

Software Security: Buffer Overflow Attacks and Beyond

Daniel Halperin
Tadayoshi Kohno

Thanks to Dan Boneh, Dieter Gollmann, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

Goals for Today

- ◆ Software security
 - Software lifecycle
 - Buffer overflow attacks
 - Other software security issues

- ◆ Lab 1 online this afternoon

Software Lifecycle (Simplified)

- ◆ Requirements
- ◆ Design
- ◆ Implementation
- ◆ Testing
- ◆ Use

Software Lifecycle (Simplified)

◆ Requirements

◆ Design

◆ Implementation

◆ Testing

◆ Use

Software problems are ubiquitous

Software Bug Halts F-22 Flight

Posted by kdawson on Sunday February 25, @06:35PM
from the [dare-you-to-cross-this-line](#) dept.

mgh02114 writes

"The new US stealth fighter, the [F-22 Raptor](#), was deployed for the first time to Asia earlier this month. On Feb. 11, twelve Raptors flying from Hawaii to Japan were forced to turn back when a software glitch crashed all of the F-22s' on-board computers as they crossed the international date line. The delay in arrival in Japan was [previously reported](#), with rumors of problems with the software. CNN television, however, this morning reported that every fighter completely lost all navigation and communications when they crossed the international date line. They reportedly had to turn around and follow their tankers by visual contact back to Hawaii. According to the CNN story, if they had not been with their tankers, or the weather had been bad, this would have been serious. CNN has not put up anything on their website yet."



Software problems are ubiquitous

1985-1987 -- Therac-25 medical accelerator. A radiation therapy device malfunctions and delivers lethal radiation doses at several medical facilities. Based upon a previous design, the **Therac-25** was an "improved" therapy system that could deliver two different kinds of radiation: either a low-power electron beam (beta particles) or X-rays. The Therac-25's X-rays were generated by smashing high-power electrons into a metal target positioned between the electron gun and the patient. A second "improvement" was the replacement of the older Therac-20's electromechanical safety interlocks with software control, a decision made because software was perceived to be more reliable.

What engineers didn't know was that both the 20 and the 25 were built upon an operating system that had been kludged together by a programmer with no formal training. Because of a subtle bug called a "**race condition**," a quick-fingered typist could accidentally configure the Therac-25 so the electron beam would fire in high-power mode but with the metal X-ray target out of position. At least five patients die; others are seriously injured.

<http://www.wired.com/software/coolapps/news/2005/11/69355>

Software problems are ubiquitous

January 15, 1990 -- AT&T Network Outage. A bug in a new release of the software that controls AT&T's #4ESS long distance switches causes these mammoth computers to crash when they receive a specific message from one of their neighboring machines -- a message that the neighbors send out when they recover from a crash.

One day a switch in New York crashes and reboots, causing its neighboring switches to **crash**, then their neighbors' neighbors, and so on. Soon, 114 switches are crashing and rebooting every six seconds, leaving an estimated 60 thousand people without long distance service for nine hours. The fix: engineers load the previous software release.

<http://www.wired.com/software/coolapps/news/2005/11/69355>

Software problems are ubiquitous

◆ NASA Mars Lander

- Bug in translation between English and metric units
- Cost taxpayers \$165 million

◆ Denver Airport baggage system

- Bug caused baggage carts to become out of “sync,” overloaded, etc.
- Delayed opening for 11 months, at \$1 million per day

◆ Other fatal or potentially fatal bugs

- US Vicennes tracking software
- MV-22 Osprey
- Medtronic Model 8870 Software Application Card

From *Exploiting Software* and <http://www.fda.gov/cdrh/recalls/recall-082404b-pressrelease.html>

Adversarial Failures

- ◆ Software bugs are bad
 - Consequences can be serious
- ◆ Even worse when an **intelligent adversary** wishes to **exploit** them!
 - Intelligent adversaries: Force bugs into “**worst possible**” conditions/states
 - Intelligent adversaries: Pick their targets
- ◆ **Buffer overflows bugs**: Big class of bugs
 - Normal conditions: Can sometimes cause systems to fail
 - Adversarial conditions: Attacker able to violate security of your system (control, obtain private information, ...)

Attacks

Buffer Overflows

A Bit of History: Morris Worm

- ◆ Worm was released in 1988 by Robert Morris
 - Graduate student at Cornell, son of NSA chief scientist
 - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
 - Now an EECS professor at MIT
- ◆ Worm was intended to propagate slowly and harmlessly measure the size of the Internet
- ◆ Due to a coding error, it created new copies as fast as it could and overloaded infected machines
- ◆ \$10-100M worth of damage

Morris Worm and Buffer Overflow

- ◆ One of the worm's propagation techniques was a **buffer overflow attack** against a vulnerable version of `fingerd` on VAX systems
 - By sending special string to finger daemon, worm caused it to execute code creating a new worm copy
 - Unable to determine remote OS version, worm also attacked `fingerd` on Suns running BSD, causing them to crash (instead of spawning a new copy)

More History

- ◆ Very common cause of Internet attacks
 - In 1998, over 50% of advisories published by CERT (computer security incident report team) were caused by buffer overflows
- ◆ Morris worm (1988): overflow in `fingerd`
 - 6,000 machines infected
- ◆ CodeRed (2001): overflow in MS-IIS server
 - 300,000 machines infected in 14 hours
- ◆ SQL Slammer (2003): overflow in MS-SQL server
 - 75,000 machines infected in **10 minutes (!)**

Attacks on Memory Buffers

- ◆ **Buffer** is a data storage area inside computer memory (stack or heap)
 - Intended to hold pre-defined amount of data
 - If more data is stuffed into it, it spills into adjacent memory
 - If executable code is supplied as “data”, victim’s machine may be fooled into executing it – we’ll see how
 - Can give attacker control over machine
- ◆ First generation exploits: stack smashing
- ◆ Later generations: heaps, function pointers, off-by-one, format strings and heap management structures

Stack Buffers

buf

- ◆ 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

Stack Buffers



buf

- ◆ 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

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

Changing Flags

buf

authenticated

- ◆ Suppose Web server contains this function

```
void func(char *str) {  
    int authenticated = 0;  
    char buf[126];  
    ...  
    strcpy(buf, str);  
    ...  
}
```

- ◆ **Authenticated** variable non-zero when user has extra privileges
- ◆ Morris worm also overflowed a buffer to overwrite an authenticated flag in fingerd

Changing Flags

buf

authenticated

- ◆ Suppose Web server contains this function

```
void func(char *str) {  
    int authenticated = 0;  
    char buf[126];  
    ...  
    strcpy(buf, str);  
    ...  
}
```

- ◆ **Authenticated** variable non-zero when user has extra privileges
- ◆ Morris worm also overflowed a buffer to overwrite an authenticated flag in fingerd

Changing Flags



buf

- ◆ Suppose Web server contains this function

```
void func(char *str) {  
    int authenticated = 0;  
    char buf[126];  
    ...  
    strcpy(buf, str);  
    ...  
}
```

- ◆ **Authenticated** variable non-zero when user has extra privileges
- ◆ Morris worm also overflowed a buffer to overwrite an authenticated flag in fingerd

Changing Flags

buf

I (yeah!)

- ◆ Suppose Web server contains this function

```
void func(char *str) {  
    int authenticated = 0;  
    char buf[126];  
    ...  
    strcpy(buf, str);  
    ...  
}
```

- ◆ **Authenticated** variable non-zero when user has extra privileges
- ◆ Morris worm also overflowed a buffer to overwrite an authenticated flag in fingerd

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

Text region

Heap

Stack

Addr 0x00...0

Addr 0xFF...F

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



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** with local variables is pushed onto the stack

Caller's frame

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** with local variables is pushed onto the stack

str

Caller's frame

Args

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** with local variables is pushed onto the stack

ret/IP

str

Caller's frame

Args

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** with local variables is pushed onto the stack



Stack Buffers

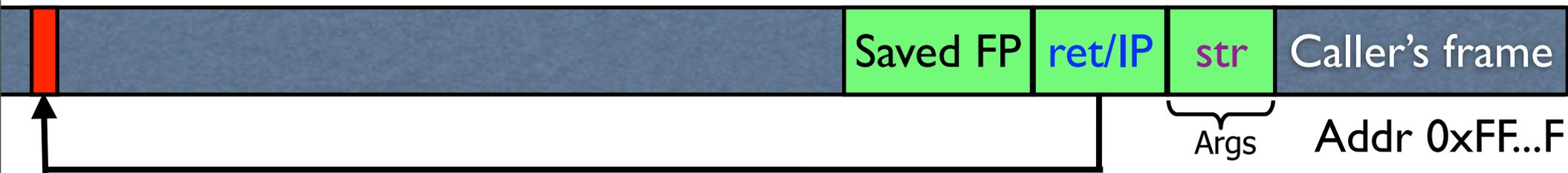
- ◆ Suppose Web server contains this function

```
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** with local variables is pushed onto the stack



Execute code at this address after func() finishes

Stack Buffers

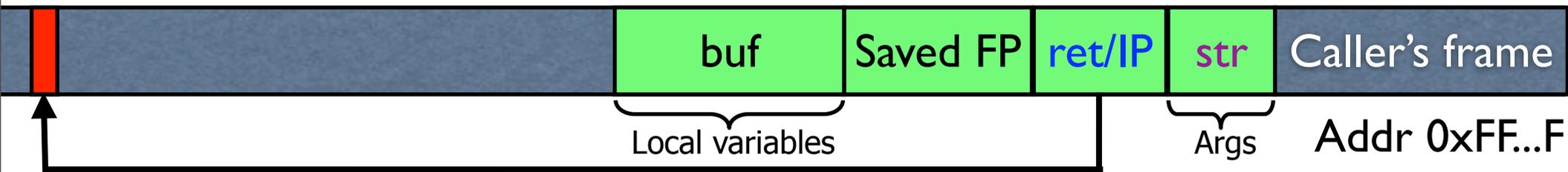
- ◆ Suppose Web server contains this function

```
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** with local variables is pushed onto the stack



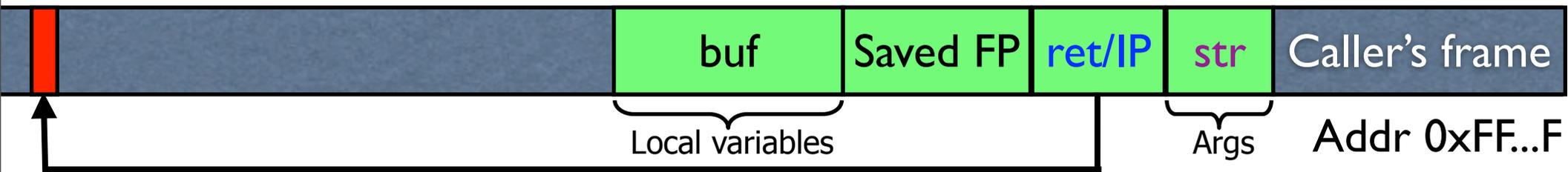
What If Buffer is Overstuffed?

- ◆ 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



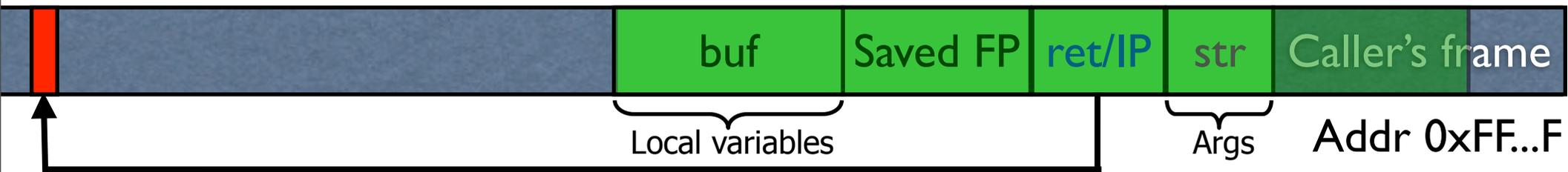
What If Buffer is Overstuffed?

- ◆ 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



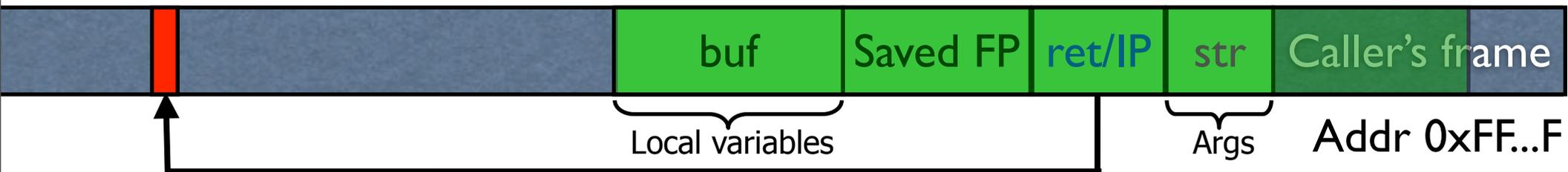
What If Buffer is Overstuffed?

- ◆ 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



Executing Attack Code

- ◆ Suppose buffer contains attacker-created string
 - For example, `*str` contains a string received from the network as input to some network service daemon



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

Executing Attack Code

- ◆ Suppose buffer contains attacker-created string
 - For example, *str contains a string received from the network as input to some network service daemon



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

Executing Attack Code

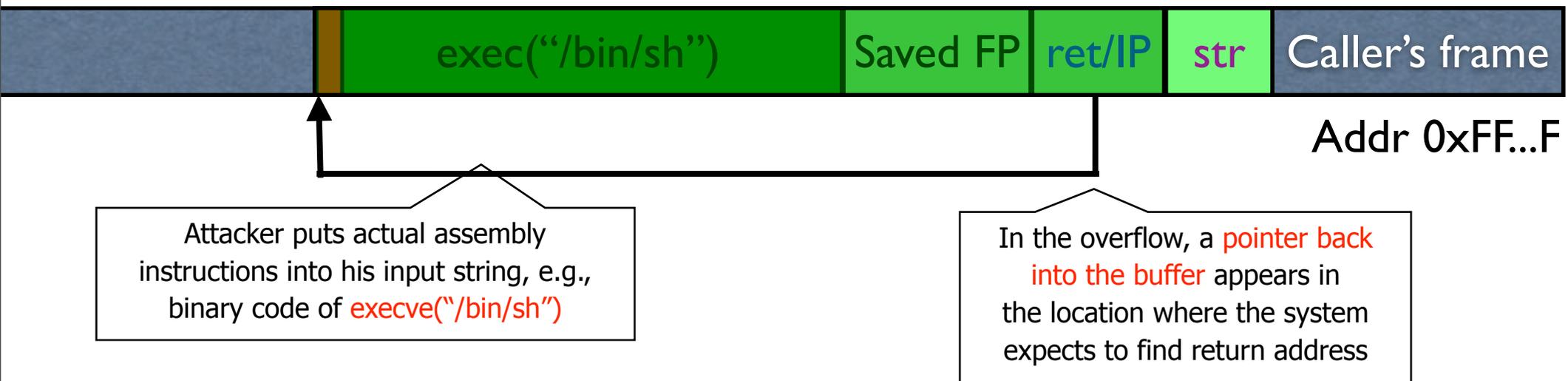
- ◆ Suppose buffer contains attacker-created string
 - For example, *str contains a string received from the network as input to some network service daemon



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

Executing Attack Code

- ◆ Suppose buffer contains attacker-created string
 - For example, `*str` contains a string received from the network as input to some network service daemon



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

Buffer Overflow Issues

- ◆ Executable attack code is stored on stack, inside the buffer containing attacker's string
 - Stack memory is supposed to contain only data, but...
- ◆ Overflow portion of the buffer must contain **correct address of attack code** in the RET position
 - The value in the RET position must point to the beginning of attack assembly code in the buffer
 - Otherwise application will (probably) crash with segmentation violation
 - Attacker must correctly guess in which stack position his buffer will be when the function is called

Problem: No Range Checking

- ◆ `strcpy` does not check input size
 - `strcpy(buf, str)` simply copies memory contents into `buf` starting from `*str` until `"\0"` is encountered, ignoring the size of area allocated to `buf`
- ◆ Many C library functions are unsafe
 - `strcpy(char *dest, const char *src)`
 - `strcat(char *dest, const char *src)`
 - `gets(char *s)`
 - `scanf(const char *format, ...)`
 - `printf(const char *format, ...)`

Does Range Checking Help?

- ◆ `strncpy(char *dest, const char *src, size_t n)`
 - If `strncpy` is used instead of `strcpy`, no more than `n` characters will be copied from `*src` to `*dest`
 - Programmer has to supply the right value of `n`
- ◆ Potential overflow in `htpasswd.c` (Apache 1.3):

```
strcpy(record,user);  
strcat(record,":");  
strcat(record,cpw); ...
```

Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

- ◆ Published "fix":

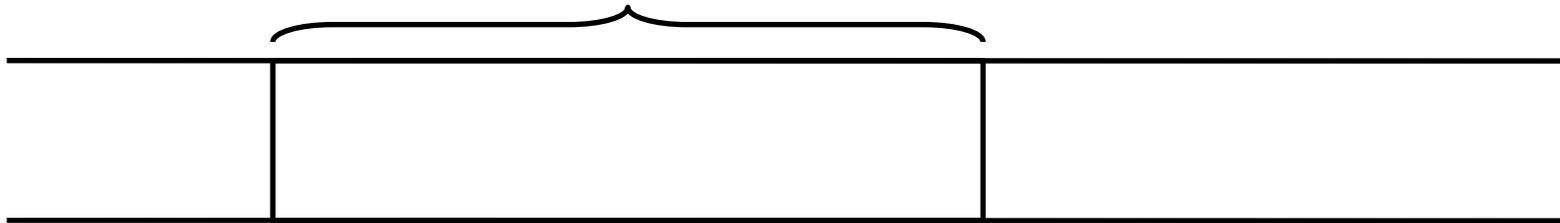
```
... strncpy(record,user,MAX_STRING_LEN-1);  
strcat(record,":");  
strncat(record,cpw,MAX_STRING_LEN-1); ...
```

Misuse of strncpy in httpasswd "Fix"

- ◆ Published "fix" for Apache httpasswd overflow:

```
... strncpy(record, user, MAX_STRING_LEN-1);  
   strcat(record, ":");  
   strncpy(record, cpw, MAX_STRING_LEN-1); ...
```

MAX_STRING_LEN bytes allocated for record buffer

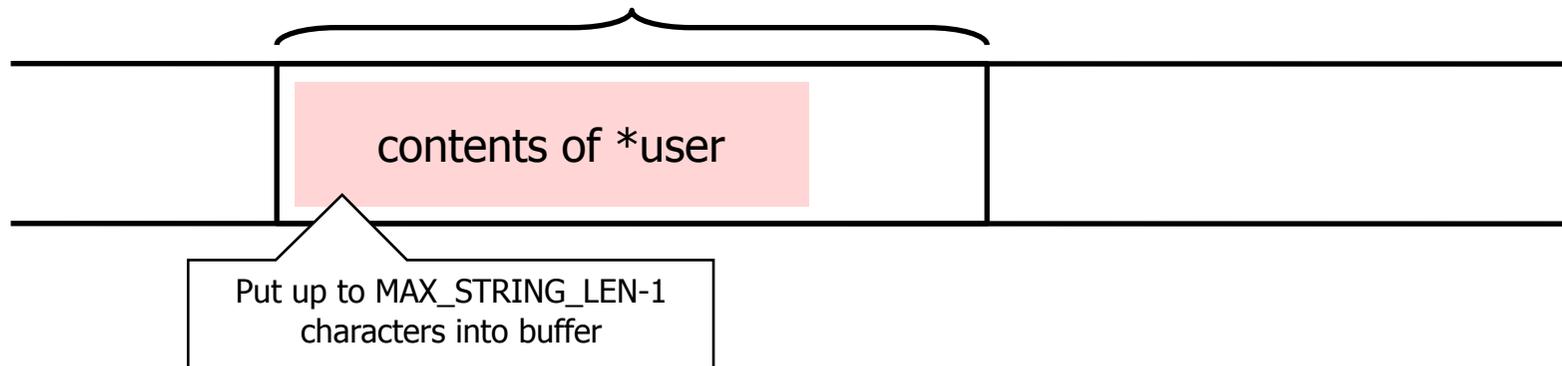


Misuse of strncpy in httpasswd "Fix"

- ◆ Published "fix" for Apache httpasswd overflow:

```
... strncpy(record,user,MAX_STRING_LEN-1);  
   strcat(record,":");  
   strncat(record,cpw,MAX_STRING_LEN-1); ...
```

MAX_STRING_LEN bytes allocated for record buffer

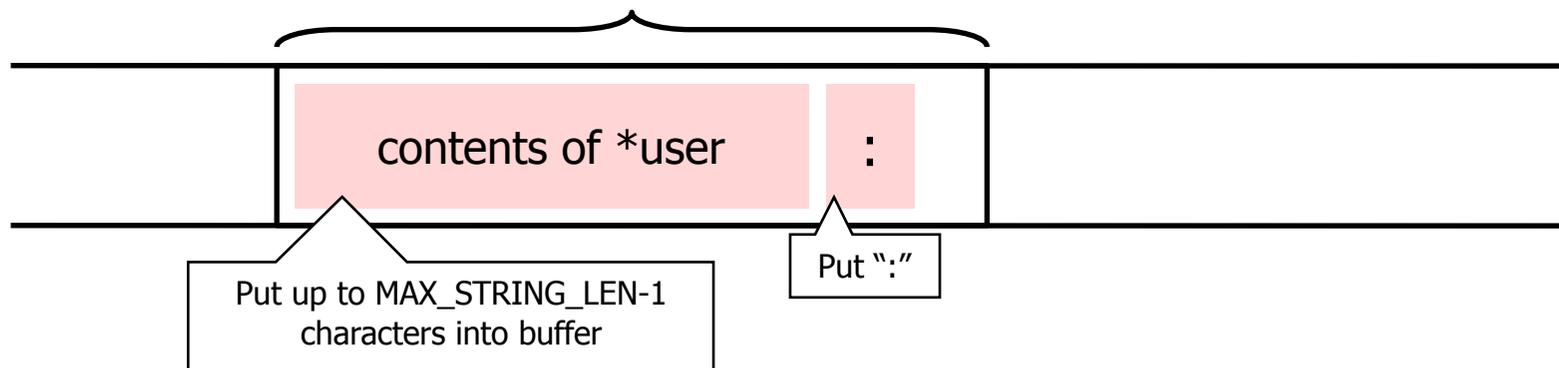


Misuse of strncpy in httpasswd "Fix"

- ◆ Published "fix" for Apache httpasswd overflow:

```
... strncpy(record, user, MAX_STRING_LEN-1);  
   strcat(record, ":");  
   strncat(record, cpw, MAX_STRING_LEN-1); ...
```

MAX_STRING_LEN bytes allocated for record buffer

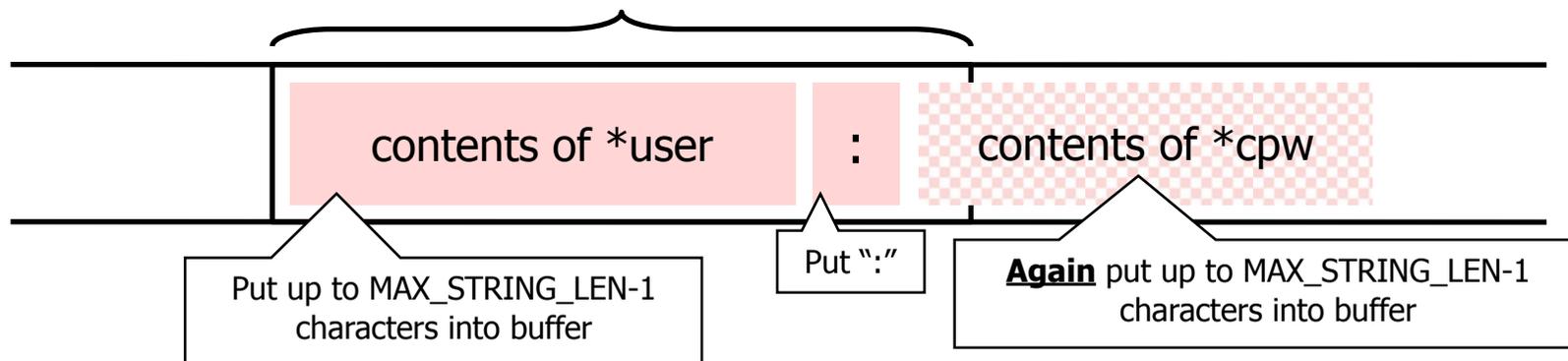


Misuse of strncpy in httpasswd "Fix"

- ◆ Published "fix" for Apache httpasswd overflow:

```
... strncpy(record, user, MAX_STRING_LEN-1);  
   strcat(record, ":");  
   strncpy(record, cpw, MAX_STRING_LEN-1); ...
```

MAX_STRING_LEN bytes allocated for record buffer



Off-By-One Overflow

◆ Home-brewed range-checking string copy

```
void notSoSafeCopy(char *input) {
    char buffer[512]; int i;
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
    if (argc==2)
        notSoSafeCopy(argv[1]);
}
```

Off-By-One Overflow

◆ Home-brewed range-checking string copy

```
void notSoSafeCopy(char *input) {
    char buffer[512]; int i;
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
    if (argc==2)
        notSoSafeCopy(argv[1]);
}
```

This will copy **513** characters into buffer. Oops!

Off-By-One Overflow

◆ Home-brewed range-checking string copy

```
void notSoSafeCopy(char *input) {
    char buffer[512]; int i;
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
    if (argc==2)
        notSoSafeCopy(argv[1]);
}
```

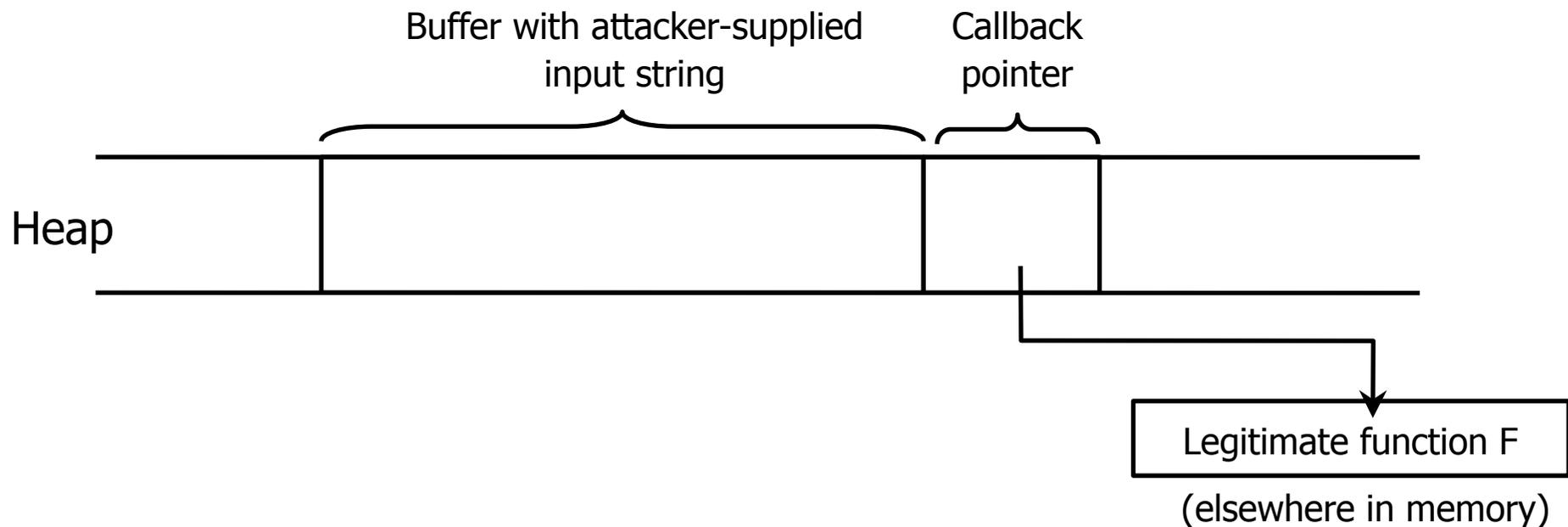
This will copy **513** characters into buffer. Oops!

◆ 1-byte overflow: can't change RET, but can change pointer to previous stack frame

- On little-endian architecture, make it point into buffer
- RET for previous function will be read from buffer!

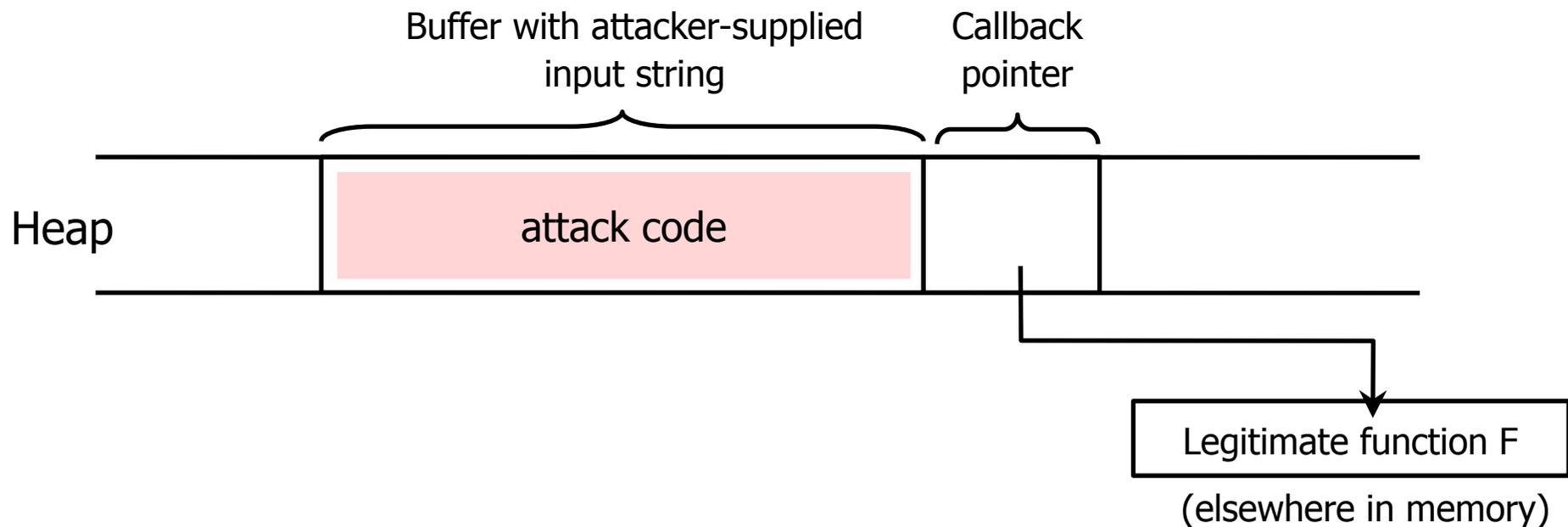
Function Pointer Overflow

- ◆ C uses **function pointers** for callbacks: if pointer to F is stored in memory location P, then another function G can call F as $(*P)(\dots)$



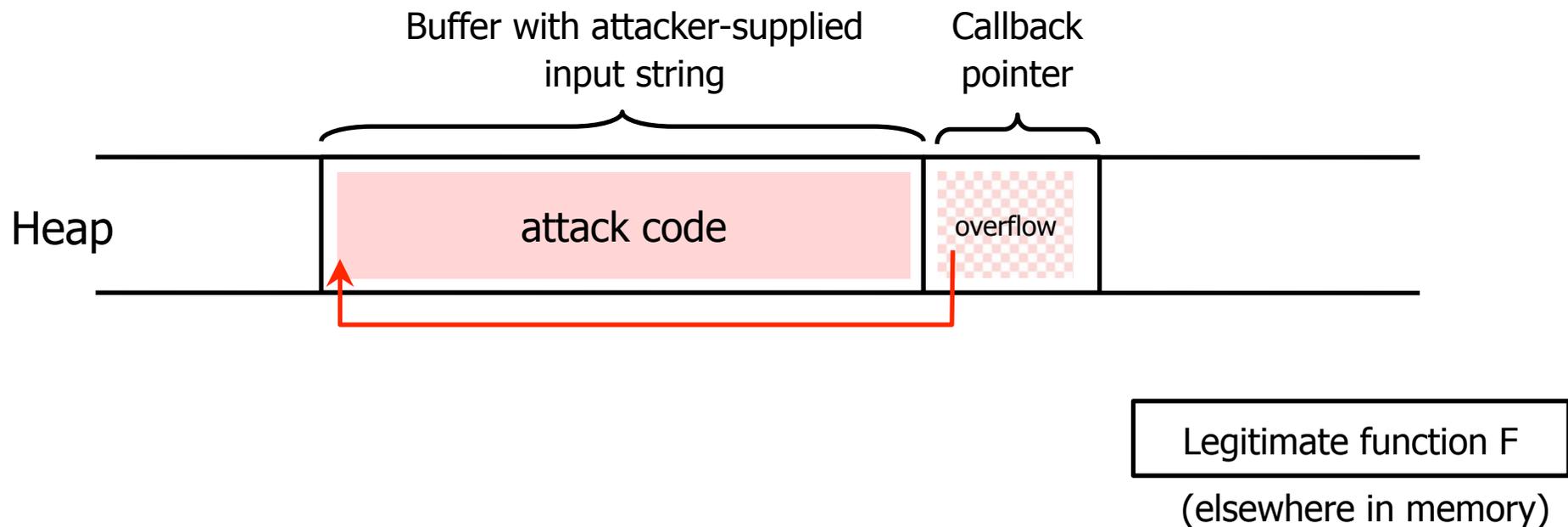
Function Pointer Overflow

- ◆ C uses **function pointers** for callbacks: if pointer to F is stored in memory location P, then another function G can call F as $(*P)(\dots)$



Function Pointer Overflow

- ◆ C uses **function pointers** for callbacks: if pointer to F is stored in memory location P, then another function G can call F as $(*P)(\dots)$



Format Strings in C

◆ Proper use of printf format string:

```
... int foo=1234;  
    printf("foo = %d in decimal, %X in hex",foo,foo); ...
```

– This will print

```
foo = 1234 in decimal, 4D2 in hex
```

◆ Sloppy use of printf format string:

```
... char buf[14]="Hello, world!";  
    printf(buf);  
    // should've used printf("%s", buf); ...
```

– If buffer contains format symbols starting with %, location pointed to by printf's internal stack pointer will be interpreted as an argument of printf. This can be exploited to move printf's internal stack pointer.

Viewing Memory

- ◆ `%x` format symbol tells `printf` to output data on stack

```
... printf("Here is an int:  %x", i); ...
```

- ◆ What if `printf` does not have an argument?

```
... char buf[16]="Here is an int:  %x";  
printf(buf); ...
```

- Stack location pointed to by `printf`'s internal stack pointer will be interpreted as an int. (What if crypto key, password, ...?)

- ◆ Or what about:

```
... char buf[16]="Here is a string:  %s";  
printf(buf); ...
```

- Stack location pointed to by `printf`'s internal stack pointer will be interpreted as a pointer to a string

Writing Stack with Format Strings

- ◆ `%n` format symbol tells `printf` to write the number of characters that have been printed

```
... printf("Overflow this!%n", &myVar); ...
```

- Argument of `printf` is interpreted as destination address
- This writes 14 into `myVar` ("Overflow this!" has 14 characters)

- ◆ What if `printf` does not have an argument?

```
... char buf[16]="Overflow this!%n";  
  
printf(buf); ...
```

- Stack location pointed to by `printf`'s internal stack pointer will be interpreted as address into which the number of characters will be written.

TOCTOU (Race Condition)

◆ TOCTOU == Time of Check to Time of Use

```
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISREG(s.st_mode)) {
        error("only allowed to regular files!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

◆ Goal: Open only regular files (not symlink, etc)

◆ Attacker can change meaning of `path` between `stat` and `open` (and access files he or she shouldn't)

Integer Overflow and Implicit Cast

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

(from www-inst.eecs.berkeley.edu—implflaws.pdf)

Integer Overflow and Implicit Cast

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```

(from www-inst.eecs.berkeley.edu—implflaws.pdf)

Integer Overflow and Implicit Cast

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```

- ◆ If **len** is negative, may copy huge amounts of input into buf

(from www-inst.eecs.berkeley.edu—implflaws.pdf)

Integer Overflow and Implicit Cast

```
size_t len = read_int_from_network();  
char *buf;  
buf = malloc(len+5);  
read(fd, buf, len);
```

(from www-inst.eecs.berkeley.edu—implflaws.pdf)

Integer Overflow and Implicit Cast

```
size_t len = read_int_from_network();  
char *buf;  
buf = malloc(len+5);  
read(fd, buf, len);
```

- ◆ What if `len` is large (e.g., `len = 0xFFFFFFFF`)?
- ◆ Then `len + 5 = 4` (on many platforms)
- ◆ Result: Allocate a 4-byte buffer, then read a lot of data into that buffer.

(from [www-inst.eecs.berkeley.edu—impl/flaws.pdf](http://www-inst.eecs.berkeley.edu/~impl/flaws.pdf))

Randomness issues

Randomness issues

- ◆ Many applications (especially security ones) require randomness

Randomness issues

- ◆ Many applications (especially security ones) require randomness
- ◆ Explicit uses:
 - Generate secret cryptographic keys
 - Generate random initialization vectors for encryption

Randomness issues

- ◆ Many applications (especially security ones) require randomness
- ◆ Explicit uses:
 - Generate secret cryptographic keys
 - Generate random initialization vectors for encryption
- ◆ Other “non-obvious” uses:
 - Generate passwords for new users
 - Shuffle the order of votes (in an electronic voting machine)
 - Shuffle cards (for an online gambling site)

C's rand() Function

- ◆ C has a built-in random function: `rand()`

```
unsigned long int next = 1;
/* rand:  return pseudo-random integer on 0..32767 */
int rand(void) {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand:  set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

- ◆ Problem: don't use `rand()` for security-critical applications!
 - Given a few sample outputs, you can predict subsequent ones

PS3

- ◆ Learn all about it in an example Current Event report from a past iteration of 484
 - <https://catalyst.uw.edu/gopost/conversation/kohno/452868>

Windows/.NET

July 22, 2001

Randomness and the Netscape Browser

How secure is the World Wide Web?

Ian Goldberg and David Wagner

No one was more surprised than Netscape Communications when a pair of computer-science students broke the Netscape encryption scheme. Ian and David describe how they attacked the popular Web browser and what they found out.

- [Email](#)
- [Discuss](#)

- [Print](#)
- [Rep](#)

.....
add to:

▪ [Del.icio.us](#) [Slash](#)

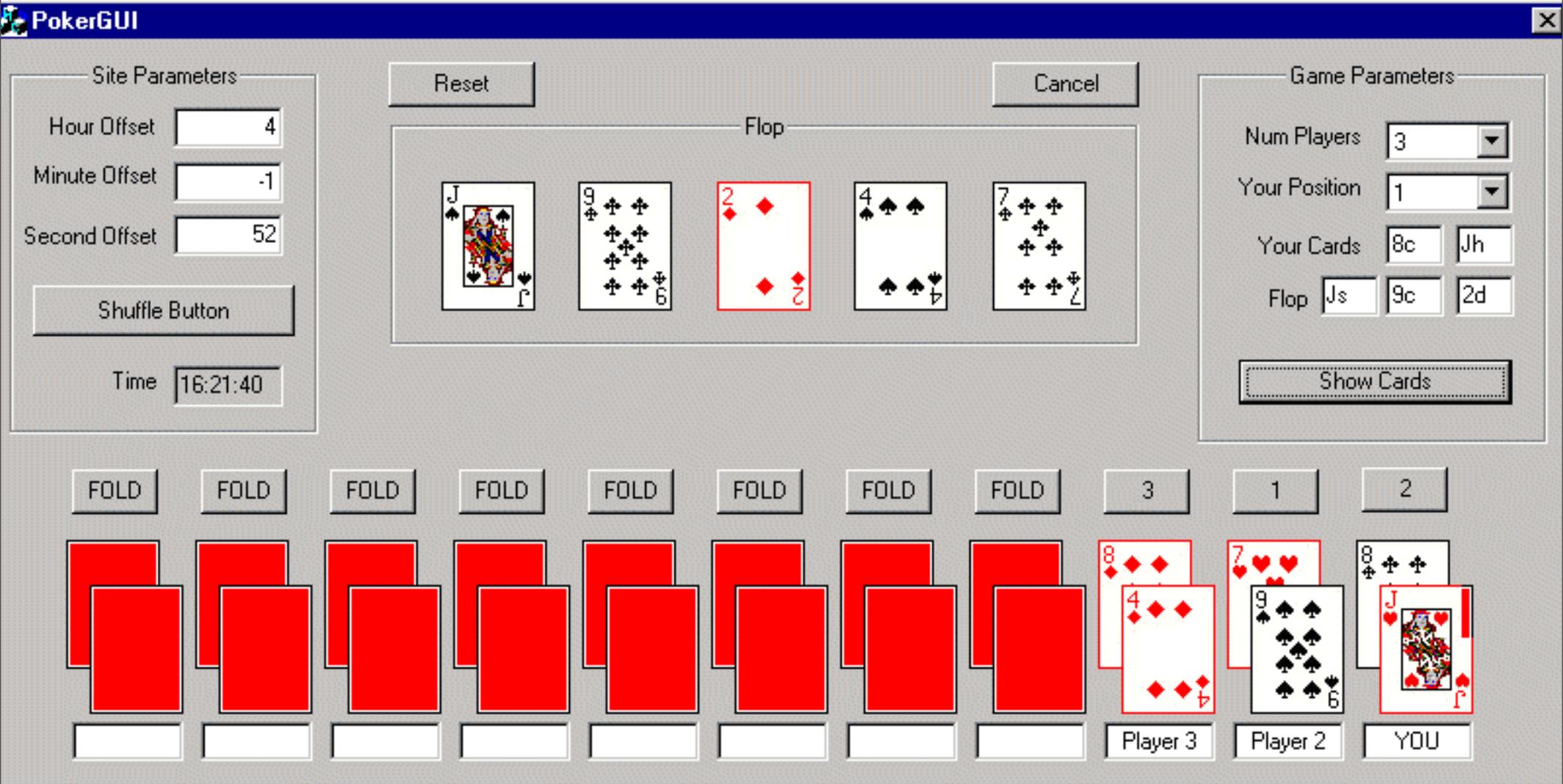
- [Digg](#)
- [Google](#)
- [Spurl](#)
- [Y!](#)
- [MyWe](#)
- [Blin](#)
- [Furl](#)

Problems in Practice

- ◆ One institution used (something like) `rand()` to generate passwords for new users
 - Given your password, you could predict the passwords of other users
- ◆ Kerberos (1988 - 1996)
 - Random number generator improperly seeded
 - Possible to trivially break into machines that rely upon Kerberos for authentication
- ◆ Online gambling websites
 - Random numbers to shuffle cards
 - Real money at stake
 - But what if poor choice of random numbers?



Images from <http://www.cigital.com/news/index.php?pg=art&artid=20>



Images from <http://www.cigital.com/news/index.php?pg=art&artid=20>



Images from <http://www.cigital.com/news/index.php?pg=art&artid=20>



Big news... CNN, etc..

Other Problems

◆ Key generation

- Ubuntu removed the randomness from SSL, creating vulnerable keys for thousands of users/servers
- Undetected for 2 years (2006-2008)

◆ Live CDs, diskless clients

- May boot up in same state every time

◆ Virtual Machines

- Save state: Opportunity for attacker to inspect the pseudorandom number generator's state
- Restart: May use same "psuedorandom" value more than once