Analyzing Systems

- When presented with a system, how do you know it’s secure?
- Often, you’re called upon to analyze a system you didn’t design — application architects and programmers build it; security people get to pick up the pieces...
- It’s better to build security in from the start, but that doesn’t happen nearly as often as it should
When to Analyze

• The earlier, the better
• Some design decisions are very hard to correct later on
• Better yet, have frequent reviews
• Early reviews concentrate on the broad architecture; later reviews can look at the pieces
Types of Analysis

- Individual programs
- Overall system flow
- Usually, a faulty program means a faulty system, but sometimes faults are containable
Individual Programs

• Look for typical errors: buffer overflow, race conditions, etc.
• Not as easy as it sounds — buffer sizes not always obvious:

```c
void buildmsg(char *dst, char *s, char *msg)
{
    sprintf(dst, "Error: %s: %s\n", s, msg);
    return;
}
```
Which Programs to Check?

• Only check security-sensitive programs

• Which are those?

• Invoking date can be — how much output does it produce?

  
  $ date
  Mon Nov 17 21:51:03 EST 2008
  $ TZ=/usr/share/zoneinfo/Pacific/Guam date
  Tue Nov 18 12:51:11 ChST 2008
  $ TZ=/usr/share/zoneinfo/Pacific/Tahiti date
  Mon Nov 17 16:51:20 TAHT 2008

• Time zones aren’t always 3 capital letters... (and remember the International Date Line)

• Also note: the invoker controls the time zone and hence the length
Another Lesson About Testing

- Blind testing, even in multiple time zones, wouldn’t have found it
- Example: EEST — Eastern European Summer Time — applies during the summer
- Other time zones are in effect only during certain years
- You can write test cases if and only if you know there’s something to test for
- What is the length of a time zone? At least three characters; maximum length is not specified
What To Look For

- Dangerous or potentially functions, i.e., `gets()`, `strcpy()`, `sprintf()`, etc.
- TOCTTOU races — look for `access()`, `stat()` instead of `fstat()`, etc.
- Trusting user input
This Isn’t Easy!

- First step — `grep` for suspect functions
- Each hit requires investigation — and a large program will have hundreds of hits
- Most are obviously and trivially ok
- Most of the rest are ok anyway — but not obviously, and not trivially
Why is it Hard?

- Subprocedures make life difficult for the analyst
- Most routines are called from many different places, with different arguments
- The arguments passed may themselves be arguments from a higher-level procedure
- Buffers may be dynamically allocated, and of uncertain length
Flow Analysis

- We need to understand the *paths* to each suspect call
- Compilers already do that, albeit intra-module
- We can use compiler technology to help us understand complex paths
$ cflow env/* .c
main() <int main (int argc, char **argv) at env/env.c:55>:  
  setlocale()
  getopt()
  usage() <void usage (void) at env/env.c:94>:  
    fprintf()
    exit()
  strchr()
  setenv()
  execvp()
  err()
  printf()
  exit()
A Graphical Call Graph via Cflow
A Partial Reverse Call Graph

$ cflow --reverse env/*.c

err():
    main() <int main (int argc, char **argv) at env/env.c:55>
execvp():
    main() <int main (int argc, char **argv) at env/env.c:55>
exit():
    main() <int main (int argc, char **argv) at env/env.c:55>
usage() <void usage (void) at env/env.c:94>:
    main() <int main (int argc, char **argv) at env/env.c:55>
fprintf():
    usage() <void usage (void) at env/env.c:94>:
    main() <int main (int argc, char **argv) at env/env.c:55>
Simple Tools Don’t Always Work
```c
int i=1, j=2, *pi;
p = subr(&i, &j);
printf("%d", *pi);               /* What will be printed? */
...
int *subr(int *v1, int *v2)
{
    int m;
    scanf("%d", &m);
    if (m > 0) return v1;
    else return v2;
}
```
Is This Analysis Necessary?

- If it’s very hard to understand, perhaps the programmer got it wrong, too
- There is little downside to using safe functions if there is any doubt at all
- There may be a slight performance hit — but the hit tends to be localized, and most sections of code are a very small part of total system performance
TOCTTOU is Harder

- Race conditions are generally part of multi-statement sequences
- Necessary to look for patterns — much more difficult
- Note: `grep` can only point to functions that are frequently involved in race conditions
We Need Tools

• Simple tools such as `grep` are just a starting point
• Custom-built tools can do a better job
• The benefit of tools is that they reduce the search space — they eliminate the many false alarms that `grep` will produce
Inappropriate Trust

● Some scans are relatively easy

● Example: look at `fopen()` calls and see if the input ultimately came from untrusted data

● The trick is knowing the sensitive destinations; depending on the program, it may or may not be easy
Digression: Run-Time Checks

- Sometimes, it’s easier to do the checks at run-time
- Best example: Perl’s “taint mode”
- Data from untrustworthy sources — command-line arguments, environment variables, file input, etc. — is marked as “tainted”
- Any variable derived from a tainted variable is marked “tainted”
- Certain operations cannot be performed with tainted input; a run-time exception is generated
- You can produce untainted variables by a regular expression memory reference; Perl assumes that you know what you’re doing
Other Checks

- See how user inputs are read
- Is the data examined and, if necessary, rejected immediately?
- Are fixed-length buffers used or is `malloc()` called?
- The declaration
  ```
  char buf[1024];
  ```
  is a danger flag
- For C++, is `String` used?
Analyzing Systems

• Both easier and harder

• Easier, because there are fewer components than lines of code

• Harder, because many of the details are abstracted away
Overall Flow

- Identify the separate system elements
- Identify the data flows
- Look for security barriers
- Look for untrusted inputs
System Elements

- System elements are things like web servers, database engines, etc.
- Each of these is itself a complex system that needs to be analyzed
- Establish the properties of each element: where its inputs come from, what its outputs are, what can happen if something is corrupted
Protecting Elements

- What are the forms of access?
- What sorts of access controls are there?
- What is logged? To where? (Who looks at the logs?)
Data Flows

• Who talks to whom?
• How do they talk?
• Is the link exposed to the outside? Is it encrypted? Authenticated?
• Is the protocol otherwise safe?
Security Barriers

- Do they block all attack vectors?
- Are they strong enough?
- Are they flexible enough?
Input Filtering

- Where can enemy input enter the entire system?
- Is it properly checked?
- What about back channels, such as DNS?
System Management

- How will the elements be managed?
- Is more connectivity needed?
- Are other network services used?
- How do system management functions authenticate themselves?
Backups

- How are disks backed up?
- Again, is more connectivity needed?
- How are the backup media protected?
Drilling Down

- Is there other connectivity, such as to the organization?
- If there isn’t now, might there be in the future? (The answer to that one is usually “yes”...) What provisions are made for such connectivity?
- What parts of the design seem more vulnerable?
Weak Spots

- What parts of the design seem problematic?
- Some pieces are weaker than others
- Experience counts here — “trust your feelings, Luke”
Weak Spots: Web Server

• Web servers are quite complex
• CGI or ASP scripts are often locally written, and may have received less scrutiny
• How is the web server checked for intrusions?
• What are the consequences if it falls?
Outcomes of a Review

- All is cool (don’t be afraid to say so, but it rarely happens...)
- A few fixable flaws
- Serious, unfixable problems
- Not deployable
Serious, Unfixable Problems

- There may be flaws that can’t easily be fixed
- Example: a piece of vital third-party software that does stupid things
- Can you layer on something else to provide necessary protection?
- Example: to protect a vendor product that used plaintext passwords, you could add firewalls or a VPN
Not Deployable

- Sometimes, that’s the right answer
- However — how important is the project?
- What is the business cost of not deploying it?
- It’s important to be both honest and realistic — and that’s a delicate balancing act
Software Engineering Code of Ethics

1. PUBLIC - Software engineers shall act consistently with the public interest.

2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.

... (See http://www.acm.org/serving/se/code.htm for the rest.)
Making “No” Stick

- Be prepared to back up your assessment
- Demonstrate *exactly* how an enemy could get in
- Estimate the likelihood of the attack
- Estimate the *business* loss if it happens
- If you can’t do that, it’s more likely the previous category
Bad Excuses You’ll Hear

- It’s closed source; no one knows how it works
  - It’s a lot easier to figure such things out than it appears to those who have never done it
  - What about corrupt insiders?
- Who’d attack us?
  - Some people will attack anything
- No one would try that
  - Some people will try anything
Making Recommendations

- This is often a political process
- Concrete suggestions for improvement are better than “this rots”
- Suggestions should be realistic in terms of cost, benefit, and business situation
- Security is *engineering*; it’s not an absolute goal to be pursued at any cost
- There are always legacy systems you can’t touch