Software Vulnerabilities in Programming Languages and Applications

A presentation to Ada Europe 2010

Stephen Michell, Maurya Software, Ottawa, Canada

Security

There are people out there trying to attack every computer that we own. Most attacks come over the internet, but for high valued assets, attacks can come from anywhere. Most of these attacks leverage vulnerabilities in the applications that we use to gain an advantage over us.

Security

- Attacks attempt to:
  - Steal Resources (money, information)
  - Create a denial of services
    - Prevent execution
    - Corrupt Data
    - prevent communications
  - Cause wrong calculation for nefarious reasons
  - Take over system for own usages
  - Destroy trust in system

Outline

- Attack and Defence
- Resources and Information
- Work of WG 23
- Programming Language Vulnerabilities
- Types of Attacks
  - Net-based
  - library and OS
  - Autorun
  - Hardware
- Avoiding Vulnerabilities

Attack and Defence

- The Easy Ones!
  - Attack over the Internet
    - CWE/SANS Top 25 (security) programming errors
      - Also easiest to defend
  - Attack from mounted devices
    - Autorun files, boot devices
  - BlueTooth
  - Back doors
    - Networks, dial-up lines, attached devices

Attack and defence (cont)

- The Harder Ones
  - Accidental or Planted defects in libraries or OS
  - Planted defects or worms in hardware
  - Programmer planted worms, cookies or Christmas Trees
    - Do you have a programmer with:
      - A grudge?
      - A blackmail-able secret (gambling, gay, etc)?
    - How do you identify code that does not match required functionality?

What's the difference?

- Many attack vectors
  - Net-based attacks need someone to figure out a weakness in the system under attack, then exploit it to get change behaviour or to get a payload in
    - Exploit almost always something illegal under normal circumstances
  - Autorun-based attacks depend on certain features of hardware and OS, and usually include payload on same media
  - H/W, Firmware, library, OS-based attacks depend on attack code already being loaded and triggering condition being transmitted to system somehow
    - Likely legal (undocumented) combination of values or commands

Some of the notorious attacks

- All of the traditional viruses and worms in executables, PDF's, emails
- 2001 (approx) IEEE 802.11 WEP encryption is broken
- 2005 – USAF has personnel database compromised over internet
  - Results in USAF ASACoE being created
- 2007/8 USN discovers that its secure networks sponsor clone CISCO routers that are sending duplicate packets somewhere
- Ongoing – BlueTooth virus attacks
- 2008 – Sequoia AVC Advantage voting machine take over

Defence

- This is hard.
  - You need to recognize every attack vector and defend against every conceivable attack.
  - Attacker only needs to identify 1 weakness and exploit it.
- Basic concept - start at the architecture level and analyse the susceptibilities
  - Architecture (client-server, open network, stand-alone, ...)
  - OS, libraries, hardware, programming language
- Design defence in depth for all possible attack vectors
- More later

Resources and Information

- DHS sites
  - Common Vulnerabilities and Exposures (cve.mitre.org)
    - Very application-specific (right down to version #)
  - Common Weakness Enumeration (cwe.mitre.org)
    - Generalization of CVE's, very language-specific
  - Open Web Security Application Project (www.owasp.org)
    - Very web-oriented
- Above sites do not look beyond the network

Resources and Information(cont)

- Build Security In Website (www.buildsecurityin.us-cert.gov)
  - Good, up-to-date educational and reference material
- ISO/IEC/JTC 1/SC 22/WG 23 Programming Language Vulnerabilities
  - (www.aitcnet.org/isai/)
  - Only truly language-independent consideration of vulnerabilities and delivery-independent consideration
  - First version of technical report published
  - Developing more vulnerabilities

Work of WG 23

- Programming Languages Vulnerabilities Working Group
- Member of ISO/IEC JTC 1/SC 22
- Developing International Report 24772
  - "Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use"
- Documents
  - 53 Language-independent vulnerabilities
  - 19 Application Vulnerabilities
  - Annexes for each of the major Languages
- Work products and drafts of TR available from http://www.aitcnet.org/isai/

Work of WG 23 - progress

- Published 2010 version without any Annexes
  - Ada Annex essentially finished
  - C, Fortran, COBOL Annexes making progress
  - C++, Java, C#, scripting languages not started

Work of WG 23 – sample

- Here are some of WG 23's published vulnerabilities
  - **Bit Representations [STR]**
  - **Enumerator Issues [CCB]**
  - **Numeric Conversion Errors [FLC]**
  - **String Termination [CJM]**
  - **Buffer Overflow [XZB]**
  - **Pointer Casting and Pointer Type Changes [HFC]**
  - **Null Pointer Dereference [XYH]**
  - **Dangling Reference to Heap [XYK]**
  - **Templates and Generics [SYM]**
  - **Inheritance [RIP]**
  - **Initialization of Variables [LAV]**

Programming Language Vulnerabilities

- What is a Programming Language vulnerability?
  - Consider buffer overflow
  - A deliberate write to a buffer that exceeds its bounds in many OS's and languages is permitted
    - On stack, may overwrite return address
    - On heap, may overwrite address of a function
  - If code has been written at same time (or existed before) and address of that code is coerced into the return address or function, attacker has just assumed control of the machine

Programming Language Vulnerabilities

- So what is the vulnerability?
  - Programming language permits the attacker (either via an input or due to an explicit deviance in the code) to write outside an object to gain some advantage.
- Does Ada have this vulnerability?
  - Not unless you
    - use unchecked programming or
    - disable runtime checks or
    - link into libraries written in another language

Types of Attacks

- Net-based
- Autorun worms
- Libraries and OSs
- Hardware
- Program itself

Network Based Attacks

- SANS/CWE Top 25 Vulnerabilities
  - Available from http://cwe.mitre.org
- All network based
- All oriented to opening up a system from the outside

Things to note about net-based attacks

- Rely upon fundamental mistakes by programmer and language systems
  - Mostly trust:
    - In the provider of input
    - In OS/libraries/language system to catch and handle errors
- Examples
  - Input that exceeds buffer sizes or character expectations (eg 8 bit vs 16/32 bit)
  - Input that gets translated into OS commands

CWE/SANS Top 25 - #1-10

- [1] CWE-79 Failure to Preserve Web Page Structure ('Cross-site Scripting')
- [2] CWE-89 Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')
- [3] CWE-120 Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- [4] CWE-352 Cross-Site Request Forgery (CSRF)
- **[5]** CWE-285 Improper Access Control (Authorization)
- **[6]** CWE-807 Reliance on Untrusted Inputs in a Security Decision
- **[7]** CWE-22 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
- **[8]** CWE-434 Unrestricted Upload of File with Dangerous Type
- **[9]** CWE-78 Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection')
- **[10]** CWE-311 Missing Encryption of Sensitive Data

CWE/SANS Top 25 #11-20

- [11] CWE-798 Use of Hard-coded Credentials
- [12] CWE-805 Buffer Access with Incorrect Length Value
- **[13]** CWE-98 Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion')
- **[14]** CWE-129 Improper Validation of Array Index
- **[15]** CWE-754 Improper Check for Unusual or Exceptional Conditions
- **[16]** CWE-209 Information Exposure Through an Error Message
- **[17]** CWE-190 Integer Overflow or Wraparound
- **[18]** CWE-131 Incorrect Calculation of Buffer Size
- **[19]** CWE-306 Missing Authentication for Critical Function
- **[20]** CWE-494 Download of Code Without Integrity Check

CWE/SANS Top 25 #21-25

- **[21]** CWE-732 Incorrect Permission Assignment for Critical Resource
- **[22]** CWE-770 Allocation of Resources Without Limits or Throttling
- **[23]** CWE-601 URL Redirection to Untrusted Site ('Open Redirect')
- **[24]** CWE-327 Use of a Broken or Risky Cryptographic Algorithm
- **[25]** CWE-362 Race Condition

Avoiding the CWE/SANS Top 25

- Fairly straightforward
- Commercial tools available that do a good job of static analysis of code for dangerous patterns.
  - Fortify, IBM Rational Purify, Coverity, etc.
  - Basic rules cover many of the CWE vulnerability list.
  - Never have complete coverage
  - Always improving and adapting

Avoiding the CWE/SANS Top 25 (cont)

- Many OS's now take precautions to make these attacks harder
  - eg, loading executable, global space, constant space, heap, stack at different address each time

Library code vulnerabilities

- Accidental
  - Code that does not obey the documentation given to callers
    - Parameter mismatch
    - Undocumented exceptions or error codes
      - Behaviour mismatch
      - Protocol mismatch
      - Data errors

Library code vulnerabilities (cont)

- Deliberate
  - Code that contains undocumented behaviour
    - Triggered on certain input values or library state

OS vulnerabilities

- See Library code vulnerabilities, except
  - Might be
    - In driver
    - In loadable modules
    - Subject to OS, hardware state, etc.

Detecting Library & OS vulnerabilities

- Use the C bounded libraries
- If you own the library, or have it in source, evaluate it with static tools mentioned before
- If you acquired binary, consider
  - Getting source,
  - Switching libraries,
  - Getting certification,
  - Testing it as well as you can

Detecting Library & OS vulnerabilities (cont)

- For libraries, tools that eliminate unused calls and dependent code help
- Easier in systems with byte code (eg Java) – lots of support in Java RT

Autorun worms

- Can reside on any device
  - Flash memory
  - CD/DVD Rom
  - USB devices – cameras internal memory, etc
- June 2010, Canon acknowledges that the Stylus Tough 6010 contains an autorun worm
- A number of cheap USB keys have been reported to contain autorun worms

Hardware

- Not just CPU
  - Video, Network, auxiliary, Outboard processors (gateways, etc)
  - Contain different memory types (ROM, EPROM, RAM)
  - Can often access ALL cpu memory in privileged mode
  - Code may be resident or loaded as part of initialization
  - remember that almost all smart hardware contains code

Safe hardware (maybe)

- Are there any safe hardwares?
- Maybe Harvard architecture
- Harvard Architecture
  - 2 memories
    - 1 for execution code
    - 1 for data
  - Impossible to execute instructions from data memory
  - Impossible to do functional or OO programming
  - Impossible to crack if instruction data is ROM

Safe hardware (maybe)

- Oh Yeah?

Safe Hardware (not)

- Checkoway, Halderman et al (UCSD) *
- Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the Sequoia AVC Advantage
  - Took a Harvard architecture voting machine (Sequoia AVC Advantage)
  - Without changing the voting machine code at all, took total control of the machine
  - Used return-oriented programming trick and available runtime libraries in the code to build a Turing machine

* http://www.usenix.org/event/evtwote09/tech/full_papers/checkoway.pdf

How does Return Oriented Programming Work?

- 2 basic principles of stack-based cpu's
  - return address on stack in data memory
  - After return, cpu pops stack
- Find small snippets of code followed immediately by a return that perform 1 function
  - catalogue those functions and addresses
- Push those addresses onto a stack to execute your instruction set to execute the program
- You return to the instruction, it executes instruction, returns through your next (logical, previous physical) address

see: https://cseweb.ucsd.edu/groups/security/avc/avc.pdf

Vulnerabilities in your code

- Deposited by a developer with a grudge, problem, etc
- Who better knows your system and where (and how) to hide something, and how to avoid any analysis tools that you use.

Avoiding vulnerabilities in your code

- Reviews – peer review, team review
- Trace requirements through design to code
- Verify code back to requirements
- Practice minimalism

Avoiding Software Vulnerabilities (1)

- Security Requirements
  - Build security awareness into the complete software development process
  - Identify the need for security and create explicit requirements for security

Avoiding Software Vulnerabilities (2)

- Design for Security
  - Become aware of security issues associated with alternate architectures
  - Make security a major criteria in the selection of architecture

Avoiding Software Vulnerabilities (3)

- Security as part of SEE
  - Become aware of security issues associated with development languages and make security a major criteria in the selection of development languages and environments

Avoiding Software Vulnerabilities (4)

- Static Analysis Tools
  - Acquire and use static analysis tools to detect code patterns that could result in known application vulnerabilities

Avoiding Software Vulnerabilities (5)

- Solid Review Processes
  - Develop explicit team code review processes to identify vulnerabilities that are not well handled by analysis tools
  - *Side effects:*
    - *Cross-training of other team members,*
    - *Better understanding of complete system*

Avoiding Software Vulnerabilities (6)

- Testing
  - Acquire and use testing tools that test for known, web-based vulnerabilities
  - Build and use your own testing tools for non-web based portions of the system

Avoiding Software Vulnerabilities (7)

- Practice defence in depth
  - Do not trust gatekeeping and security modules to protect
  - Provide error detection and recovery or attack detection and recovery/notification at every level

Avoiding Software Vulnerabilities (8)

- Use
  - encryption,
  - checksums,
  - validation techniques
  - principle of least privilege,
  - principle of privacy,
  - principle of restricting information

Avoiding Software Vulnerabilities

- Implement a security development similar to IEC 61508-3 safety process

Conclusions

- Software Security is a pernicious problem
- Must predict and avoid all possible attacks
- Attacker only needs to find 1 way in
- Tools, processes and intelligence are necessary

Questions?