Finding vulnerabilities

CS642: Computer Security

Professor Ristenpart

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

rist at cs dot wisc dot edu
Administrative

• Do people have access to running the HW1 VM?

• Are people still on wait list?
  – send me your name and ID
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

- Obtain Product
  - Manual Network Vulnerability Analysis
  - Private Source Research
- Protocol Analysis
  - Fuzzing
  - Source/Binary Analysis
  - Open Source Research
- Exploit Development

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

<table>
<thead>
<tr>
<th>chunk.left</th>
<th>chunk.right</th>
<th>fb</th>
</tr>
</thead>
<tbody>
<tr>
<td>user data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chunk has:
1) left ptr (to previous chunk)
2) right ptr (to next chunk)
3) free bit which denotes if chunk 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 )
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

\[ b1 = \text{malloc}( \text{BUF\_SIZE1} ) \]
\[ b2 = \text{malloc}( \text{BUF\_SIZE2} ) \]

free()
- Consolidate with free neighbors

\[ \text{free}( b1 ) \]
\[ \text{free}( b2 ) \]
\[ \text{free}( b2 ) \]
\[ b3 = \text{malloc}( \text{BUF\_SIZE1} + \text{BUF\_SIZE2} ) \]
\[ \text{strncpy}( b3, \text{argv}[1], \text{BUF\_SIZE1}+\text{BUF\_SIZE2}-1 ) \]
\[ \text{free}( b2 ) \]

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

\[ (b2 - 8)\rightarrow\text{left}\rightarrow\text{right} = (b2-8)\rightarrow\text{right} \]
\[ (b2 - 8)\rightarrow\text{right}\rightarrow\text{left} = (b2-8)\rightarrow\text{left} \]
What type of vulnerability might this be?

This is ridiculously simple example. Manual analysis is very time consuming.
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 oflow</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 oflow</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>

analyze large code bases

potentially reports many warnings

may emit false alarms

false alarm
false alarm

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>Complete</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td></td>
</tr>
<tr>
<td>Reports all errors</td>
<td>Reports all errors</td>
</tr>
<tr>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td><strong>Undecidable</strong></td>
<td><strong>Decidable</strong></td>
</tr>
<tr>
<td><strong>Unsound</strong></td>
<td></td>
</tr>
<tr>
<td>May not report all errors</td>
<td>May not report all errors</td>
</tr>
<tr>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td><strong>Decidable</strong></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)

Often look for buffer overflows, race conditions
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

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

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

• How many code paths are exercised
• Mutation based
  – Start with known-good examples
  – Mutate them to new test cases
    • heuristics: increase string lengths (AAAAAAAAAAAAAAA...)
    • randomly change items
• Generative
  – Start with specification of protocol, file format
  – Build test case files from it
    • Rarely used parts of spec
White-box fuzz testing

• Start with real input
  – 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

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
}

Example from Godefroid et al.

Say input = “good”

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

This gives set of constraints on input
Negate them one at a time.

Example:
i0 ! = ‘b’ and i1 ! = ‘a’ and i2 ! = ‘d’ and i3 = ‘!’
Figure 2. Search space for the example of Figure 1 with the value of the variable \texttt{cnt} at the end of each run and the corresponding input string.

Example from Godefroid et al.
Fuzz testing

• Black-box is dynamic
• Whitebox is dynamic + static analysis

• Neither sound nor complete
Error

Repeated Error

Sound Over-approximation of Behaviors

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

False Alarm

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 ← 0
Is Y = 0 ?

X ← X + 1
Is Y = 0 ?

X ← X - 1
Is Y = 0 ?

Is X < 0 ?

infeasible path!
... program will never crash
```

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

\[ X \leftarrow 0 \]

Is \( Y = 0 \) ?

\[ X \leftarrow X + 1 \]

\[ X \leftarrow X - 1 \]

Is \( Y = 0 \) ?

Is \( X < 0 \) ?

\[ X = 3 \]

non-termination!

... therefore, need to approximate
\[ d_{\text{out}} = f(d_{\text{in}}) \]

Slide credit: Prof Mitchell Stanford’s CS 155
\[ X = 0 \]

\[
X \leftarrow X + 1
\]

\[ d_{\text{in1}} \leftarrow d_{\text{out1}} \]

Is \( Y = 0 \)?

\[ d_{\text{in2}} \leftarrow d_{\text{out2}} \]

\[ d_{\text{out1}} = f_1(d_{\text{in1}}) \]

\[ d_{\text{out1}} = d_{\text{in2}} \]

\[ d_{\text{out2}} = f_2(d_{\text{in2}}) \]

Slide credit: Prof Mitchell Stanford’s CS 155
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
Try analyzing with “signs” approximation...

- Is $Y = 0$?
  - Yes: $X \leftarrow X + 1$
  - No: $X \leftarrow X - 1$
    - Is $Y = 0$?
      - Yes: $X = 0$
      - No: $X = 0$
- Is $X < 0$?
  - Yes: $X = 0$
  - No: $X = 0$

... but reports false alarm
... therefore, need more precision

Slide credit: Prof Mitchell Stanford’s CS 155
\[ \text{refined signs lattice} \]
Try analyzing with “path-sensitive signs” approximation...

Slide credit: Prof Mitchell Stanford’s CS 155
Coverity

• One of the big names nowadays in static analysis
• Lots of customers (including ones here at Wisconsin)
• One key insight:
  – Statistical bug finding
Gunkerty Jeb writes

"In a keynote speech at the United Security Summit, Stephen Dubner, co-author of *Freakonomics*, drew parallels between the increasingly popular (and successful) practice of software vendors offering bug bounties and a new industry springing up in Johannesburg, South Africa, where the population has recently found itself beset with a growing rat problem. In order to help mitigate their rodent problem, officials in Johannesburg began offering a small monetary rewards for each dead rat turned in. It was wildly successful, and it didn't take long for fresh batch of entrepreneurs to pop up and exploit the situation. Of course, I'm talking about rat farming. Evidently, business minded individuals have taken to breeding rats, only to kill them and turn them in for rewards. Obviously, rat farming is somewhat unscrupulous, but security researchers are doing the same thing: breeding bugs in the lab, then leading them to the slaughter for a nice payday. And it's a good thing."
Tales in insecurity...

"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