mask.of.sanity writes

"A security researcher has demonstrated a series of attacks that are capable of disabling touch tone and voice activated phone systems, forcing them to disclose sensitive information. The commands can be keyed in using touchtones or even using the human voice. In one test, a phone system run by an unnamed Indian bank had dumped customer PINs. In another, a buffer overflow was triggered against a back-end database. Other attacks can be used to crash phone systems outright."
Finding vulnerabilities

CS642: Computer Security

Professor Ristenpart

http://www.cs.wisc.edu/~rist/

rist at cs dot wisc dot edu
Finding vulnerabilities

Manual analysis

Simple example: double free

Fuzzing tools

Static analysis, dynamic analysis

...
Hackers use People, Processes and Technology to obtain a singular goal: Information dominance

From “How Hackers Look for Bugs”, Dave Aitel
Take a sample product X and attack it remotely

From “How Hackers Look for Bugs”, Dave Aitel
Manual analysis

• You get a binary or the source code
• You find vulnerabilities
IDA Pro
IDA Pro
What type of vulnerability might this be?

```c
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
Double-free vulnerabilities

Can corrupt the state of the heap management

Say we use a simple doubly-linked list malloc implementation with control information stored alongside data

Chunk has:
1) left ptr (to previous chunk)
2) right ptr (to next chunk)
3) free bit which denotes if chunk is free
   this reuses low bit of right ptr
   because we will align chunks
4) user data
malloc()
- search left-to-right for free chunk
- modify pointers
malloc()
- search left-to-right for free chunk
- modify pointers

b1 = malloc( BUF_SIZE1 );
malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors
malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors

free( b1 )
malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors

free( b1 )
free( b2 )
NULL

chunk1.right 0

data1

chunk3.left

chunk3.right 1

data2

chunk2.left

NULL 1

`malloc()`
- search left-to-right for free chunk
- modify pointers

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

`free()`
- Consolidate with free neighbors

free( b1 )
free( b2 )
b3 = malloc( BUF_SIZE1 + BUF_SIZE2 )
malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors

free( b1 )
free( b2 )
b3 = malloc( BUF_SIZE1 + BUF_SIZE2 )
strncpy( b3, argv[1], BUF_SIZE1+BUF_SIZE2-1 )
malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors

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

With a clever argv[1]:
write a 4-byte word to an arbitrary location in memory

(b2 - 8)->left->right = (b2-8)->right
(b2 - 8)->right->left = (b2-8)->left
What type of vulnerability might this be?

This is very simple example. Manual analysis is very time consuming.

Security analysts use a variety of tools to augment manual analysis
Security tools

• Testing tools for helping find security-critical bugs
  – Scanners
  – Taint trackers
  – Fuzzers
    • “dumb”, “smart”, whitebox
  – Static analysis tools
• Static analysis vs. dynamic analysis
Program analyzers

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer overfl</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack overfl</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>10,921</td>
</tr>
</tbody>
</table>

Program analyzers potentially report many warnings, but may emit false alarms. Analyze large code bases.

Slide credit: Prof Mitchell Stanford’s CS 155
Program analyzers

• Static analysis
  – Do not execute program

• Dynamic analysis
  – Execute program on test cases
## Soundness, Completeness

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soundness</td>
<td>If the program contains an error, the analysis will report a warning. “Sound for reporting correctness”</td>
</tr>
<tr>
<td>Completeness</td>
<td>If the analysis reports an error, the program will contain an error. “Complete for reporting correctness”</td>
</tr>
</tbody>
</table>

---

*Slide credit: Prof Mitchell Stanford’s CS 155*
<table>
<thead>
<tr>
<th></th>
<th>Complete</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td>Reports all errors</td>
<td>Reports all errors</td>
</tr>
<tr>
<td></td>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td></td>
<td>No false positives</td>
<td>No false negatives</td>
</tr>
<tr>
<td></td>
<td>No false negatives</td>
<td>False positives</td>
</tr>
<tr>
<td><strong>Undecidable</strong></td>
<td></td>
<td><strong>Decidable</strong></td>
</tr>
<tr>
<td><strong>Unsound</strong></td>
<td>May not report all errors</td>
<td>May not report all errors</td>
</tr>
<tr>
<td></td>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td></td>
<td>False positives</td>
<td>False negatives</td>
</tr>
<tr>
<td></td>
<td>No false negatives</td>
<td>False positives</td>
</tr>
<tr>
<td><strong>Decidable</strong></td>
<td></td>
<td><strong>Decidable</strong></td>
</tr>
</tbody>
</table>

*Slide credit: Prof Mitchell Stanford’s CS 155*
Source code scanners

Look at source code, flag suspicious constructs

```
... strcpy( ptr1, ptr2 );
...
```

Warning: Don’t use strcpy

Simplest example: grep
Lint is early example
RATS  (Rough auditing tool for security)
ITS4   (It’s the Software Stupid Security Scanner)

Circa 1990’s technology: *shouldn’t* work for reasonable modern codebases
Taint tracking

Track information flow from user input to it’s use

Can be either static or dynamic

Useful to augment manual testing

```c
strcpy(buf, argv[1]);
```
Fuzzing

“The term first originates from a class project at the University of Wisconsin 1988 although similar techniques have been used in the field of quality assurance, where they are referred to as robustness testing, syntax testing or negative testing.”
Wikipedia
http://en.wikipedia.org/wiki/Fuzz_testing

Choose a bunch of inputs
See if they cause program to misbehave
Example of dynamic analysis
Black-box fuzz testing

Normal input → Program → Output(s)

Mutated input → Program → Output(s)

Mutated input 2 → Program → Program crash
Black-box fuzz testing

x=10
y=20
z=30

Program

output(s)

x = random
y = random
z = random

Program

output(s)

Achieving code coverage can be very difficult
Black-box fuzz testing

if (x == 11) {
    //vulnerable code
}

output(s)

Achieving code coverage can be very difficult

If x is 32 bits, then probability of crashing is at most?
Fuzzing is a lot about code coverage

• Code coverage defined in many ways
  – # of basic blocks reached
  – # of paths followed
  – # of conditionals followed

• Mutation based
  – Start with known-good examples
  – Mutate them to new test cases
    • heuristics: increase string lengths (AAAAAAAAAAA...)
    • randomly change items

• Generative
  – Start with specification of protocol, file format
  – Build test case files from it
    • Rarely used parts of spec
Example from Miller slides

Multiplayer game
Fuzz for remote exploits
• Capture packets during normal use
• Replace some packet contents with random values
• Send to game, determine code coverage

Initial: 614 out of 36183 basic blocks

One big switch statement controlled by third byte of packet
Update fuzz rules to exhaust the values of this third byte

Improves coverage by 4x.
Repeat several times to improve coverage.
Heap overflow found.

From Wikipedia:
Symbolic execution

void myfree(int* p, int x) {
    int y = 0;
    free(p);
    y = 10*x;
    if( y > 20 ) free(p);
}

Let $x'$ be symbolic variable for $x$

$y' = 2x'$ is new symbolic variable

$y' > 20$ is constraint on reaching second free()

$2x' > 20$ must hold for input $x$ to cause double free

• Technique for statically analyzing code paths and finding inputs
• Associate to each input variable a special symbol
  – called symbolic variable
• Simulate execution symbolically
  – Update symbolic variable’s value appropriately
  – Conditionals add constraints on possible values
• Cast constraints as satisfiability, and use SAT solver to find inputs
White-box fuzz testing

• Start with real input and do static analysis
  – Symbolic execution of program
  – Gather constraints (control flow) along way
  – Systematically negate constraints backwards
  – Eventually this yields a new input

• Repeat

Godefroid, Levin, Molnar. “Automated Whitebox Fuzz Testing”
Symbolic execution + fuzzing

void top(char input[4]) {
    int cnt=0;
    if (input[0] == ’b’) cnt++;
    if (input[1] == ’a’) cnt++;
    if (input[2] == ’d’) cnt++;
    if (input[3] == ’!’) cnt++;
    if (cnt >= 3) abort(); // error
}

Start with some input.
Run program for real & symbolically
Say input = “good”

Example from Godefroid et al.

i0 != ’b’
i1 != ’a’
i2 != ’d’
i3 != ’!’

This gives set of constraints on input
Negate them one at a time to generate a new input that explores new path

Example
i0 != ’b’ and i1 != ’a’ and i2 != ’d’ and i3 = ’!’
input would be ``goo!”

Repeat with new input
Figure 2. Search space for the example of Figure 1 with the value of the variable `cnt` at the end of each run and the corresponding input string.

Example from Godefroid et al.

Larger programs have too many paths to explore so they specify various heuristics

In-use at Microsoft
Formal verification

• Can we prove absence of security vulnerabilities?
• No...but we might rule out certain specific vulnerabilities

Model checking

1) Specify a property (typically via FSA) e.g., temporal safety property such as “drop privileges” before running untrusted program
2) Prove that no path exists that violates property (on some sound abstraction)
Bug finding is a big business

• Grammatech (Prof Reps here at Wisconsin)
• Coverity (Stanford startup)
• Fortify
• many, many others...
SoJware

Behaviors

Sound Over-approximation of Behaviors

False Alarm

Repeated Error

approximation is too coarse... yields too many false alarms

Software

Slide credit: Prof Mitchell Stanford’s CS 155
Does this program ever crash?

Slide credit: Prof Mitchell Stanford’s CS 155
Does this program ever crash?

entry

$X \leftarrow 0$

Is $Y = 0$ ?

- yes: $X \leftarrow X + 1$
- no: $X \leftarrow X - 1$

Is $Y = 0$ ?

- yes: Is $X < 0$ ?
  - yes: crash
  - no: exit
- no: Is $X < 0$ ?
  - yes: crash
  - no: exit

infeasible path! ...
program will never crash

Slide credit: Prof Mitchell Stanford’s CS 155
Try analyzing without approximating...

non-termination!
... therefore, need to approximate

Slide credit: Prof Mitchell Stanford’s CS 155
\[ d_{out} = f(d_{in}) \]

Slide credit: Prof Mitchell Stanford’s CS 155
$d_{out1} = f_1(d_{in1})$

$d_{out1} = d_{in2}$

$d_{out2} = f_2(d_{in2})$
What is the space of dataflow elements, $\Delta$? 
What is the least upper bound operator, $\sqcup$?

$d_{out1} = f_1(d_{in1})$
d_{out2} = f_2(d_{in2})
d_{join} = d_{out1} \sqcup d_{out2}$
d_{join} = d_{in3}
d_{out3} = f_3(d_{in3})

least upper bound operator
Example: union of possible values

Slide credit: Prof Mitchell Stanford’s CS 155
We give a lattice (partially ordered list with elements representing union and intersection) to specify the possible values we assign to symbolic variables.
Try analyzing with “signs” approximation...

```
X \leftarrow 0
```

```
\text{Is } Y = 0 ?
```

- If yes, then:
  - \( X = 0 \)
  - \( X = 0 \)
  - \( X = \text{pos} \)
  - \( X = T \)

- If no, then:
  - \( X \leftarrow X + 1 \)
  - \( X \leftarrow X - 1 \)
  - \( X = \text{neg} \)
  - \( X = T \)

```
\text{Is } Y = 0 ?
```

- If yes, then:
  - \( X = T \)
  - \( X = T \)

- If no, then:
  - \( \text{Is } X < 0 ? \)
    - If yes, then:
      - \( X = T \)
      - \( X = T \)
      - crash
    - If no, then:
      - exit

```
\text{terminates...}
```

... but reports false alarm

... therefore, need more precision

Slide credit: Prof Mitchell Stanford’s CS 155
We give a lattice (partially ordered list with elements representing union and intersection) to specify the possible values we assign to symbolic variables.
Try analyzing with “path-sensitive signs” approximation...

- entry
- $X \leftarrow 0$
- Is $Y = 0$?
  - yes: $X \leftarrow X + 1$
  - no: $X \leftarrow X - 1$
- Is $Y = 0$?
  - yes: crash
  - no: exit

- $X = 0$
- $Y = 0$
- $X = \text{pos}$
- $X = \text{neg}$
- $Y \neq 0$

- true
- false
- no precision loss
- refinement

terminates...
... no false alarm
... soundly proved never crashes

Slide credit: Prof Mitchell Stanford’s CS 155
"The most critical servers contain malicious software that can normally be detected by anti-virus software," it says. "The separation of critical components was not functioning or was not in place. We have strong indications that the CA-servers, although physically very securely placed in a tempest proof environment, were accessible over the network from the management LAN."

All CA servers were members of one Windows domain and all accessible with one user/password combination. Moreover, the used password was simple and susceptible to brute-force attacks.

http://www.net-security.org/secworld.php?id=11570