Security in the Software Life Cycle

OMG SwA Workshop
5 March 2007

Karen Mercedes Goertzel, CISSP
703.902.6981 goertzel_karen@bah.com

Booz | Allen | Hamilton

Who this tutorial is for

- Software engineers and developers who want to build software whose dependability isn't easily compromised
- System engineers interested in the security of the software components of their systems
- IT security practitioners interested in software-specific security issues

Booz | Allen | Hamilton

Questions this tutorial should answer

- What do we mean by “secure software”?
- What are the threats to all software?
- What makes software vulnerable to those threats?
- How does the way software comes into existence affect its security?
- What techniques and tools can be used to produce (more) secure software?
- What resources are available to help developers do this?

Out of Scope

- Using software to implement system security functions
- System-level security concerns
  - e.g., identity and trust management, access control, network security, user accountability, session management
- Operational security concerns
  - e.g., incident response, anti-virus/anti-spyware, secure system administration
- Information assurance concerns
  - e.g., encryption/decryption, data labeling, information flow security, privacy

Making sure software doesn't make information vulnerable: important but out of scope

In scope: Making sure software itself isn't vulnerable

Booz | Allen | Hamilton

Why care?

- Software is everywhere.
- It isn't just applications. It's also
  - operating systems
  - frameworks
  - middleware
  - security systems
  - communications/networking systems
  - embedded systems
  - firmware (shares with software: executable, readable, writeable, and at risk).
- Software monitors and controls life-critical physical systems.
- Software manipulates, protects, and exposes extremely sensitive information.
- Software is itself protected by other software.
- The vast majority of software is *not* "built from scratch".

Threats to software

- External
  - Human attackers
  - Malicious processes
- Inside
  - Rogue developers
  - Rogue administrators
  - Rogue users
- Embedded
  - Malicious logic
  - Intentional vulnerabilities
  - Backdoors

Threats are gaining in sophistication, variety, persistence, and impact.

Booz | Allen | Hamilton

But we're not connected to the Internet!

When software is threatened

- In development and maintenance, by
  - “Rogue” developer sabotage and subversion by planting
    - malicious code (“ bombs” and other undocumented functions)
    - intentional faults, weaknesses, vulnerabilities
    - exploitable backdoors, trapdoors
- In distribution and deployment, by
  - External attackers (intercepting and tampering with distribution)
  - Insider threats (administrators intentionally tampering, misconfiguring, planting malware, rootkits, etc.)
- In operation, by
  - External attackers (level of exposure varies with level of network connectivity/exposure)
  - Insider threats (users and administrators abusing privileges, not applying patches)

Categories of attack patterns

- Direct attacks
  - To exploit known or suspected faults, vulnerabilities, weaknesses, backdoors
  - To insert malicious code
  - To execute malicious code already embedded in the software
  - To observe or reverse engineer the software
- Indirect attacks
  - Intentional activation of external faults at the software's boundaries
  - Intentional changes to execution environment state
  - "Hogging" of the software's processing resources
  - Sabotage or subversion of external services or defense-in-depth measures on which the software relies

Attack objectives (desired direct results)

- Reconnaissance
  - To learn more about the software in order to craft more effective attacks
- Subversion
  - To change the software's functionality, by tampering or insertion of logic
- Sabotage
  - To make the software fail
    - suddenly crash or gradually degrade in performance
  - To make the software inaccessible
    - by moving or deleting its executable
    - by corrupting its user interface or communications capability

      Note: changing the executable's file system permissions would have the same result, but is a system-level threat.

What makes software vulnerable?

- It’s big and complicated, and getting more so – humans can no longer fully comprehend it.
- Component-based development: COTS, OSS, and reuse means no-one really knows where most of it comes from, or how it was built.
- It contains lots of faults and weaknesses. Many of these are exploitable.
- It comes in binary executable form, which makes finding those faults and weaknesses a lot harder.
- It’s exposed to threats *all* the time, *even while it’s under development*.

Where vulnerabilities originate (1)

During development

- Inadequate or spurious requirements
- Inadequate architecture, assembly option, detailed design
- Use of vulnerable processing models, software technologies
- Insecure use of development tools, languages, libraries
- Use of insecure development tools, languages, libraries
- Poor coding practices
- Coding errors
- Use of vulnerable/unpatched components
- Incorrect or mismatched security assumptions
- Inadequate reviews, testing, assessments

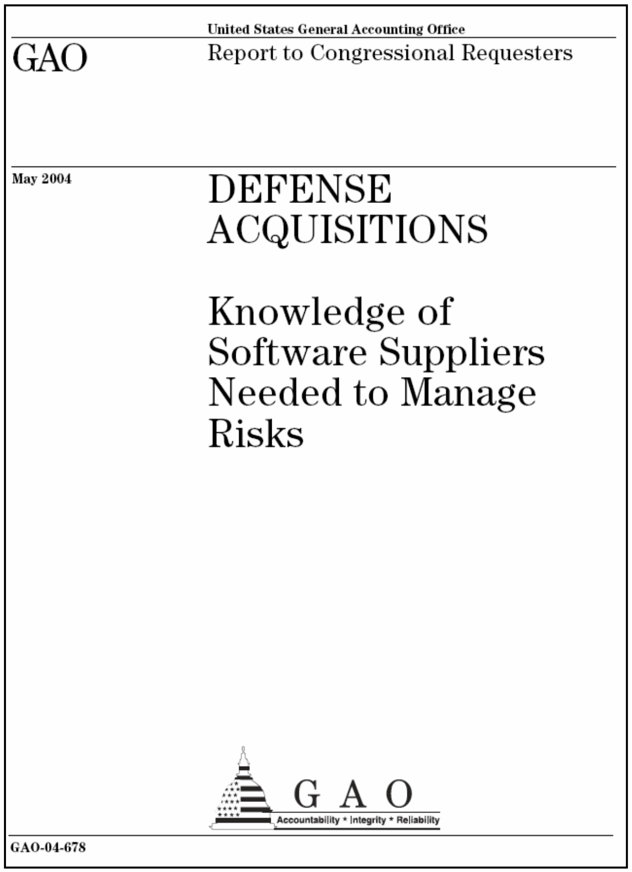
Where vulnerabilities originate (1) cont'd

- Sabotaged test results
- Residual backdoors
- Sensitive info about software problems in user-viewable comments/error messages
- Inadequate configuration documentation
- Insecure installation procedures, scripts, tools

Malicious code planted during development

- Trojan horses
  - Software that seems to do one thing, but actually does another
- Time bombs
  - Software whose execution is triggered at a predefined time (on computer clock)
- Logic bombs
  - Software whose execution is triggered by a predefined event or input
- Malicious undocumented functions ("rotten Easter eggs")

Hard Problem: Software of Unknown Pedigree (SOUP)



United States General Accounting Office

GAO

Report to Congressional Requesters

May 2004

DEFENSE ACQUISITIONS

Knowledge of Software Suppliers Needed to Manage Risks

GAO

Accountability * Integrity * Reliability

GAO-04-678

December 1999: Defense Science Board Task Force on Globalization and Security reports on “Vulnerability of essential U.S. systems incorporating commercial software”

December 2002: Business Week reports “ Software, security, and ethnicity. The U.S. government’s probe at software maker Ptech, owned by a Lebanese, has lots in common with the 1998 Wen Ho Lee case”

July 2003: The New York Times reports “ Uneasiness about security as government buys software”

April 2004: LinuxInsider/ECT News Network reports “Expert says Linux a “threat” to U.S. national security”

May 2004: GAO publishes Report to Congressional Requesters entitled “Defense Acquisitions: Knowledge of Software Suppliers Needed to Manage Risk”

June 2004: IDG News Service reports “Security vendor says offshore development needs check. Extra steps called for to ensure secure code”

2006: ACM publishes “Globalization and Offshoring of Software”, pointing out risks to national security from government use of offshored software. No. 1 risk: difficulty understanding code pedigree may allow hostile nations, terrorists, criminals to subvert or sabotage software used in government systems.

November 2006: Computerworld announces “DOD report to detail dangers of foreign software. Task force says U.S. adversaries may sabotage code developed overseas”

Where vulnerabilities originate (2)

- **During deployment and operation**
- Insecure configuration of software and its environment
- Inadequate allocation of resources
- Failure to apply patches
- Software aging

Secure software…

- Preserves all of its required properties in the face of threats to those properties
  - Dependability is the #1 desirable property for all software
    - If it doesn't work correctly and predictably at all times, what good is it?
- Can resist and/or tolerate most threats that attempt to subvert or sabotage it
- Can terminate, limit the damage, and rapidly recover from the few that succeed

Dependability properties

- Quality (correctness and predictability)
- Reliability
- Fault-tolerance
- Trustworthiness
- Safety (the above intensified: failure threatens human life or health)

Security properties

- Integrity
  - can’t be subverted
- Availability
  - can’t be sabotaged
- Trustworthiness
  - won’t do the unexpected
    - not the same as trustworthiness of software as non-human “user”

Booz | Allen | Hamilton

Security properties

- Confidentiality (of the software itself)
  - as a subject: behaviors, states, actions
  - as an object: executable file location, characteristics, contents
  - deters reconnaissance, reverse engineering
  - less likely to be a requirement for software than for information
- Assurability
  - ability to verify software’s required properties, including security
  - aided by smallness, simplicity, traceability

What makes software secure?

- Attack-resistance
  - Components and whole system recognize and *resist* attack patterns.
  - System recognizes suspicious component behaviour and either
    - isolates/constrains that behavior
    - terminates execution of the component
- Attack-tolerance
  - Components keep operating in spite of errors caused attacks
  - System keeps operating in spite of attack-caused component errors/failures
- Attack-resilience
  - System constrains damage from attacks it could not tolerate, isolates itself from attack source
  - System rapidly recovers (at least to minimum acceptable performance)

Security throughout the life cycle

- Security-enhancing process improvement model
  - e.g., FAA iCMM/CMMI safety & security extensions, SSE-CMM
- Security-enhancing life cycle methodologies
  - e.g., CLASP, SDL, McGraw's 7 Touchpoints, TSP-Secure, AEGIS, RUPSec, SSDM, Oracle Secure SW Assurance, Waterfall-Based SW Security Engineering Process
- Establishing security entry and exit criteria for each life cycle phase
- Including appropriate and sufficient security reviews, analyses, tests at each phase
  - e.g., threat models, attack trees, vulnerability analyses, code reviews, black box tests, risk analyses, assurance cases
- Secure SCM
- Education, training, awareness, professional certification
- QA of security of software processes and practices

Secure requirements engineering

- Risk-driven vs. functionality-driven:
  - non-functional requirements (what software must *be*, vs. what it must *do*)
  - constraint requirements
  - negative requirements
    - Need to allow time for translating these into requirements for functionality (what can be built/tested)
      - e.g., no BOFs = must do input validation; must be fault-tolerant = must have exception handling that…)

Reducing SW security risk: acquisition

- Include security requirements and evaluation criteria in all RFPs
- Strict monitoring/control of “non-traditional” acquisitions (e.g., OSS, shareware, freeware downloads)
- Supplier and integrator background checks (COTS)
- Supplier and integrator SDLC process reviews
- Contract language requiring COTS suppliers to warrant safe, secure product behaviour
- Pedigree analysis, security testing of *all* candidate components *before (!)* purchase (COTS, shareware) or integration (OSS, freeware)
  - Ideal world: acquisition policy that favors software with *known* pedigree

Problems with technological precommitments

- Commitments to use specific technologies and products are increasingly made at the enterprise level, then backed up by policy.
- Requirements of individual systems are seldom considered.
- Software and system engineering become exercises in working around undesirable features and properties.
  - Requirements have to be written in ways that ensure they can be satisfied within the constraints imposed by technological precommitments.
  - Additional requirements must be added to mitigate known vulnerabilities and security mismatches that use of precommitted technologies/products introduce.
- Thorough, iterative risk analyses throughout system lifecycle should capture unacceptably high cost of workarounds and countermeasures, make case for waiving precommitments to high-risk technologies/products.

What does Common Criteria evaluation say about software's security?

- It doesn't look at the right products.
  - Products without significant security functionality are not eligible for CC evaluation.
- It doesn't ask the right questions.
  - Focus of CC evaluations is on correctness and security policy conformance of TOE's security functions/controls.
  - Little if any CC language addresses software security concerns.
    - Software assurance language was added to draft CC v.3.
      - ISO/IEC period for considering draft expired before v.3 adoption.

What does Common Criteria evaluation say about software security? cont'd

- It doesn't look at the product in helpful ways.
  - CC evaluation is based predominantly on documentation analysis.
  - Direct testing of TOE limited to correctness of security functions.
- It doesn't adequately address the product's development process.
  - No rigour in product security engineering required below EAL5.
  - No formal methods are required until EAL7.
    - Most products are evaluated at EAL4 or below.

Reducing SW security risk: source selection

- Analyze individual components
  - code review, security tests, vulnerability scans
  - identify mismatches of security assumptions in pairs of components (including candidate component and environment component pairs)
  - evaluate other evidence (published vulnerability reports/patch history, C&A or CC history, supplier reputation, development process)
  - identify security/countermeasure requirements for component-based architecture
  - determine feasibility and cost of security measures and countermeasures needed to minimise exposure of component vulnerabilities

Booz | Allen | Hamilton

Secure software architecture and design

- System processing model doesn't preclude secure behaviors, interactions
- Minimisation of vulnerabilities—quantity and exposure—through security measures and countermeasures (discussed later)
- Secure intercomponent and extrasystem interfaces (APIs, RPCs, UIs) Prevents excessive trust in high risk (including SOUP) components
- Absolutely minimises privileges granted to *all* processes/components at all times
- Isolates and constrains environment in which high-risk software operates
- Minimises untrusted software access to/interaction with trusted software

Secure software architecture and design

- Addresses mismatches in components' assumptions about each other:
  - Component A may expect Component B to provide certain
    - functionality (e.g., signature validation)
    - properties (e.g., fault tolerance)
    - outputs (format, length, etc.)
    - interfaces (APIs, RPCs, protocols)
- Addresses inaccurate assumptions about the environment:
  - Component may expect the execution environment to provide
    - certain functionality (e.g., PKI)
    - certain protection (e.g., sandboxing)
    - certain inputs (i.e., environment parameters

Security issues of component-based software

- Mismatches in component assumptions about each other and execution environment: Component may expect…
  - certain functionality in another component (e.g., signature validation)
  - certain functionality in the environment (e.g., PKI)
  - certain properties in other components (e.g., fault tolerance)

Sources of inaccurate assumptions

- Incomplete, omitted, overly-general, or poorly-stated functionality-constraining and nonfunctional property requirements
- Failure to translate such requirements into actionable requirements
- Architecture and design that do not satisfy their actionable non-functional (property) and negative (constraint) requirements
- Ignoring the security implications of different languages, tools, and technologies, and how they are used in implementing the software
- Failure to evaluate security of nondevelopmental components, alone and in combination with other components, before selection
- Security reviews/tests not included in each SDLC phase

Sources of inaccurate assumptions cont'd

- Test cases limited to normal operating conditions
- Lack of risk-driven security testing, i.e., abnormal conditions, test cases based on attack patterns
- Lack of stress testing, i.e., abnormal activity, inputs, etc. to validate design assumptions
- Inadequate preparation of the software for distribution/deployment
- No verification that security standards have been conformed to
- Software design does not match intended operational environment

SOUP = inaccurate security assumptions

- Unable to infer component trustworthiness from knowledge of development process
- Unable to infer component trustworthiness from supplier reputation
- Disjoint product and patch release schedules
- Disjoint supplier priorities vs. system requirements
- Publishing of known vulnerabilities: attackers know at least as much as system developers
  - Attackers don't care about license Ts&Cs "preventing" reverse engineering, which means they probably know much more.
- Potential hostile foreign influence on offshore developers may result in products with embedded malicious code, rotten Easter eggs, intentional vulnerabilities

Booz | Allen | Hamilton

Reduce SOUP risk: architecture

- Define different candidate system architectures in which to evaluate components, model component risks
  - include threat, attack, vulnerability modeling for each candidate architecture
  - evaluate both architecture and components together
    - architecture provides framework for revealing intercomponent behaviors, assumption (mis)matches
    - candidate components verify security of architecture-defined component combinations, configurations, process flows

Secure implementation and testing

- Secure coding practices supported by tools
- Write, acquire, reuse only components proven dependable, free of exploitable faults and weaknesses
- Security testing
  - White box:
    - static and dynamic code analysis
    - fault injection/propagation analysis
  - Black box
    - fault injection
    - fuzzing
    - penetration testing
    - vulnerability scanning

Booz | Allen | Hamilton

Reduce SOUP risk: testing, risk management

- Black box—and when source code is available, white box—security testing
  - individual components
  - pairs of components
  - whole system
- Ongoing risk analysis and reengineering
  - find known-pedigree components with req'd capabilities to replace SOUP
  - redesign system so SOUP components' capabilities are no longer needed
  - apply new countermeasures to further reduce SOUP component risk

Secure distribution, deployment, maintenance

- Trusted distribution techniques
  - code obfuscation
  - digital watermarking
  - code signing
  - authenticated, encrypted download channels
- Install. configuration that ensures
  - secure interactions with execution environment
  - adequate allocation and safe management of environment resources
- Maintenance
  - impact analyses of new requirements, own and supplier updates, patches
  - ongoing risk assessment to identify new requirements
  - forensic analysis (post-incident) to identify new requirements

SW security measures and countermeasures

- Programmatic
  - input and output validation wrappers
  - obfuscation (to deter reverse engineering)
  - secure exception handling (in custom software)
  - fault tolerance measures
    - redundancy
    - diversity (redundancy using different components with comparable functions)

Security measures and countermeasures cont'd

- Development tools and languages
  - type-safe languages
  - safe versions of libraries
  - secure compilers
  - secure compilation techniques
- Environment-level measures
  - virtual machines/sandboxes
  - chroot jails
  - trusted OS with mandatory integrity policy/compartments
  - secure microkernels
  - TPMs
  - program shepherding
  - altered memory maps
  - system call filters

Security measures and countermeasures cont'd

- Add-ons
  - code signing with signature validation
  - obfuscation and digital watermarking (to deter reverse engineering)
  - malware/spyware scanners (host level)
  - application security gateways/firewalls
  - intrusion detection/prevention (network and host based)
- Development process (more on this

Booz | Allen | Hamilton

Resources

- K.M. Goertzel, *et al: Security in the Software Life Cycle* Draft Version 1.2 (DHS NCSD Software Assurance Program, Sept. 2006) – new version planned in 2007
- IATAC/DACS: *Software Security Assurance: a State of the Art Report* (to be published June 2007)
- US-CERT BuildSecurityIn portal

  `https://buildsecurityin.us-cert.gov/`

- NIST SAMATE portal

  `http://samate.nist.gov/`

Not only do the above have useful content, they include extensive pointers to other online and print resources (too numerous to list here).

Booz | Allen | Hamilton