Lecture 4: Memory Safety

https://cs161.org
Announcements

• Homework 0 due Friday.
• Expect Homework 1 to be released later this week.
Buffer Overflows
4

- user stack
- shared libraries
- run time heap
- static data segment
- text segment (program)
- unused

- arguments
- return address
- saved frame pointer
- exception handlers
- local variables
- callee saved registers

To previous saved frame pointer
To the point at which this function was called
```c
void safe() {
    char buf[64];
    ...
    fgets(buf, 64, stdin);
    ...
}
```
```c
void safer() {
    char buf[64];
    ...
    fgets(buf, sizeof(buf), stdin);
    ...
}
```
void vulnerable(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}

Assume these are both under the control of an attacker.

size_t is **unsigned**:
What happens if len == -1?

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 f(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';
}

Vulnerable!
If len = 0xffffffff, allocates only 1 byte

Is it safe? Talk to your partner.
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.
Memory Safety
void vulnerable() {
    char buf[64];
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
printf("you scored %d\n", score);
printf("you scored %d\n", score);
printf("a %s costs $%d\n", item, price);
printf("a \% costs \$%\n", item, price);

\0 \n d %
$ s t
s o c
s % a

0x8048464
Fun With `printf` format strings...

```c
printf("100% dude.");
```

Format argument is missing!
printf("100% dude!");
More Fun With `printf` format strings...

```
printf("100% dude!");
⇒ prints value 4 bytes above retaddr as integer
printf("100% sir!");
⇒ prints bytes pointed to by that stack entry
    up through first NUL
printf("%d %d %d %d ..." untitled
⇒ prints series of stack entries as integers
printf("%d %s");
⇒ prints value 8 bytes above retaddr plus bytes
    pointed to by preceding stack entry
printf("100% nuke’m!");
```

What does the `%n` format do??
printf("item %d:%n $%d\n", item_num, &colon_offset, price);
    return colon_offset;
}

report_cost(3, 22) prints "item 3: $22"
    and returns the value 7

report_cost(987, 5) prints "item 987: $5"
    and returns the value 9

%n writes the number of characters printed so far into the corresponding format argument.
Fun With `printf` format strings...

```c
printf("100% dude!");
⇒ prints value 4 bytes above retaddr as integer

printf("100% sir!");
⇒ prints bytes pointed to by that stack entry up through first NUL

printf("%d %d %d %d ...");  
⇒ prints series of stack entries as integers

printf("%d %s");
⇒ prints value 8 bytes above retaddr plus bytes pointed to by preceding stack entry

printf("100% nuke’m!");
⇒ writes the value 3 to the address pointed to by stack entry
```
void safe() {
    char buf[64];
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf("%s", buf);
}
It isn't just the stack...

- Control flow attacks require that the attacker overwrite a piece of memory that contains a pointer for future code execution
  - The return address on the stack is just the easiest target
- You can cause plenty of mayhem overwriting memory in the heap...
  - And it is made easier when targeting C++
- Allows alternate ways to hijack control flow of the program
class Foo {
    int i, j, k;
    public virtual void bar() { ... }
    public virtual void baz() { ... }
    ....

    | vtable ptr (class Foo) |
    |------------------------|
    | i                      |
    | j                      |
    | k                      |

    | ptr to Foo::bar        |
    |------------------------|
    | ptr to Foo::baz        |
    | ...                    |
    | ...                    |
A Few Exploit Techniques

• If you can overwrite a vtable pointer…
  • It is effectively the same as overwriting the return address pointer on the stack:
    When the function gets invoked the control flow is hijacked to point to the attacker’s code
  • The only difference is that instead of overwriting with a pointer you overwrite it with a pointer to a table of pointers...

• Heap Overflow:
  • A buffer in the heap is not checked:
    Attacker writes beyond and overwrites the vtable pointer of the next object in memory

• Use-after-free:
  • An object is deallocated too early:
    Attacker writes new data in a newly reallocated block that overwrites the vtable pointer
  • Object is then invoked
Magic Numbers & Exploitation…

- Exploits can often be **very** brittle
  - You see this on your Project 1: Your ./egg will not work on someone else’s VM because the memory layout is different

- Making an exploit robust is an art unto itself
  - EXTRABACON is an NSA exploit for Cisco ASA “Adaptive Appliances”
  - It had an exploitable stack-overflow vulnerability in the SNMP read operation
  - But actual exploitation required two steps:
    - Query for the particular version (with an SMTP read)
    - Select the proper set of magic numbers for that version
A hack that helps: NOOP sled...

• Don't just overwrite the pointer and then provide the code you want to execute...

• Instead, write a large number of NOOP operations
  • Instructions that do nothing

• Now if you are a little off, it doesn't matter
  • Since if you are close enough, control flow will land in the sled and start running...
ETERNALBLUE

- ETERNALBLUE is another NSA exploit
- Stolen by the same group ("ShadowBrokers")
- Remote exploit for Windows through SMBv1
- Eventually it was very robust...
- But initially it was jokingly called ETERNALBLUE crash Windows computers more reliably than e
Memory Safety

- Memory Safety: No accesses to undefined memory
  - "Undefined" is with respect to the semantics of the programming language
  - Read Access: attacker can read memory that he isn't supposed to
  - Write Access: attacker can write memory that she isn't supposed to
  - Execute Access: transfer control flow to memory they aren’t supposed to
- Spatial safety: No access out of bounds
- Temporal safety: No access before or after lifetime of object
Below is a brief listing of the weaknesses in the 2019 CWE Top 25, including the overall score of each.

<table>
<thead>
<tr>
<th>Rank</th>
<th>ID</th>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>CWE-119</td>
<td>Improper Restriction of Operations within the Bounds of a Memory Buffer</td>
<td>75.56</td>
</tr>
<tr>
<td>[2]</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
<td>45.69</td>
</tr>
<tr>
<td>[3]</td>
<td>CWE-20</td>
<td>Improper Input Validation</td>
<td>43.61</td>
</tr>
<tr>
<td>[6]</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
<td>24.54</td>
</tr>
<tr>
<td>[7]</td>
<td>CWE-416</td>
<td>Use After Free</td>
<td>17.94</td>
</tr>
<tr>
<td>[8]</td>
<td>CWE-190</td>
<td>Integer Overflow or Wraparound</td>
<td>17.35</td>
</tr>
<tr>
<td>[9]</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
<td>15.54</td>
</tr>
<tr>
<td>[10]</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')</td>
<td>14.10</td>
</tr>
<tr>
<td>[11]</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
<td>11.47</td>
</tr>
<tr>
<td>[12]</td>
<td>CWE-787</td>
<td>Out-of-bounds Write</td>
<td>11.08</td>
</tr>
<tr>
<td>[14]</td>
<td>CWE-476</td>
<td>NULL Pointer Dereference</td>
<td>9.74</td>
</tr>
<tr>
<td>[16]</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
<td>5.50</td>
</tr>
<tr>
<td>[17]</td>
<td>CWE-611</td>
<td>Improper Restriction of XML External Entity Reference</td>
<td>5.48</td>
</tr>
</tbody>
</table>
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)
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.

Needs to be expressed in a way that a *person* writing code to call the function knows how to evaluate.
```c
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

*Postcondition?*
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
```c
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

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

This is an abstract notion, not something built into C (like sizeof).
```c
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?
Let’s simplify, given that \texttt{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?
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?

```c
/* 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?

```c
/* 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.

```c
/* 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: 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?
How to prove our candidate invariant?

\( n \leq \text{size}(a) \) is straightforward because \( n \) never changes.
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++) {
        total += a[i];
    }
    return total;
}

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

... and we’re done!
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        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;
}

This may still be memory safe
but it can still have undefined behavior!
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 <= retval < N, (b) 0 <= retval, (c) retval < N, (d) none of the above.
Discuss with a partner.
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: ??? */
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: ??? */
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: ??? */
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 <= 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; /* 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;
}

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 <= 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);
}
Memory safe languages

• Do you honestly think a human is going to go through this process for all their code?
  • Because that is what it takes to prevent undefined memory behavior in C or C++

• Instead, use a safe language:
  • Turns "undefined" memory references into an immediate exception or program termination
  • Now you simply don't have to worry about buffer overflows and similar vulnerabilities

• Plenty to chose from:
  • Python, Java, Go, Rust, Swift, C#, … Pretty much everything other than C/C++/Objective C