overflow vulnerability.

3 Intro to Memory Safety

3.1 Stack Smashing

Consider the C code sample in Listing 1. Line 4 introduces a buffer overflow vulnerability: the program reads user supplied input into a fixed length buffer until it reads in a new line or null terminator. However, because the length of this input is not checked, "gets()" can supply a string with length greater than 15 characters; this will cause the program to write content beyond the bounds of "buf". **Stack Smashing** attacks exploit buffer overflow vulnerabilities by crafting inputs to buffers to overwrite security-related variables on the stack.[1]

To see how stack smashing attacks enable malicious behavior, it's important to understand the memory layout used by common systems architectures; Figure 1 illustrates the layout of the login function's stack frame. For each function invoked by a program, the operating system generates a stack frame that includes the "return address" (the address of the next line of code to execute once the function has returned), a stack frame pointer for book keeping purposes, and the function's local variables.[2]

In x86 architectures, the stack starts at the top of the address space (or some high address value) and grows downward. On the other hand, buffers "grow" upward to higher memory addresses (since buffers are just arrays, the natural way to index into the next element is to add one to the pointer value). Thus, when a buffer overflow occurs, the program will write past the allocated buffer and overwrite other local variables or control flow data such as the frame pointer and return address. For example, if Mallory, an attacker, supplies the 16 character password, "aaaassssddddfff1", the

[1] The following Phrack article provides a good, in-depth overview of basic stack smashing attacks: `http://phrack.org/issues/49/14.html`

[2] For functions that have parameters/arguments, the parameters are typically stored above the return address (not shown here because login() does not have function parameters).

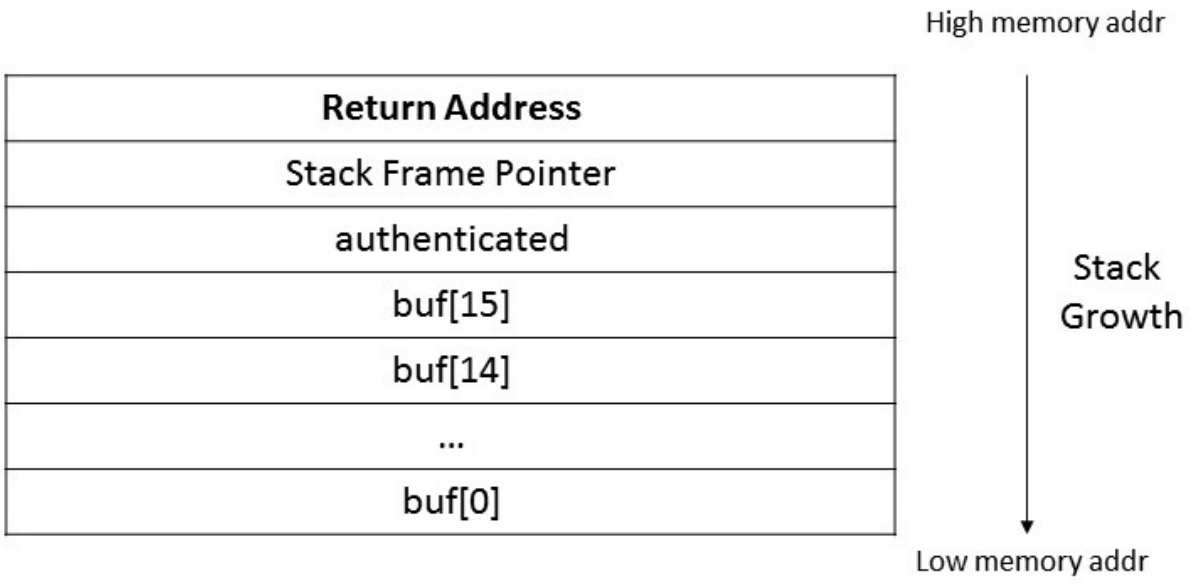

Figure 1: Stack Frame for Sample code

gets function will overwrite one byte of the *authenticated* variable; this will cause the comparison in line 14 to succeed, automatically authenticating the attacker even though she might not know a valid password.

However, the dangers of buffer overflows go beyond merely overwriting local variables. Notice that if Mallory supplies an even longer string to the gets() input, she can overwrite not only local variables, but also "control data" such as the stack frame pointer and the return address. Consider an input string of "aaaassssddddfff11112222\xff \xff \xff \xff". Assuming a 32-bit architecture where *authenticated* and the stack frame pointer each take up 4 bytes, this input string will cause gets() to overwrite the value of the return address to be 0xffffffff. As a result, when the login() function finishes (returns), the program will attempt to read and execute a line of code at 0xffffffff. While this will likely crash the program, if Mallory carefully chooses the overwritten return address to point to code she controls, she can trick the program into executing arbitrary code. For instance, Mallory could put malicious code in a string/buffer and overwrite login()'s return address to point to the address of this code buffer. Memory safety attacks like these, which cause a vulnerable program to begin executing unintended/malicious code, are often called **control flow hijacking attacks**.