Hopelessly Ambitious Reversing Talk
Applying Reverse Engineering to Web Security
about:matasano

★ An Indie Security Firm: Founded Q1’05, Chicago and NYC.

★ Research 2006:
  ‣ endpoint agent vulnerabilities
  ‣ hardware virtualized rootkits
  ‣ a protocol debugger
  ‣ windows vista (on contract to msft)
  ‣ storage area networks (broke netapp)
  ‣ 40+ pending advisories
You may remember me from such research papers as: “Insertion, Evasion, Denial of Service”

or such companies as: Secure Networks, Network Associates, Arbor Networks

or such ISPs as: EnterAct

or such high schools as: St. Ignatius

etc, etc.
about:owasp_talk

★ Reversing and Code-Assisted Pen Test
  › add hours-not-days to projects, find 10x as many flaws

★ Binary Reversing
  › all source is now open; C++, Java, .NET

★ Protocol Reversing
  › busting secret protocols that hide in HTTP
a question:

why did overflows take 7 years to break out?
why reversing matters (1)

- Reversing Will “Break Out” For Attackers
- 1994 Attacker: Shell Scripts, .rhosts
- 2006 Attacker: Assembly, Kernel Heap
why reversing matters (2)

★ The Easy Findings Are Drying Up
★ Pond Fished With Dynamite: Random Binary Fuzzing
★ Matters More For Attackers, But Professionals Must Follow
dueling methodologies: pen test vs. code review
pen test: fast, tactical
pen test: misses stuff
(unexposed form fields, hidden injection)
pen test: limited range

(just CGI variables ala scarab, pantera)
code review: thorough
code review: slow

frequent effort/reward risk
code review: need code
forget third-party dependencies
middle ground

★ Code Assisted Penetration Test
  ‣ use info about code to improve tests
  ‣ test-driven, tactical
  ‣ exploit source, but minimize effort
reverse engineering is now practical
intersection

tool evolution

hex edit  hit trace  decompilation

hardware dependence

.net CLR  C++
rce myth #1

★ End results need to be compilable, nearly as good as the original source code!

› No. Results just need to map out the inputs and operations. We’ll never recompile. We don’t need your algorithms.
rce myth #2

★ All reversed source code needs to be read.

› No. We’re barely going to read any code. We isolate the few functions that matter, figure out their inputs, and test them.
rce myth #3

★ If there are no symbols, reversing is impractical.

› No. Real code is littered with giveaways about which functions are which. Stripping function names adds hours, not days.
rce myth #4

★ The goal of reversing is to get back to the original source language.

› No. All we need is “better than assembly”. We can “decompile” to a call graph, or a low-level language, and analyze that.
rce myth #5

★ All decompilation is static, file-at-a-time.

› No. We’ll use debuggers, system call tracing, filesystems, logging, and single-stepping to help.
int
main(int argc, char **argv) {
    printf("hello, world\n");
    exit(0);
}
closed
push %ebp
mov %esp,%ebp
push %ebx
sub $0x14,%esp
call 0 <LC_SEGMENT.__TEXT.__text>
lea 0x1a(%ebx),%eax
mov %eax,(%esp)
call 37 <___i686.get_pc_thunk.bx-0x5>
movl $0x1,(%esp)
call 32 <___i686.get_pc_thunk.bx-0xa>
call graphed

- read()
- open()
- read()
- write()
- unknown
- memcmp()
- close()
bblock graphed

prologue

condition

funcall

false?

retval

epilogue
hit traced

Diagram:
- read()
- unknown
- memcmp()
- open()
- close()
- read()
- write()
bblock diffed patch

prologue

condition

funcall

false?

retval

epilogue

prologue

condition

funcall

false?

retval

epilogue
class Program {
    public static void main(String args[]) {
        System.out.println("hello, world");
    }
}
class Program {
    public static void main(String args[]) {
        System.out.println("hello, world");
    }
}
Why Java Decompiles

- **Simple instructions**: fits on a Wikipedia page
- **Embedded types**: everything’s an object, objects have names.
- **Storage model**: arguments, locals, instance variables all predictable, along with stack frames
- **Verified code**: can’t jump to the middle of an instruction.
- **Minimal indirection**: no computed function pointers
demo: ida
demo: paimei minesweeper
demo: binnavi eye candy
demo: jad
demo: xcode java
demo: .net reflector
the 8 steps

1. **Configure** the Application: *set up a working lab.*
2. **Sniff Test:** *see if it survives silly stuff.*
3. **Capture Traffic:** *get data to work with.*
4. **Decode and Frame:** *break up messages.*
5. **Establish Replayability:** *start talking to target.*
6. **Establish Variability:** *start attacking target.*
7. **Establish Generation:** *build fuzzing framework.*
8. **Write Test Cases:** *test for coverage.*
(1) configure

★ Get the product working in its normal state.
  › Consider disabling security features for now.

★ We lose more time here than anywhere else.

★ Objective: A VMware “just-add-water” lab.
(2) sniff test

★ Is there any authentication?
★ Can I crash it with random data?
★ **Objective: Qualify the target.**
  ‣ *don’t waste time with totally broken apps.*
(3) capture

★ I use tcpdump to figure out what ports an application uses.

★ I use a simple socket-based plugboard for everything else.

★ **Objective:** files for each side of connection
  
  ‣ *inspect in hexdump*
(4) frame

★ The hardest step.
  › but usually much simpler for web apps
★ Take one capture file.
★ **Objective: files for each protocol message.**
(5) replay

★ Cat message files back at the server
  ‣ (in the right order)
★ Objective #1: successful responses
★ Objective #2: see what varies
(6) vary

★ Now we have examples of protocol messages.

★ **Objective: fuzzing templates**
  ‣ Change strings
  ‣ Change length
  ‣ Change things at random
Now we have a good idea of how the protocol works.

Objective: code to generate from scratch

- I’ve used C, Python, Ruby, and Bash
- I actually prefer Bash.
(8) test cases

★ Start finding flaws.
★ You should be minutes-not-hours for each new test case now.
## protocol decoder ring

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>web</strong></td>
<td></td>
<td><strong>RPC</strong></td>
</tr>
<tr>
<td>HTTP</td>
<td>transport</td>
<td>IIOP</td>
</tr>
<tr>
<td>POST</td>
<td>pdu</td>
<td>Message</td>
</tr>
<tr>
<td>Apache</td>
<td>server</td>
<td>ORB</td>
</tr>
<tr>
<td>Page</td>
<td>service</td>
<td>Object</td>
</tr>
<tr>
<td>URL</td>
<td>request</td>
<td>IOR</td>
</tr>
<tr>
<td>DNS</td>
<td>resolver</td>
<td>CosNaming</td>
</tr>
<tr>
<td>&amp;action=</td>
<td>action</td>
<td>Method</td>
</tr>
<tr>
<td>Cookie</td>
<td>session</td>
<td>SvcContext</td>
</tr>
<tr>
<td>POST Args</td>
<td>data</td>
<td>MessageBody</td>
</tr>
</tbody>
</table>
predictable sessions

<table>
<thead>
<tr>
<th></th>
<th>web</th>
<th>RPC</th>
<th>corba</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cookie</td>
<td>session</td>
<td>SvcContext</td>
<td></td>
</tr>
</tbody>
</table>

proprietary session cookies are almost always monotonically increasing 32 bit integers.
## Forced Browsing

<table>
<thead>
<tr>
<th></th>
<th><strong>Web</strong></th>
<th><strong>RPC</strong></th>
<th><strong>Corba</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Page</td>
<td>service</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>URL</td>
<td>request</td>
<td>IOR</td>
<td></td>
</tr>
<tr>
<td>&amp;action=</td>
<td>action</td>
<td>Method</td>
<td></td>
</tr>
<tr>
<td>Cookie</td>
<td>session</td>
<td>SvcContext</td>
<td></td>
</tr>
</tbody>
</table>

Often, every service/action is left to fend for itself to verify the caller: requests with no session are honored.
memory corruption

<table>
<thead>
<tr>
<th>web</th>
<th>RPC</th>
<th>corba</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>transport</td>
<td>IIOP</td>
</tr>
<tr>
<td>POST</td>
<td>pdu</td>
<td>Message</td>
</tr>
<tr>
<td>POST Args</td>
<td>data</td>
<td>MessageBody</td>
</tr>
</tbody>
</table>

most web apps are built in Java/.NET. most custom protocols are C/C++. 
injection

<table>
<thead>
<tr>
<th></th>
<th>web</th>
<th>RPC</th>
<th>corba</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>POST Args</td>
<td>pdu</td>
<td>Message</td>
</tr>
<tr>
<td></td>
<td>data</td>
<td>MessageBody</td>
<td></td>
</tr>
</tbody>
</table>

requests usually still hit an SQL database, but there’s no off-the-shelf validator code to use. don’t forget ‘90s shell metacharacters and UNC paths!
almost all of these apps have a web front-end somewhere; "submarine" XSS lets us inject javascript into backend database.
conclusion
it seems vanishingly unlikely I’ll make it to this slide.
chisec:
third thursday, every other month, houlihan’s on wacker.