

Software Security: Defenses & Principles

CS 161: Computer Security

Prof. David Wagner

January 27, 2014

C review session

- CSUA will offer a review session on C:
Saturday, February 1st, 2-4pm, 306 Soda
- Highly recommended!

```
void vulnerable(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

```
memcpy(void *s1, const void *s2, size_t n);
```

```
void safe(size_t len, char *data) {  
    char buf[64];  
    if (len > 64)  
        return;  
    memcpy(buf, data, len);  
}
```

```
void vulnerable(size_t len, char *data) {  
    char *buf = malloc(len+2);  
    if (buf == NULL) return;  
    memcpy(buf, data, len);  
    buf[len] = '\n';  
    buf[len+1] = '\0';  
}
```

If `len = 0xffffffff`, *allocates only 1 byte*

Broward Vote-Counting Blunder Changes Amendment Result

POSTED: 1:34 pm EST November 4, 2004

BROWARD COUNTY, Fla. -- The Broward County Elections Department has egg on its face today after a computer glitch misreported a key amendment race, according to WPLG-TV in Miami.

Amendment 4, which would allow Miami-Dade and Broward counties to hold a future election to decide if slot machines should be allowed at racetracks, was thought to be tied. But now that a computer glitch for machines counting absentee ballots has been exposed, it turns out the amendment passed.

"The software is not geared to count more than 32,000 votes in a precinct. So what happens when it gets to 32,000 is the software starts counting backward," said Broward County Mayor Ilene Lieberman.

That means that Amendment 4 passed in Broward County by more than 240,000 votes rather than the 166,000-vote margin reported Wednesday night. That increase changes the overall statewide results in what had been a neck-and-neck race, one for which recounts had been going on today. But with news of Broward's error, it's clear amendment 4 passed.



Broward County Mayor Ilene Lieberman says voting counting error is an "embarrassing mistake."

Testing for Software Security Issues

- What makes testing a program for security problems difficult?
 - We need to test for the *absence* of something
 - Security is a *negative* property!
 - “nothing bad happens, even in really unusual circumstances”
 - Normal inputs rarely stress security-vulnerable code
- How can we test more thoroughly?

Testing for Software Security Issues

- What makes testing a program for security problems difficult?
 - We need to test for the absence of something
 - Security is a negative property!
 - “nothing bad happens, even in really unusual circumstances”
 - Normal inputs rarely stress security-vulnerable code
- How can we test more thoroughly?
 - Random inputs (*fuzz testing*)



Testing for Software Security Issues

- What makes testing a program for security problems difficult?
 - We need to test for the absence of something
 - Security is a negative property!
 - “nothing bad happens, even in really unusual circumstances”
 - Normal inputs rarely stress security-vulnerable code
- How can we test more thoroughly?
 - Random inputs (*fuzz testing*)
 - Mutation
 - Spec-driven
- How do we tell when we’ve found a problem?
 - Crash or other deviant behavior; now enable expensive checks

Working Towards Secure *Systems*

- Along with securing individual components, we need to keep them up to date ...
- What's hard about **patching**?
 - Can **require restarting** production systems
 - Can **break** crucial functionality
 - Management burden:
 - It never stops (the “*patch treadmill*”) ...

IT administrators give thanks for light Patch Tuesday

07 November 2011

Microsoft is giving IT administrators a break for Thanksgiving, with only four security bulletins for this month's Patch Tuesday.

Only one of the [bulletins](#) is rated critical by Microsoft, which addresses a flaw that could result in remote code execution attacks for the newer operating systems – Windows Vista, Windows 7, and Windows 2008 Server R2.

The critical bulletin has an exploitability rating of 3, suggesting

Working Towards Secure *Systems*

- Along with securing individual components, need to keep them up to date ...
- What's hard about **patching**?
 - Can require restarting production systems
 - Can break crucial functionality
 - Management burden:
 - It never stops (the “*patch treadmill*”) ...
 - ... and can be difficult to track just what's needed where
- Other (complementary) approaches?
 - **Vulnerability scanning**: probe your systems/networks for known flaws
 - **Penetration testing** (“*pen-testing*”): **pay** someone to break into your systems ...
 - ... provided they take excellent notes about how they did it!

Extremely critical Ruby on Rails bug threatens more than 200,000 sites

Servers that run the framework are by default vulnerable to remote code attacks.

by Dan Goodin - Jan 8 2013, 4:35pm PST

HARDENING 38

Hundreds of thousands of websites are potentially at risk following the discovery of an extremely critical vulnerability in the Ruby on Rails framework that gives remote attackers the ability to execute malicious code on the underlying servers.

The bug is present in Rails versions spanning the past six years and in default configurations gives hackers a simple and reliable way to pilfer database contents, run system commands, and cause websites to crash, according to Ben Murphy, one of the developers who has confirmed the vulnerability. As of last week, the framework was used by **more than 240,000 websites**, including Github, Hulu, and Basecamp, underscoring the seriousness of the threat.

"It is quite bad," Murphy told Ars. "An attack can send a request to any Ruby on Rails sever and then execute arbitrary commands. Even though it's complex, it's reliable, so it will work 100 percent of the time."

Murphy said the bug leaves open the possibility of attacks that cause one site running rails to seek out and infect others, creating a worm that infects large swaths of the Internet. Developers with the Metasploit framework for hackers and penetration testers are in the process of creating a module that can scan the Internet for vulnerable sites and exploit the bug, said HD Moore, the CSO of Rapid7 and chief architect of Metasploit.

Maintainers of the Rails framework are urging users to update their systems as soon as possible to

Reasoning About Safety

- How can we have *confidence* that our code executes in a safe (and correct, ideally) fashion?
- Approach: build up confidence on a function-by-function / module-by-module basis
- Modularity provides *boundaries* for our reasoning:
 - *Preconditions*: what must hold for function to operate correctly
 - *Postconditions*: what holds after function completes
- These basically describe a *contract* for using the module
- Notions also apply to individual statements (what must hold for correctness; what holds after execution)
 - Stmt #1' s postcondition should logically imply Stmt #2' s precondition
 - *Invariants*: conditions that always hold at a given point in a function

```
int deref(int *p) {  
    return *p;  
}
```

Precondition?

```
/* requires: p != NULL  
           (and p a valid pointer) */  
int deref(int *p) {  
    return *p;  
}
```

Precondition: what needs to hold
for function to operate correctly


```
void *mymalloc(size_t n) {  
    void *p = malloc(n);  
    if (!p) { perror("malloc"); exit(1); }  
    return p;  
}
```

Postcondition?

```
/* ensures: retval != NULL (and a valid pointer) */  
void *mymalloc(size_t n) {  
    void *p = malloc(n);  
    if (!p) { perror("malloc"); exit(1); }  
    return p;  
}
```

Postcondition: what the function promises will hold upon its return

```
int sum(int a[], size_t n) {  
    int total = 0;  
    for (size_t i=0; i<n; i++)  
        total += a[i];  
    return total;  
}
```

Precondition?

```
int sum(int a[], size_t n) {  
    int total = 0;  
    for (size_t i=0; i<n; i++)  
        total += a[i];  
    return total;  
}
```

General correctness proof strategy for memory safety:

- (1) Identify each point of memory access
- (2) Write down precondition it requires
- (3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {  
    int total = 0;  
    for (size_t i=0; i<n; i++)  
        total += a[i];  
    return total;  
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access?

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {  
    int total = 0;  
    for (size_t i=0; i<n; i++)  
        total += a[i];  
    return total;  
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* ?? */
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires?

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: a != NULL &&
                   0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function


```
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: a != NULL &&
                   0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:

- (1) Identify each point of memory access
- (2) Write down precondition it requires
- (3) Propagate requirement up to beginning of function?

```
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: a != NULL &&
                   0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```

Let's simplify, given that `a` never changes.

```
/* requires: a != NULL */  
int sum(int a[], size_t n) {  
    int total = 0;  
    for (size_t i=0; i<n; i++)  
        /* requires: 0 <= i && i < size(a) */  
        total += a[i];  
    return total;  
}
```

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:

- (1) Identify each point of memory access
- (2) Write down precondition it requires
- (3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```

?


General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?


```
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```



General correctness proof strategy for memory safety:

- (1) Identify each point of memory access
- (2) Write down precondition it requires
- (3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```



Let's simplify given that the $0 \leq i$ part is clear.

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```



```
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

?

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```

/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}

```

?

General correctness proof strategy for memory safety:

- (1) Identify each point of memory access
- (2) Write down precondition it requires
- (3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

?

How to prove our candidate invariant?

$n \leq \text{size}(a)$ is straightforward because n never changes.

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

What about $i < n$?

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

What about $i < n$? That follows from the loop condition.

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

At this point we know the proposed invariant will always hold...

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant: a != NULL &&
           0 <= i && i < n && n <= size(a) */
        total += a[i];
    return total;
}
```

... and we're done!


```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant: a != NULL &&
           0 <= i && i < n && n <= size(a) */
        total += a[i];
    return total;
}
```

A more complicated loop might need us to use *induction*:

Base case: first entrance into loop.

Induction: show that *postcondition* of last statement of loop plus loop test condition implies invariant.

```
int sumderef(int *a[], size_t n) {  
    int total = 0;  
    for (size_t i=0; i<n; i++)  
        total += *(a[i]);  
    return total;  
}
```

```
/* requires: a != NULL &&  
    size(a) >= n &&  
    ??? */  
int sumderef(int *a[], size_t n) {  
    int total = 0;  
    for (size_t i=0; i<n; i++)  
        total += *(a[i]);  
    return total;  
}
```

```
/* requires: a != NULL &&  
    size(a) >= n &&  
    for all j in 0..n-1, a[j] != NULL */  
int sumderef(int *a[], size_t n) {  
    int total = 0;  
    for (size_t i=0; i<n; i++)  
        total += *(a[i]);  
    return total;  
}
```

```
char *tbl[N]; /* N is of type int */
```

```
int hash(char *s) {  
    int h = 17;  
    while (*s)  
        h = 257*h + (*s++) + 3;  
    return h % N;  
}
```

```
bool search(char *s) {  
    int i = hash(s);  
    return tbl[i] && (strcmp(tbl[i], s)==0);  
}
```

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */
```

```
int hash(char *s) {  
    int h = 17;  
    while (*s)  
        h = 257*h + (*s++) + 3;  
    return h % N;  
}
```

```
bool search(char *s) {  
    int i = hash(s);  
    return tbl[i] && (strcmp(tbl[i], s)==0);  
}
```

```
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
    int h = 17;                /* 0 <= h */
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */  
int hash(char *s) {  
    int h = 17; /* 0 <= h */  
    while (*s) /* 0 <= h */  
        h = 257*h + (*s++) + 3;  
    return h % N;  
}
```

```
bool search(char *s) {  
    int i = hash(s);  
    return tbl[i] && (strcmp(tbl[i], s)==0);  
}
```



```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */  
int hash(char *s) {  
    int h = 17; /* 0 <= h */  
    while (*s) /* 0 <= h */  
        h = 257*h + (*s++) + 3; /* 0 <= h */  
    return h % N;  
}
```

```
bool search(char *s) {  
    int i = hash(s);  
    return tbl[i] && (strcmp(tbl[i], s)==0);  
}
```

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */  
int hash(char *s) {  
    int h = 17; /* 0 <= h */  
    while (*s) /* 0 <= h */  
        h = 257*h + (*s++) + 3; /* 0 <= h */  
    return h % N; /* 0 <= retval < N */  
}
```

```
bool search(char *s) {  
    int i = hash(s);  
    return tbl[i] && (strcmp(tbl[i], s)==0);  
}
```

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */  
int hash(char *s) {  
    int h = 17; /* 0 <= h */  
    while (*s) /* 0 <= h */  
        h = 257*h + (*s++) + 3; /* 0 <= h */  
    return h % N; /* 0 <= retval < N */  
}
```

```
bool search(char *s) {  
    int i = hash(s);  
    return tbl[i] && (strcmp(tbl[i], s)==0);  
}
```

```
char *tbl[N];
```

```
/* ensures:  $0 \leq \text{retval} \ \&\& \ \text{retval} < N$  */
```

```
int hash(char *s) {
```

```
    int h = 17; /*  $0 \leq h$  */
```

```
    while (*s) /*  $0 \leq h$  */
```

```
        h = 257*h + (*s++) + 3; /*  $0 \leq h$  */
```

```
    return h % N; /*  $0 \leq \text{retval} < N$  */
```

```
}
```

```
bool search(char *s) {
```

```
    int i = hash(s);
```

```
    return tbl[i] && (strcmp(tbl[i], s)==0);
```

```
}
```

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */  
int hash(char *s) {  
    int h = 17; /* 0 <= h */  
    while (*s) /* 0 <= h */  
        h = 257*h + (*s++) + 3; /* 0 <= h */  
    return h % N; /* 0 <= retval < N */  
}
```

```
bool search(char *s) {  
    int i = hash(s);  
    return tbl[i] && (strcmp(tbl[i], s)==0);  
}
```

Fix?

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */  
unsigned int hash(char *s) {  
    unsigned int h = 17;          /* 0 <= h */  
    while (*s)                   /* 0 <= h */  
        h = 257*h + (*s++) + 3;  /* 0 <= h */  
    return h % N; /* 0 <= retval < N */  
}
```

```
bool search(char *s) {  
    unsigned int i = hash(s);  
    return tbl[i] && (strcmp(tbl[i], s)==0);  
}
```

Questions?

Coming Up ...

- Discussion section meets tomorrow
- Wednesday guest lecture:
Access control & OS security
- Friday guest lecture:
Malware
- Homework 0 due **Friday**
- C review session, Saturday, February 1st,
2-4pm, 306 Soda