itwbennett writes:

Cisco Systems has released a new batch of security patches for flaws affecting a wide range of products, including for a critical vulnerability in its RV220W wireless network security firewalls. The RV220W vulnerability stems from insufficient input validation of HTTP requests sent to the firewall's Web-based management interface. This could allow remote unauthenticated attackers to send HTTP requests with SQL code in their headers that would bypass the authentication on the targeted devices and give attackers administrative privileges.
Homework 1 will be posted after class today

Due: Feb 22

Should be fun!

TAs can help with setup

Use Piazza as first step if you need help getting setup
Today

* More vulnerabilities from Monday
* Program analyzers
* Software fuzzing
* Static + dynamic analysis
#include <stdio.h>
#include <string.h>

void greeting( char* temp1, char* temp2 )
{
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}

int main(int argc, char* argv[])
{
    greeting( argv[1], argv[2] );
    printf( "Bye %s %s\n", argv[1], argv[2] );
}
heap overflows

Low memory addresses

High memory addresses

attacker buffer
attacker ptr

heap overflows
void printf(const char* format, ...) 
printf("Hi %s %s\n", argv[1], argv[2]);

void main(int argc, char* argv[]) {
    printf(argv[1]);
}

What if? argv[1] = "%s%s%s%s%s%s%s%s%s%s"

What if argv[1] = "%p%p%p%p%p%p%p%p%p%p%p"

Adversary-controlled format string gives all sorts of control
Can do control flow hijacking directly
What type of vulnerability is this?

```
main(int argc, char* argv[]) {
    char* b1;
    char* b2;
    char* b3;
    b1 = (char*)malloc(248);
    b2 = (char*)malloc(248);
    free(b1);
    free(b2);
    b3 = (char*)malloc(512);
    strncpy(b3, argv[1], 511);
    free(b2);
    free(b3);
}
```

Double-free vulnerability
Example malloc/free implementation

malloc control data stored alongside process data
Uses a doubly-linked list to manage heap

When a free(ptr) occurs:
- Try to consolidate with prev if empty
- Try to consolidate to next if empty
- Otherwise: mark as empty
double-free example
double-free example

```c
l=null

malloc()
- Searches left-to-right for free chunk
- Modifies pointers

b1 = malloc(BUF_SIZE1);
```

Diagram:
- A node `l` with `null` as its left (l) and right (r) children.
- The `malloc()` function searches left-to-right for a free chunk and modifies pointers.
- An example allocation `b1 = malloc(BUF_SIZE1);` is shown.
double-free example

l=null  right  0

data1

left  right  0

data2

left  right  1

empty

left  right  1

b1 = malloc(BUF_SIZE1);
b2 = malloc(BUF_SIZE2);

malloc()
- Searches left-to-right for free chunk
- Modifies pointers
double-free example

malloc()
- Searches left-to-right for free chunk
- Modifies pointers

```c
b1 = malloc(BUF_SIZE1);
b2 = malloc(BUF_SIZE2);
free(b1);
free(b2);
```
double-free example

malloc()
- Searches left-to-right for free chunk
- Modifies pointers

b1 = malloc(BUF_SIZE1);
b2 = malloc(BUF_SIZE2);
free(b1);
free(b2);
b3 = malloc(BUF_SIZE1 + BUF_SIZE2);
strncpy(b3, argv[1], BUF_SZ1+BUF_SZ2-1);
free(b2);
free(b3);

(b2-8)->left->right = (b2-8)->right
(b2-8)->right->left = (b2-8)->left
exercise: think-pair-share

Adversary controls argv[1]: what can she do?
Write 4 bytes anywhere in memory using data2.left and data2.right

malloc()
- Searches left-to-right for free chunk
- Modifies pointers

b1 = malloc(BUF_SIZE1);
b2 = malloc(BUF_SIZE2);
free(b1);

strncpy(b3, argv[1], BUF_SZ1+BUF_SZ2-1);
free(b2);
free(b3);

(b2-8)->left->right = (b2-8)->right
(b2-8)->right->left = (b2-8)->left
Manual analysis can be effective, but time consuming. Security analysts use tools to augment analysis.
program analyzers
<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soundness</td>
<td>Any error reported by the analyzer is an actual error in the program.</td>
</tr>
<tr>
<td></td>
<td>&quot;Sound for reporting errors&quot;</td>
</tr>
<tr>
<td>Completeness</td>
<td>The analyzer finds all error, if any, present in the program.</td>
</tr>
<tr>
<td></td>
<td>&quot;Complete for reporting errors&quot;</td>
</tr>
</tbody>
</table>
* True positive: Actual error

* False positive: Error reported, not a real error

* False negative: Missed an error; not reported

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>rlib.c:324</td>
</tr>
<tr>
<td>2</td>
<td>buffer</td>
<td>splib.c:98</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>mem.c:30</td>
</tr>
<tr>
<td>4</td>
<td>stack oflow</td>
<td>fst.c:1092</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>app.h:431</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>fst.c:421</td>
</tr>
<tr>
<td>Sound</td>
<td>Complete</td>
<td>Incomplete</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td>Reports all errors</td>
<td>May not report all errors</td>
</tr>
<tr>
<td></td>
<td>Reports no false alarms</td>
<td>Reports no false alarms</td>
</tr>
<tr>
<td></td>
<td>No false positives</td>
<td>No false positives</td>
</tr>
<tr>
<td></td>
<td>No false negatives</td>
<td>False negatives</td>
</tr>
<tr>
<td>Undecidable</td>
<td><strong>Undecidable</strong></td>
<td>Decidable</td>
</tr>
</tbody>
</table>

- **Undecidable**
  - Reports all errors
  - May report false alarms
  - False positives
  - No false negatives

- **Decidable**
  - Reports all errors
  - May report false alarms
  - False positives
  - No false negatives

- **Decidable**
  - May not report all errors
  - Reports no false alarms
  - False negatives
  - False positives

*soundness/completeness over error reports*
* Static analysis
  / examines code (source or binary)
  / analyze program without running the program

* Dynamic analysis
  / runs programs on test inputs
  / examines run-time behavior
  / may instrument analysis target
program analyzers

* Static analyzers
  /grep, lint, gcc -Wall [source code scanners]
    // Look for suspicious code/patterns (strcpy, gets)
  /Klee, Fie [symbolic execution]
  /Coverity

* Dynamic analyzers
  /assert()
  /Valgrind, Purify
  /Fuzzers

Soundness and completeness of these tools?
fuzzing

Feed randomized inputs into a program.
Does this work? Why would it work?
What are the limitations?
black box fuzzing

Normal input → Program → output(s)

Mutated input → Program → output(s)

Mutated input n → Program → Program crash
Normal input

Program

Crash tells you:
Something unexpected happened

Crash gives:
Stack at time of crash
Possible location for vulnerabilities
Input to repeat the experiment

Use this to start manual analysis

Program crash

black box fuzzing
* When do you stop fuzzing?

* Can use code-coverage to determine how effective your fuzzing is
  / # of LOCs executed
  / # of basic blocks reached
  / # of paths follows
  / # of conditionals followed
* Heap overflows, format string vulnerabilities

* Double-free vulnerabilities

* Program analyzers
  / Soundness + completeness
  / Static + dynamic analyzers
  / Fuzzing