Code safety (cont’d)

&& Access control

CS 161: Computer Security

Prof. Raluca Ada Popa

January 23, 2018
Announcements

• Homework 1 is out, due in a week
• Dean approved class expansion, three new discussion sections, stay tuned for details
• Scrapped lecture slides available before class
  • Do not use them for answering in class
• Full lecture slides available after class
Precondition

• A precondition for a function $f()$ is an assertion that must hold about the inputs to $f$
• $f()$ is assumed to behave correctly and produce correct output as long as the precondition is met
• The caller must make sure the precondition is met
• The callee (the code inside $f()$) can assume that the precondition is met
Example

Q: What is the precondition?

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

/* requires: a != NULL && size(a) >= n &&
for all j in 0..n-1, a[j] != NULL && (sum_i *a[i]<=MAX_INT) */

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

• A postcondition on $f()$ is an assertion that holds when $f()$ returns
• The caller of $f()$ can assume that the postcondition holds
• $f()$ must make sure the postcondition holds
Example

Q: What is the postcondition?

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

/* ensures: retval != NULL && retval points to n bytes of memory */
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) {
        perror("Out of memory");
        exit(1);
    }
    return p;
}
Specification vs implementation

• A function has a specification = precondition+postcondition
• And an implementation that should meet the specification: for all inputs satisfying the precondition, it must satisfy the postcondition.
Reasoning about code

To prove that a function whose inputs satisfy the precondition, matches the postcondition, you can:

• Write down a precondition and postcondition for every line of code, and prove this
  • Each statement’s postcondition must imply the precondition of the next statement. This is an invariant that is true at any point in time.

• Final postcondition is the postcondition for the function
/* requires: n >= 0 */
void binpr(int n) {
    char digits[] = "0123456789";  /* n >= 0 */
    while (n != 0) {  /* n>0 */
        int d = n % 10;  /* 0<=d && d < 10 && n > 0*/
        putchar(digits[d]);
        n = n / 10;  /* 0<=d && d<10 && n>=0*/
    }
    putchar('0');
}
int sumderefer(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}

What is the precondition?
What is the precondition?

```c
/* 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;
}
```
What is the precondition?

```c
/* requires: a != NULL &&
   size(a) >= n &&
   for all j in 0..n-1, a[j] != NULL
   (&& sum *(a[i]) <= MAXINT )*/

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);
}
```c
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 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.
```
char *tbl[N];

/* ensures: \(0 \leq retval && retval < N\) */
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 */
text 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 */
}

Is the postcondition correct?
(a) Yes, (b) 0 <= retval is correct,
(c) retval < N is correct, (d) both are wrong.
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 <= 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];

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 <= retval < N, (b) 0 <= retval,
(c) 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);
}
Access Control and OS Security
Types of Security Properties

• Confidentiality
• Integrity
• Availability
Access Control

- Some resources (files, web pages, ...) are sensitive.
- How do we limit who can access them?
- This is called the *access control* problem
Access Control Fundamentals

• *Subject* = a user, process, ...
  (someone who is accessing resources)

• *Object* = a file, device, web page, ...
  (a resource that can be accessed)

• *Policy* = the restrictions we’ll enforce

• \( access(S, O) = true \)
  if subject \( S \) is allowed to access object \( O \)
Example

• $\text{access}(\text{Alice}, \text{Alice’s wall}) = \text{true}$
  $\text{access}(\text{Alice}, \text{Bob’s wall}) = \text{true}$
  $\text{access}(\text{Alice}, \text{Charlie’s wall}) = \text{false}$

• $\text{access}(\text{raluca}, /\text{home/cs161/gradebook}) = \text{true}$
  $\text{access}(\text{Alice}, /\text{home/cs161/gradebook}) = \text{false}$
Access Control Matrix

• $\text{access}(S, O) = \text{true}$ if subject $S$ is allowed to access object $O$

<table>
<thead>
<tr>
<th></th>
<th>Alice’s wall</th>
<th>Bob’s wall</th>
<th>Charlie’s wall</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Permissions

- We can have finer-grained permissions, e.g., read, write, execute.

- $access(\text{raluca, /cs161/grades/alice}) = \{\text{read, write}\}$
  $access(\text{alice, /cs161/grades/alice}) = \{\text{read}\}$
  $access(\text{bob, /cs161/grades/alice}) = \{\}$

<table>
<thead>
<tr>
<th></th>
<th>/cs161/grades/alice</th>
</tr>
</thead>
<tbody>
<tr>
<td>daw</td>
<td>read, write</td>
</tr>
<tr>
<td>alice</td>
<td>read</td>
</tr>
<tr>
<td>bob</td>
<td>-</td>
</tr>
</tbody>
</table>
Access Control

• Authorization: who *should* be able to perform which actions
• Authentication: verifying who is requesting the action
Access Control

• Authorization: who *should* be able to perform which actions
• Authentication: verifying who is requesting the action
• Audit: a log of all actions, attributed to a particular principal
• Accountability: hold people legally responsible for actions they take.
Web security

• Let’s talk about how this applies to web security...
How should we implement access control policy?
Option 1: Integrated Access Control

Record username.
Check policy at each place in code that accesses data.
Record username. Database checks policy for each data access.
Option 1: Integrated Access Control

Record username.
Check policy at each place in code that accesses data.

Option 2: Centralized Enforcement

Record username.
Database checks policy for each data access.

Which option would you pick? Discuss.
Analysis

• Centralized enforcement might be less prone to error
  • All accesses are vectored through a central chokepoint, which checks access
  • If you have to add checks to each piece of code that accesses data, it’s easy to forget a check (and app will work fine in normal usage, until someone tries to access something they shouldn’t)

• Integrated checks might be more flexible
Complete mediation

• The principle: complete mediation

• Ensure that all access to data is mediated by something that checks access control policy.
  • In other words: the access checks can’t be bypassed
If you don’t have complete mediation, your access control will fail
Reference monitor

• A reference monitor is responsible for mediating all access to data

• Subject cannot access data directly; operations must go through the reference monitor, which checks whether they’re OK
Criteria for a reference monitor

Ideally, a reference monitor should be:

• Unbypassable: all accesses go through the reference monitor

• Tamper-resistant: attacker cannot subvert or take control of the reference monitor (e.g., no code injection)

• Verifiable: reference monitor should be simple enough that it’s unlikely to have bugs
Example: OS memory protection

- All memory accesses are mediated by memory controller, which enforces limits on what memory each process can access
More broadly, the trusted computing base (TCB) is the subset of the system that has to be correct, for some security goal to be achieved.

- Example: the TCB for enforcing file access permissions includes the OS kernel and filesystem drivers.

- Ideally, TCBs should be unbypassable, tamper-resistant, and verifiable.
Robustness

• Security bugs are a fact of life

• How can we use access control to improve the security of software, so security bugs are less likely to be catastrophic?
Privilege separation

- How can we improve the security of software, so security bugs are less likely to be catastrophic?

- Answer: privilege separation. Give each module only the privilege it needs.
  
  - In particular, architect the software so it has a separate, small TCB.
  - Then any bugs outside the TCB will not be catastrophic.
Naïve web browser

“Drive-by malware”: malicious web page exploits a browser bug to read/write local files or infect them with a virus
The Chrome browser

Two pieces: rendering engine and browser kernel

Rendering engine:
- Interprets HTML and turns it into bitmap image to display on screen
- Most bugs are here so it is ran inside a sandbox
- Sandbox isolates the engine from the rest of the system, including files, and allows only narrow API to the outside

Browser kernel:
- Mediates all access to the file system
The Chrome browser

Goal: prevent “drive-by malware”, where a malicious web page exploits a browser bug to read/write local files or infect them with a virus.
The Chrome browser architecture aims to prevent an attacker from reading or writing files to an entire instance of the rendering engine, even if the attacker has the following abilities:

- **Mimic the user:** attacker can convince the user's browser to render malicious content.
- **Mimic the web:** attacker can control the domain name and route traffic.
- **Mimic the kernel:** attacker can execute arbitrary code.

The security architecture is designed to mitigate the most severe vulnerabilities, namely those vulnerabilities that let an attacker execute arbitrary code. If an attacker exploits such a vulnerability in the rendering engine, the architecture extracts most of the security benefits of sandboxing while maintaining performance.

The rendering engine contains a large number of vulnerabilities, with 70% of critical browser vulnerabilities allowing an attacker to execute arbitrary code. Chromium's architecture mitigates approximately 4% (38 of 54) of all disclosed vulnerabilities that allow arbitrary code execution. These account for 70% of the 87 rendering engine vulnerabilities analyzed, compared to 4% (87 of 129) of the total disclosed vulnerabilities.

In order to characterize the security properties of Chromium's architecture, we define a threat model by enumerating attacker abilities and goals. The security architecture aims to prevent an attacker with these abilities from reaching these goals. We can use this threat model to evaluate how effectively Chromium's architecture protects users from attack.
<table>
<thead>
<tr>
<th>Browser</th>
<th>Known unpatched vulnerabilities</th>
<th>SecurityFocus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Extremely critical (number / oldest)</td>
<td>Highly critical (number / oldest)</td>
</tr>
<tr>
<td>Internet Explorer 6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Internet Explorer 7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Internet Explorer 8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Internet Explorer 9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Firefox 3.6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Firefox 38</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Google Chrome 42</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Opera 11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Safari 5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
BE GOOD WITH YOUR MONEY FROM THE BIG PICTURE TO THE DETAILS THAT MATTER

Effortlessly manage your cash flow, budgets and bills from one place.

All-in-one? Done
From money and budgeting to customized tips and more—get a clear view of your total financial life.

Budgets? You betcha
Effortlessly create budgets that are easy to stick to. We even make a few for you.

Credit? Checked
Find out yours and learn how you can improve it. It’s totally free.
Discuss with a partner

• How would you architect mint.com to reduce the likelihood of a catastrophic security breach?
  • E.g., where attacker steals all users’ stored passwords or empties out all their bank accounts overnight
Summary

• Access control is a key part of security.

• Privilege separation makes systems more robust: it helps reduce the impact of security bugs in your code.

• Architect your system to make the TCB unbypassable, tamper-resistant, and verifiable (small).
More principles for designing more secure software
TRTL-30
“Security is economics.”
What does this program do?
What can this program do?

Can it delete all of your files?

YES. Why?
“Least privilege.”
Touchstones for *Least Privilege*

- When assessing the security of a system’s design, identify the *Trusted Computing Base* (**TCB**).
  - What components does security rely upon?

- Security requires that the TCB:
  - Is *correct*
  - Is *complete* (can’t be bypassed)
  - Is itself *secure* (can’t be tampered with)

- Best way to be assured of correctness and its security?
  - **KISS** = *Keep It Simple, Stupid!*
  - Generally, *Simple* = **Small**

- One powerful design approach: *privilege separation*
  - Isolate privileged operations to as small a component as possible
  - (See lecture notes for more discussion)
Check for Understanding

• We’ve seen that PC platforms grant applications a lot of privileges

• Quiz: Name a platform that does a better job of least privilege
“Ensure complete mediation.”
Ensuring Complete Mediation

• To secure access to some capability/resource, construct a *reference monitor*

• Single point through which all access must occur
  • E.g.: a network firewall

• Desired properties:
  • Un-bypassable (“complete mediation”)
  • Tamper-proof (is itself secure)
  • Verifiable (correct)
  • (Note, just restatements of what we want for TCBs)

• One subtle form of reference monitor flaw concerns *race conditions* ...
procedure withdrawal(w)
   // contact central server to get balance
   1. let b := balance

   2. if b < w, abort

Balance could have decreased at this point due to another action

   // contact server to set balance
   3. set balance := b - w

   4. dispense $w to user

TOCTTOU = Time of Check To Time of Use
public void buyItem(Account buyer, Item item) {
    if (item.cost > buyer.balance)
        return;
    buyer.possessions.put(item);
    buyer.possessionsUpdated();
    buyer.balance -= item.cost;
    buyer.balanceUpdated();
}
NO LONE ZONE
SAC TWO MAN POLICY
MANDATORY

CAUTION
DO NOT KEY RTNX IN L/D
EXCEPT IN CASE OF EMERGENCY MUST BE AT
LEAST 5FT FROM MSL.
“Division of trust.”
- reduce the trust in each party