

# Software Security: Defenses

***CS 161: Computer Security***

**Prof. Vern Paxson**

TAs: Paul Bramsen, Apoorva Dornadula,  
David Fifield, Mia Gil Epner, David Hahn, Warren He,  
Grant Ho, Frank Li, Nathan Malkin, Mitar Milutinovic,  
Rishabh Poddar, Rebecca Portnoff, Nate Wang

*<http://inst.eecs.berkeley.edu/~cs161/>*

**January 26, 2017**

# Reasoning About Memory Safety

***Memory Safety***: no accesses to *undefined* memory.

“Undefined” is with respect to the semantics of the programming language used.

“Access” can be reading / writing / executing.

# 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 (this particularly matters for loops)

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

***Precondition:*** what needs to hold for function to operate correctly.

Needs to be expressed in a way that a *person* writing code to call the function knows how to evaluate.

```
/* 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.

Likewise, expressed in a way that a person using the call in their code knows how to make use of.

```
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++)
        /* requires: a != NULL &&
                    0 <= i && i < size(a) */
        total += a[i];
    return total;
}

```

*size(X)* = number of *elements* allocated for region pointed to by X  
*size(NULL)* = 0

Gen

- (1) This is an abstract notion, *not* something built into C (like sizeof).
- (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;
}
```



The  $0 \leq i$  part is clear, so let's focus for now on the rest.



```
/* 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  $i < n$ ? 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.



```
/* 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 > 0, has 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: ??? */
```

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

```
    int h = 17;
```

```
    while (*s)
```

```
        h = 257*h + (*s++) + 3;
```

```
    return h % N;
```

```
}
```

What is the correct *postcondition* for hash()?

(a)  $0 \leq \text{retval} < N$ , (b)  $0 \leq \text{retval}$ ,

(c)  $\text{retval} < N$ , (d) none of the above.

Discuss with a partner.

```
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) /* 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);  
}
```

```
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 */  
}
```

What is the correct *postcondition* for hash()?

(a)  $0 \leq \text{retval} < N$ , (b)  $0 \leq \text{retval}$ ,

(c)  $\text{retval} < N$ , (d) none of the above.

Discuss with a partner.

```
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);
```

```
}
```

# Class Ideas: Approaches for Building Secure Software/Systems

- Write less code
  - Diminishes attack surface
  - However: may not be practical for a given system
  - Also: could lead to dense/hard-to-understand code
- Write clear documentation
  - Bring API clarity to how to correctly use libraries
  - Takes time
  - Now there are two things to keep consistent
  - Helps attackers learn about system

# Class Ideas, con't

- **Regular tests**
  - Ensure unit integrity
  - Potentially automate validation of pre/post conditions
  - But: costs time, may be incomplete, which may lead to a false sense of security
- **Use a memory-safe language like Java**
  - May diminish performance
  - May be incompatible with the installed base
  - Could increase attack surface (difficult to gauge)

# Class Ideas, con't

- **Make things crash**
  - i.e., alter code so if potential integrity issue, it leads to a crash rather than possible code execution
  - A better outcome than “pwnage”
  - Creates a denial-of-service vulnerability
- **Use open source**
  - Can enable better assessment of risk ...
  - ... though this might not be practical given code size
  - Easier for attacker to assess vulnerabilities
  - May not be practical (availability of needed functionality)
  - Possibly the open-source system becomes a juicy attacker target

# Class Ideas, con't

- Hire people to break into your systems
  - Penetration testers, aka “pen-testers”
  - Can also consider “bounties” paid to those who find vulnerabilities (need non-destructive rules-of-engagement)
  - Can gain detailed understanding of threats
  - Can cost quite bit
- Regular code reviews
  - Can be very effective
  - Costs considerable time & money



# Class Ideas, con't

- **Sandboxing**
  - Run code in isolated environments
  - A way to achieve privilege separation / minimizes TCB
  - Can require significant effort to implement
  - Problem might not naturally fit into such partitioning
  - Adds communication overhead

# Why does software have vulnerabilities?

- Programmers are humans.  
And humans make mistakes.
  - Use tools.
- Programmers often aren't security-aware.
  - Learn about common types of security flaws.
- Programming languages aren't designed well for security.
  - Use better languages (Java, Python, ...).

# 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
- How do we tell that we’ve tested enough?
  - **Hard**: but *code coverage* tools can help

# 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



Threat Level:

GREEN

YELLOW

ORANGE

RED

Storm Center Tools |

# ISC Diary

Refresh Latest Diaries

previous

next

## Oracle quietly releases Java 7u13 early

Published: 2013-02-01,

Last Updated: 2013-02-01 21:59:59 UTC

by Jim Clausing (Version: 2)



F Recommend



Tweet



+1



2 comment(s)

First off, a huge thank you to readers Ken and Paul for pointing out that Oracle has released Java 7u13. As the [CPU \(Critical Patch Update\) bulletin](#) points out, the release was originally scheduled for 19 Feb, but was moved up due to the active exploitation of one of the critical vulnerabilities in the wild. Their [Risk Matrix](#) lists 50 CVEs, 49 of which can be remotely exploitable without authentication. As Rob discussed in [his diary](#) 2 weeks ago, now is a great opportunity to determine if you really need Java installed (if not, remove it) and, if you do, take additional steps to protect the systems that do still require it. I haven't seen jusched pull this one down on my personal laptop yet, but if you have Java installed you might want to do this one manually right away. On a side note, we've had reports of folks who installed Java 7u11 and had it silently (and unexpectedly) remove Java 6 from the system thus breaking some legacy applications, so that is something else you might want to be on the lookout for if you do apply this update.



Threat Level:



Storm Center Tools |

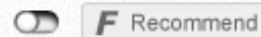
# ISC Diary

[Refresh Latest Diaries](#)

[previous](#) [next](#)

## Oracle quietly releases Java 7u13 early

Published: 2013-02-01,  
Last Updated: 2013-02-01 21:59:59 UTC  
by Jim Clausing (Version: 2)



2 comment(s)

First off, a huge thank you to readers Ken and Paul for pointing out that Oracle has released Java 7u13. As the [CPU \(Critical Patch Update\) bulletin](#) points out, the release was originally scheduled for 19 Feb, but was moved up due to the active exploitation of one of the critical vulnerabilities in the wild. Their [Risk Matrix](#) lists 50 CVEs, 49 of which can be remotely exploitable without authentication. As Rob discussed in [his diary](#) 2 weeks ago, now is a great opportunity to determine if you really need Java installed (if not, remove it) and, if you do, take additional steps to protect the systems that do still require it. I haven't seen jusched pull this one down on my personal laptop yet, but if you have Java installed you might want to do this one manually right away. On a side note, we've had reports of folks who installed Java 7u11 and had it silently (and unexpectedly) remove Java 6 from the system thus breaking some legacy applications, so that is something else you might want to be on the lookout for if you do apply this update.

# 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

*(Stopped here in lecture)*

# 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*”) ...
    - ... 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

# Some Approaches for Building Secure Software/Systems

- Run-time checks
  - Automatic bounds-checking (overhead)
  - What do you do if check fails?
- Address randomization
  - Make it hard for attacker to determine layout
  - But they might get lucky / sneaky
- Non-executable stack, heap
  - May break legacy code
  - See also *Return-Oriented Programming (ROP)*
- Monitor code for run-time misbehavior
  - E.g., illegal calling sequences
  - But again: what do you do if detected?

# Approaches for Secure Software, con't

- Program in checks / “defensive programming”
  - E.g., check for null pointer even though sure pointer will be valid
  - Relies on programmer **discipline**
- Use safe libraries
  - E.g. `strncpy`, not `strcpy`; `snprintf`, not `sprintf`
  - Relies on discipline or tools ...
- Bug-finding tools
  - Excellent resource as long as not many **false positives**
- Code review
  - Can be very effective ... but **expensive**

# Approaches for Secure Software, con't

- Use a safe language
  - E.g., Java, Python, C#
  - Safe = memory safety, strong typing, hardened libraries
  - Installed base?
  - Performance?
- Structure user input
  - Constrain how untrusted sources can interact with the system
  - Perhaps by implementing a **reference monitor**
- Contain potential damage
  - E.g., run system components in *jails* or VMs
  - Think about **privilege separation**