CSE 564 – Winter 2019
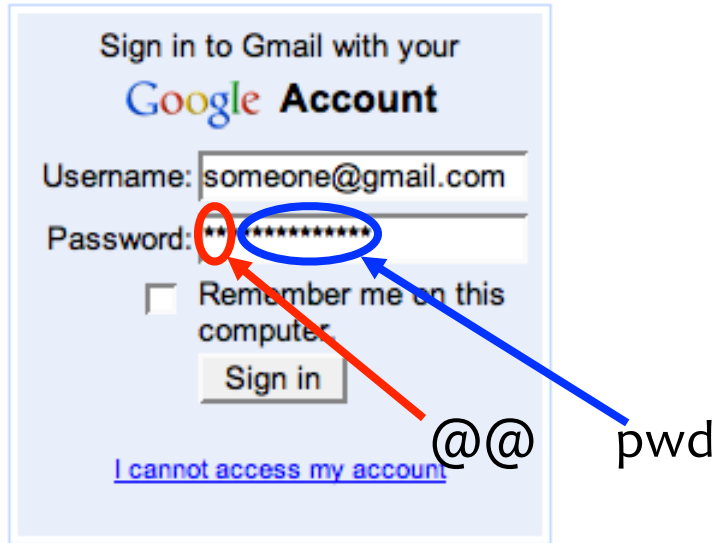
Addendum: Password Manager User Studies

Franziska (Franzi) Roesner

franzi@cs.washington.edu

PwdHash   Password Multiplier

@@   pwd

@@ in front of passwords
to protect; or F2

sitePwd = Hash(pwd,domain)

↑

Prevent phishing attacks

Activate with Alt-P or
double-click

sitePwd = Hash(username,
            pwd, domain)

Task Completion Results

| | Success | Potentially Causing Security Exposures | | | |
| | | Dangerous Success | Failures | | |
			Failure	False Completion	Failed due to Previous
PwdHash					
Log In	48%	44%	8%	0%	N/A
Migrate Pwd	42%	35%	11%	11%	N/A
Remote Login	27%	42%	31%	0%	N/A
Update Pwd	19%	65%	8%	8%	N/A
Second Login	52%	28%	4%	0%	16%
Password Multiplier					
Log In	48%	44%	8%	0%	N/A
Migrate Pwd	16%	32%	28%	20%	N/A
Remote Login	N/A	N/A	N/A	N/A	N/A
Update Pwd	16%	4%	44%	28%	N/A
Second Login	16%	4%	16%	0%	16%

Problem: Transparency

- Unclear to users whether actions successful or not.
  - Should be obvious when plugin activated.
  - Should be obvious when password protected.
- Users feel that they should be able to know their own password.

Problem: Mental Model

- Users seemed to have misaligned mental models

  - Not understand that one needs to put "@@" before *each* password to be protected.

  - Think different passwords generated for each session.

  - Think successful when were not.

  - Not know to click in field before Alt-P.

  - Don't understand what's happening: "Really, I don't see how my password is safer because of two @'s in front"

When "Nothing Works"

- Tendency to try all passwords
  - A poor security choice – phishing site could collect many passwords!
  - May make the use of PwdHash or Password Multiplier *worse* than not using any password manager.
- Usability problem leads to security vulnerabilities.

CSE 564 – Winter 2019

Buffer Overflows 101

Franziska (Franzi) Roesner

franzi@cs.washington.edu

Attacks on Memory Buffers

- Buffer is a pre-defined data storage area inside computer memory (stack or heap)

- Typical situation:
  - A function takes some input that it writes into a pre-allocated buffer.
  - The developer forgets to check that the size of the input isn't larger than the size of the buffer.
  - Uh oh.
    - "Normal" bad input: crash
    - "Adversarial" bad input : take control of execution

Stack Buffers

| | buf | uh oh! | |

- Suppose Web server contains this function

```
void func(char *str) {
        char buf[126];
        ...
        strcpy(buf,str);
        ...
}
```

- No bounds checking on strcpy()

- If str is longer than 126 bytes
  - Program may crash
  - Attacker may change program behavior

Example: Changing Flags

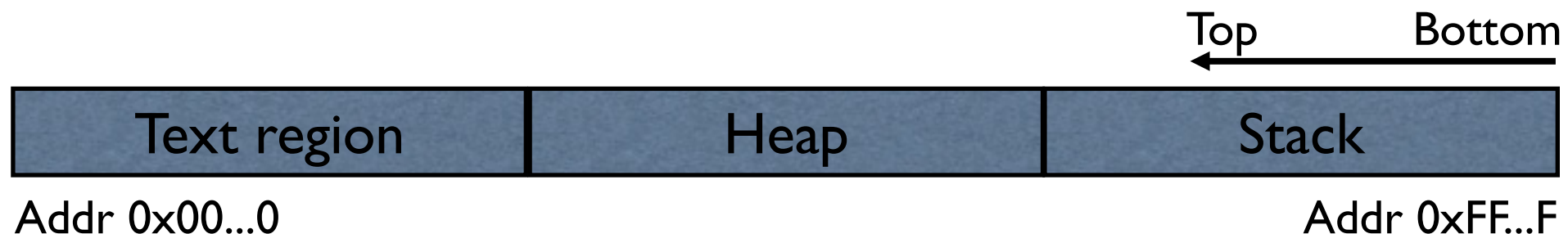|	buf	! (:-) !)	

- Suppose Web server contains this function

```
void func(char *str) {
        char buf[126];
        ...
        strcpy(buf,str);
        ...
}
```

- Authenticated variable non-zero when user has extra privileges

Memory Layout

- Text region: Executable code of the program
- Heap: Dynamically allocated data
- Stack: Local variables, function return addresses; grows and shrinks as functions are called and return

Top ← Bottom

Text region	Heap	Stack

Addr 0x00...0   Addr 0xFF...F
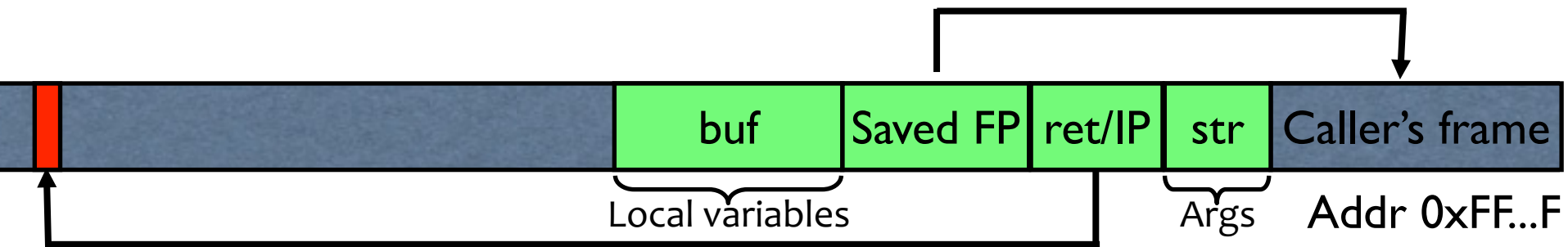
Stack Buffers

- Suppose Web server contains this function:

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
}
```

Allocate local buffer
(126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame (activation record) is pushed onto the stack.

			buf	Saved FP	ret/IP	str	Caller's frame

Local variables

Args

Addr 0xFF…F

Execute code at this address after func() finishes

What if Buffer is Overstuffed?

- Memory pointed to by str is copied onto stack...

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
    }
```

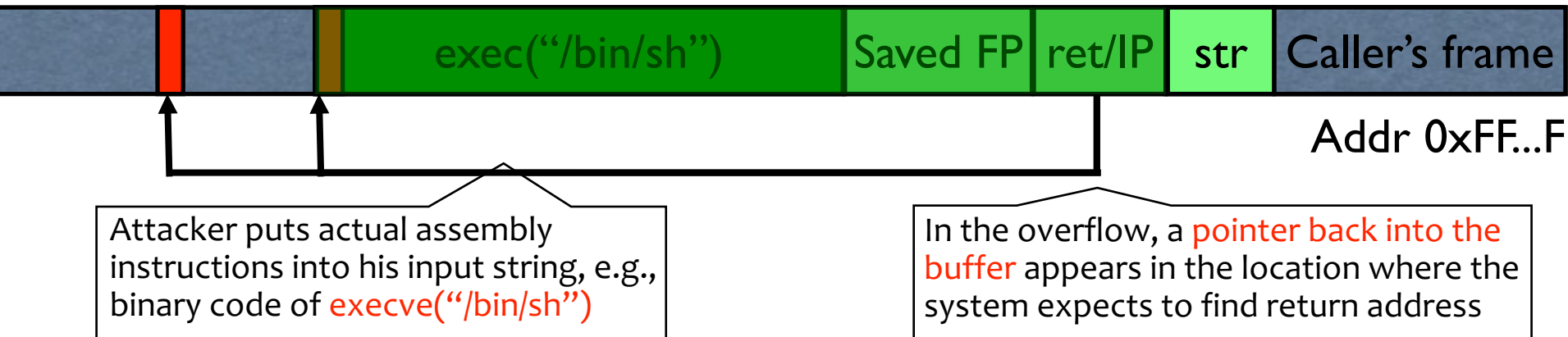strcpy does NOT check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

This will be interpreted as return address!

				buf	Saved FP	ret/IP	str	Caller's frame	

Local variables   Args   Addr 0xFF...F
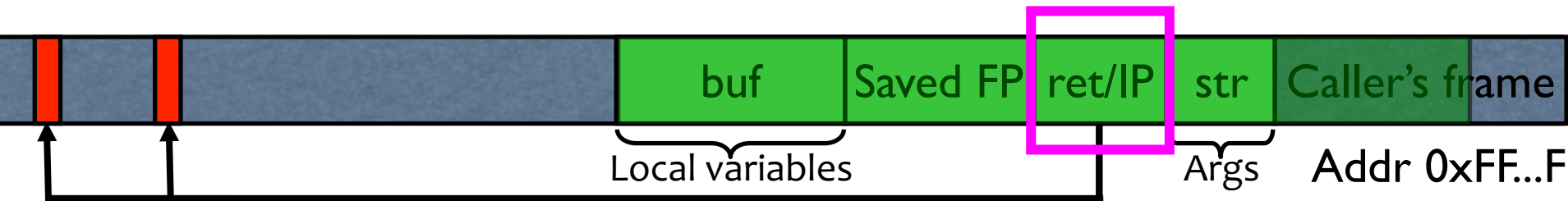
Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, str points to a string received from the network as the URL

		exec("/bin/sh")	Saved FP	ret/IP	str	Caller's frame

Addr 0xFF...F

Attacker puts actual assembly instructions into his input string, e.g., binary code of execve("/bin/sh")

In the overflow, a pointer back into the buffer appears in the location where the system expects to find return address

- When function exits, code in the buffer will be executed, giving attacker a shell **("shellcode")**
  - Root shell if the victim program is setuid root