# Automatic vs. Manual Code Analysis

Ari Kesäniemi
Senior Security Architect
Nixu Oy
ari.kesaniemi@nixu.com

**OWASP**
2009-11-17



**The OWASP Foundation**
http://www.owasp.org

# Agenda

- Automatic and manual security verification overview
- Pros and cons for both techniques
- Statistics from WASC
- Application Security Verification Standard and automatic verification
- Source code scan problem areas
- Mixing automatic tools and manual work
- Conclusion

# Application Security Verification

- Automated vs. manual
- Dynamic vs. static
- Dynamic can be black-box or white-box
- This presentation focuses on static

| | Dynamic | Static |
|---|---|---|
| **Automatic** | Dynamic Scan | Source Code Scan |
| **Manual** | Security Test | Code Review |

# Code Review Techniques

- Automatic source code scan:
  - Text matching in source code
  - Token matching
  - Abstract syntax tree analysis
  - Input/output path analysis
  - Complexity analysis
  - Statistical analysis

  - Do most automatic code scan findings relate to *how data is handled in the application*, and not so much to the actual behavior and its consequences?

# Code Review Techniques

- Manual code review:
  - Look for specific signs (→ text matching)
  - Attack surface discovery
  - Input/output path analysis
  - Component usage and configuration analysis
  - Authorization logic validation
  - Custom security constraints, e.g. approval procedures
  - Privacy issues
  - Architecture analysis
  - … etc

# Automatic vs. Manual

- Manual review by an expert gives
  - probably less false negatives
  - certainly less false positives
  - insight also concerning design and architecture, overall quality etc
- OTOH, automatic review is
  - faster
  - broader
  - repeatable

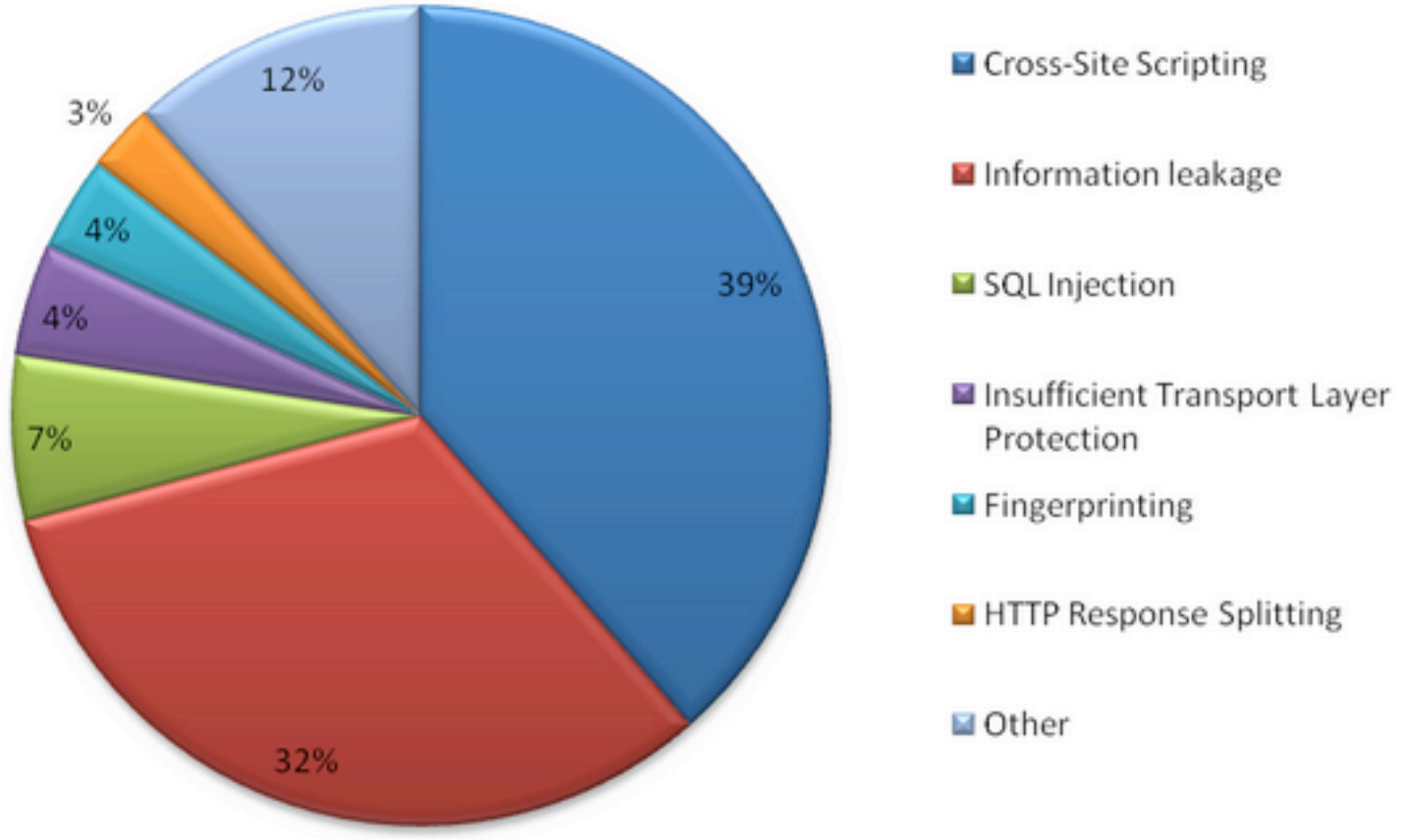
# Interpreting automatic analysis results

- Every security verification needs conclusions of results
- Is security expertise needed?
  - Assessing true positives (real findings)
  - Determining false positives (false alerts)
  - Estimating false negatives (undiscovered vulnerabilities)
  - Making sure scanning configuration is correct
- Can we get good suggestions on how to remedy vulnerabilities automatically?

# Some tools for source code scan

| Tool | Languages | Type |
|---|---|---|
| ITS4 | C/C++ | Token matching |
| Splint | C | Semantic matching |
| Flawfinder | C/C++ | Text matching |
| RATS | C/C++, Perl, PHP, Python | Text matching |
| LAPSE | Java EE | Data flow analysis |
| | | |
| Fortify 360 SCA | | |
| Ounce 6 (IBM AppScan Source Edition) | | |
| IBM AppScan | | |

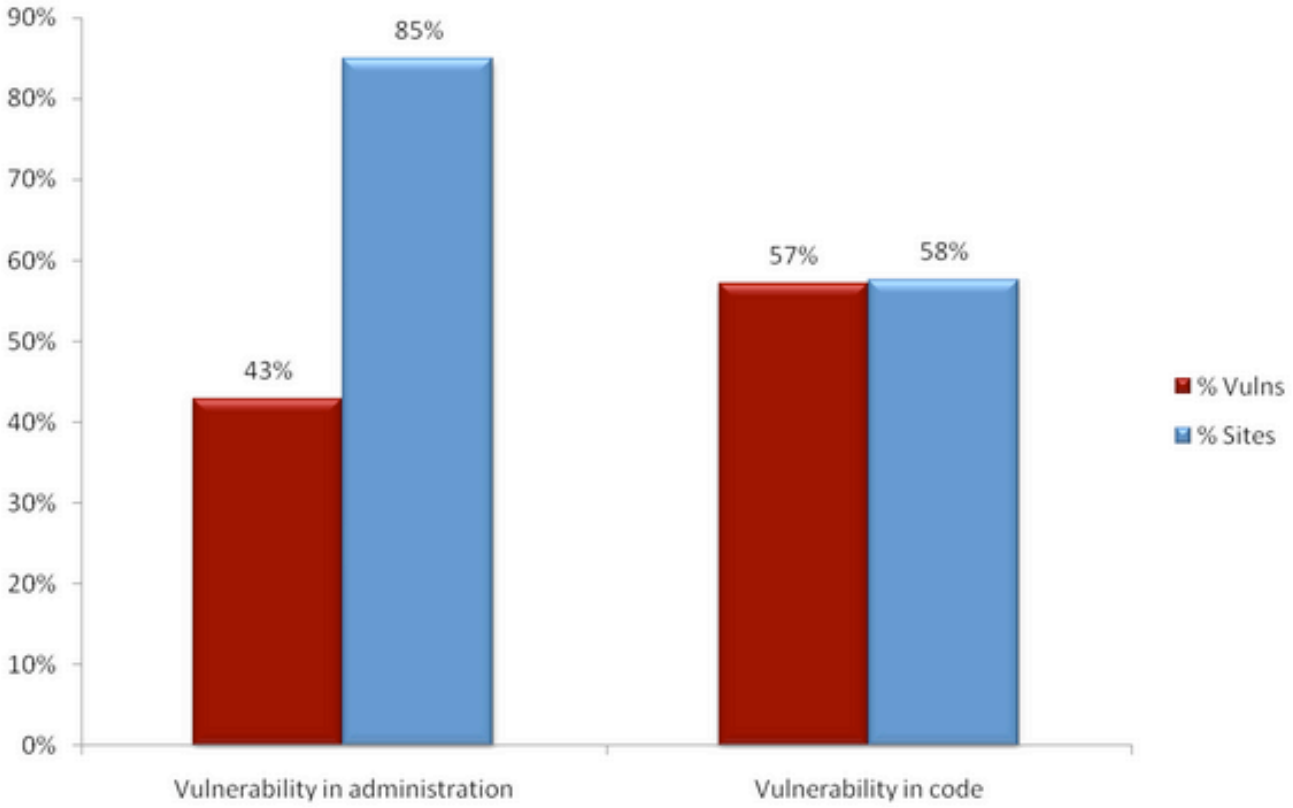
# WASC Web App Security Statistics 2008




Source: Web Application Security Consortium (WASC)
http://projects.webappsec.org/Web-Application-Security-Statistics

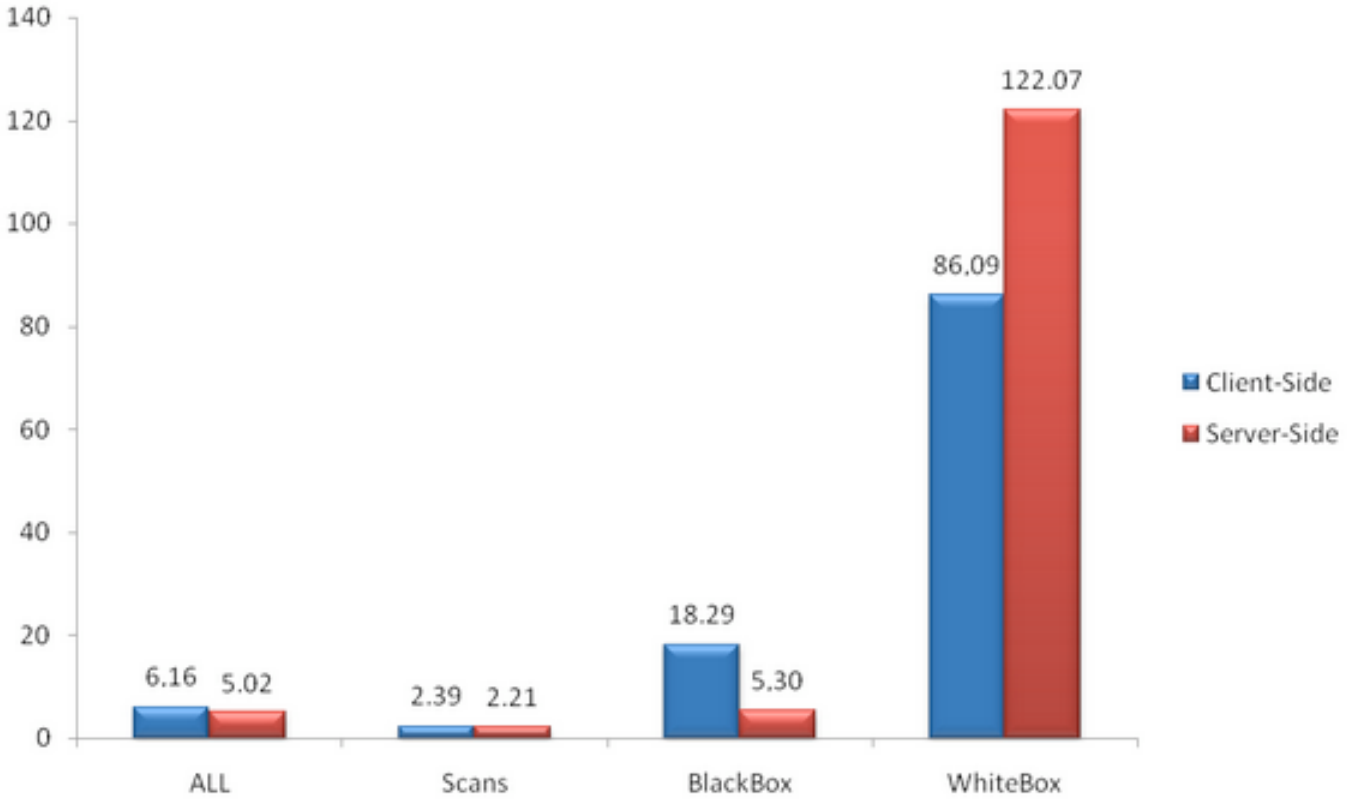
# WASC Web App Security Statistics 2008

- Less than 60% of vulnerabilities are in code

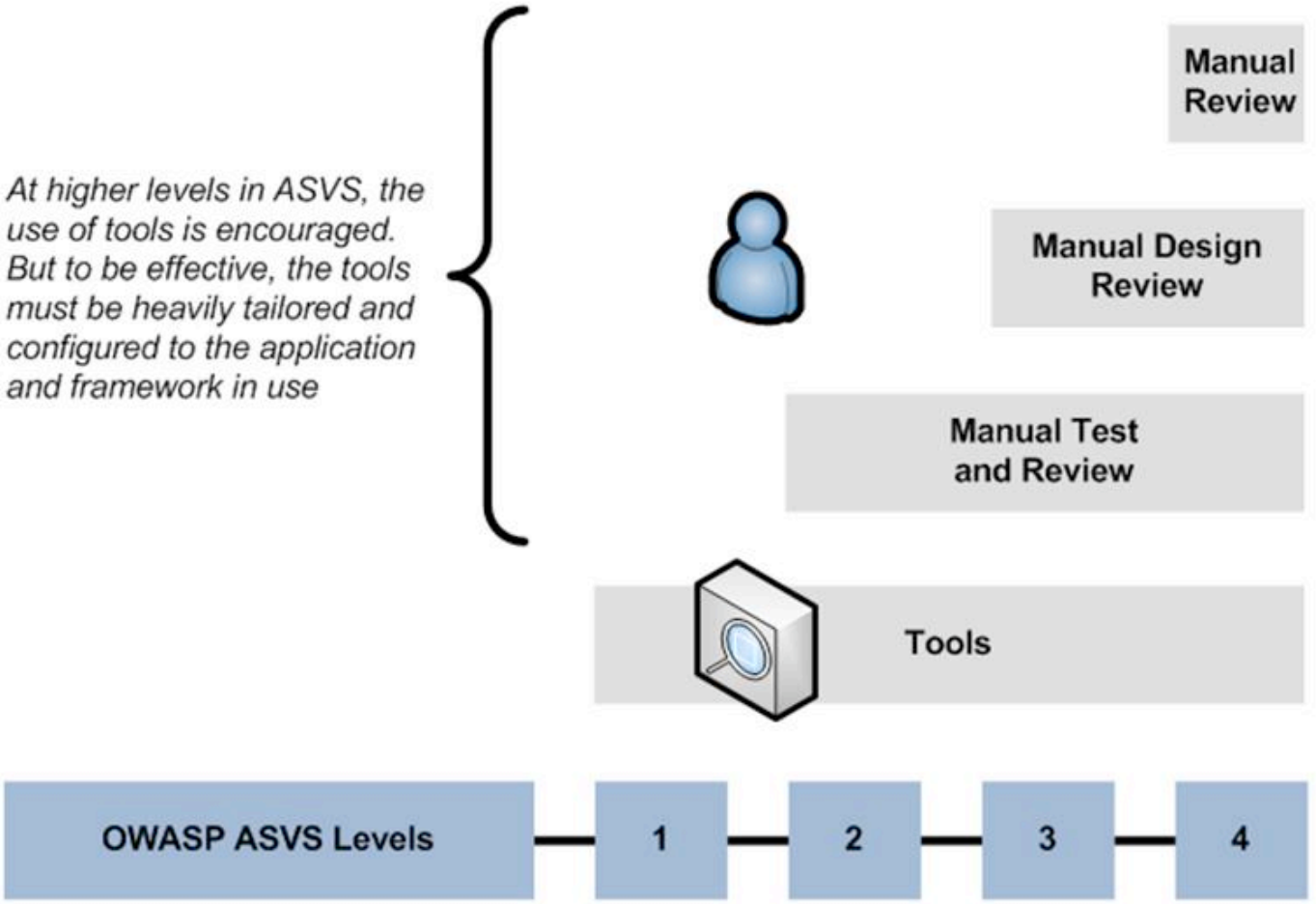


# WASC Web App Security Statistics 2008

- Whitebox approach (dynamic or static) is needed to catch vulnerabilities



# OWASP Application Security Verification Standard (ASVS)



# Types of security verification

| | Dynamic | Static |
|---|---|---|
| **Automatic** | Dynamic Scan (1A) | Source Code Scan (1B) |
| **Manual** | Security Test (2A) | Code Review (2B) |

- In addition, in ASVS
  - level 3 is “Design Verification” (manual)
  - level 4 is “Internal Verification” (manual)

# ASVS

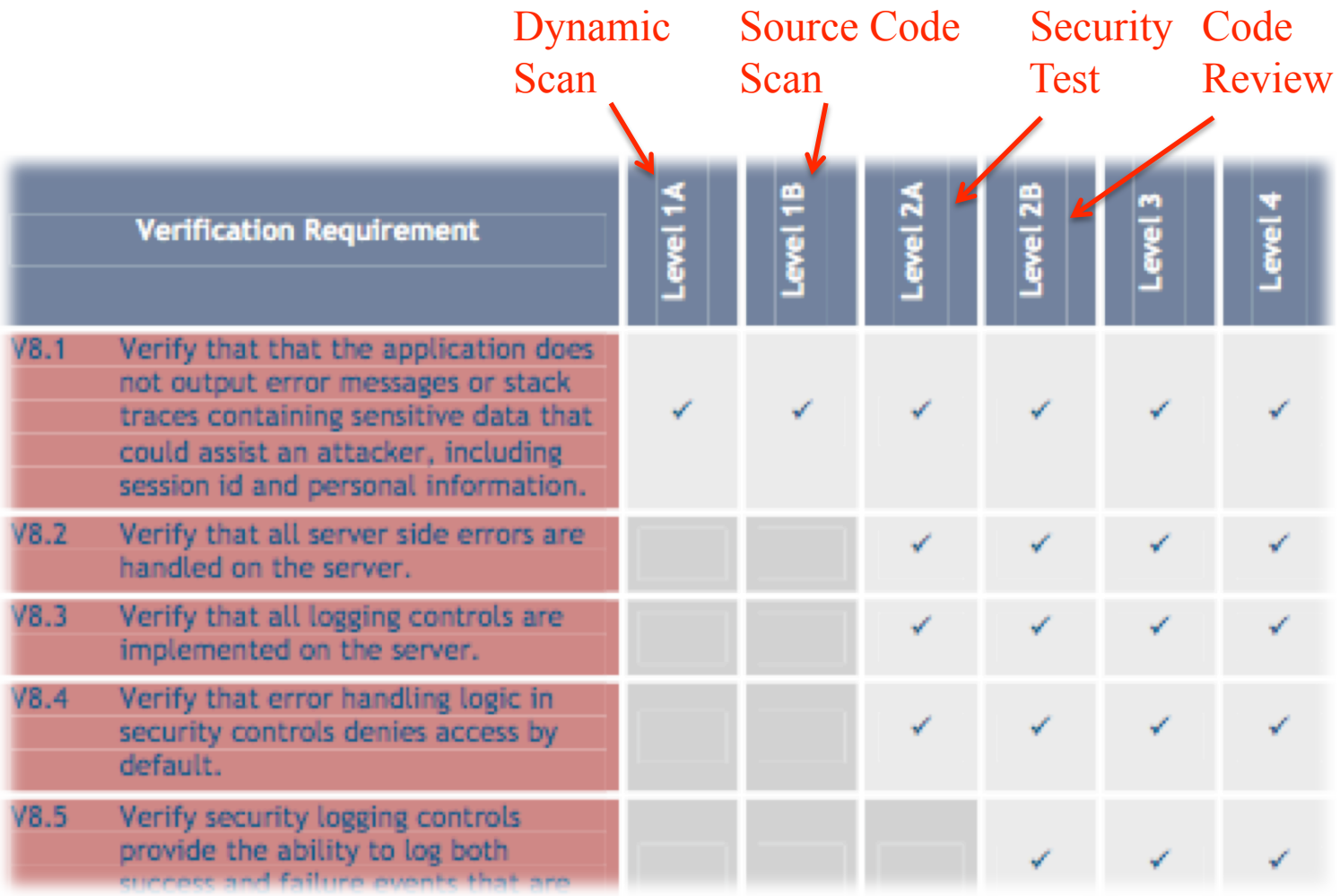
- High-level requirements
- Detailed requirements
- Reporting requirements

- *"Tools are an important part of every ASVS level. At higher levels in ASVS, the use of tools is encouraged. But to be effective, the tools must be heavily tailored and configured to the application and framework in use. And, at all levels, tool results must be manually verified."*

# ASVS Detailed requirements

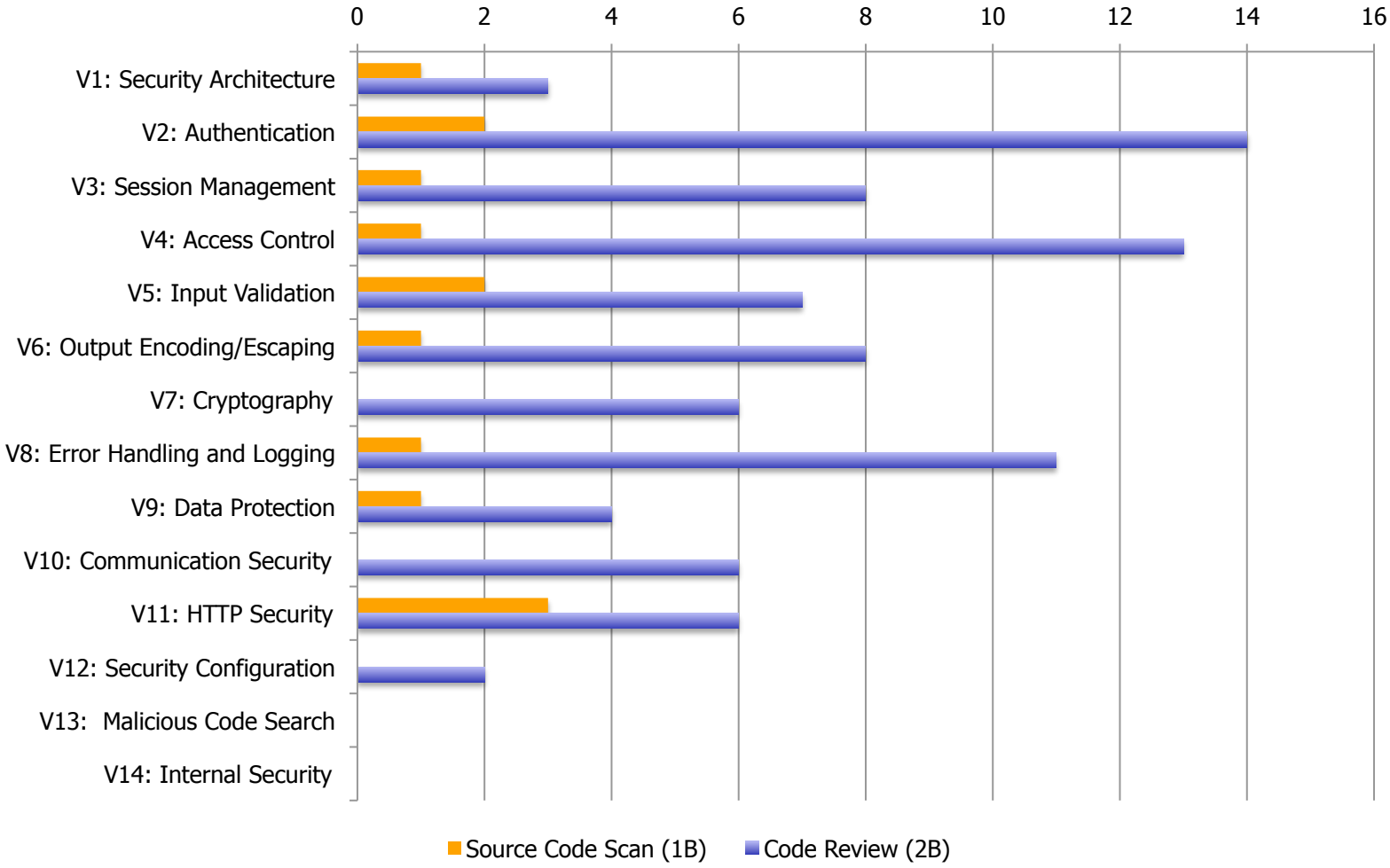
- V1. Security Architecture
- V2. Authentication
- V3. Session Management
- V4. Access Control
- V5. Input Validation
- V6. Output Encoding/Escaping
- V7. Cryptography
- V8. Error Handling and Logging
- V9. Data Protection
- V10. Communication Security
- V11. HTTP Security
- V12. Security Configuration
- V13. Malicious Code Search
- V14. Internal Security

# ASVS Verification Requirements Matrix




| Verification Requirement | | Level 1A | Level 1B | Level 2A | Level 2B | Level 3 | Level 4 |
|---|---|---|---|---|---|---|---|
| V8.1 | Verify that that the application does not output error messages or stack traces containing sensitive data that could assist an attacker, including session id and personal information. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| V8.2 | Verify that all server side errors are handled on the server. | | | ✓ | ✓ | ✓ | ✓ |
| V8.3 | Verify that all logging controls are implemented on the server. | | | ✓ | ✓ | ✓ | ✓ |
| V8.4 | Verify that error handling logic in security controls denies access by default. | | | ✓ | ✓ | ✓ | ✓ |
| V8.5 | Verify security logging controls provide the ability to log both success and failure events that are | | | | ✓ | ✓ | ✓ |

# Quasi-scientific quantitative matrix analysis



# Examples of what requirements CAN be verified using automatic code scan

- *[V5.2] Verify that a positive validation pattern is defined and applied to all input.*
- *[V6.1] Verify that all untrusted data that are output to HTML (including HTML elements, HTML attributes, javascript data values, CSS blocks, and URI attributes) are properly escaped for the applicable context.*
- *[V8.1] Verify that that the application does not output error messages containing sensitive data that could assist an attacker, including session id and personal information.*
- *[V11.2] Verify that the application accepts only a defined set of HTTP request methods, such as GET and POST.*

# Examples of what requirements CANNOT be verified using automated code scan

- *[V2.5] Verify that all authentication controls (including libraries that call external authentication services) have a centralized implementation.*

- *[V2.13] Verify that account passwords are salted using a salt that is unique to that account (e.g., internal user ID, account creation) and hashed before storing.*

- *[V4.4] Verify that direct object references are protected, such that only authorized objects are accessible to each user. (This can be of course checked dynamically.)*

# Examples of what CANNOT … (cont'd)

- *[V5.7] Verify that all input validation failures are logged.*
- *[V8.6] Verify that each log event includes: 1. a time stamp from a reliable source, 2. severity level of the event […] and 7. a description of the event.*
- *[V9.2] Verify that the list of sensitive data processed by this application is identified, and that there is an explicit policy for how access to this data must be controlled, and when this data must be encrypted (both at rest and in transit). Verify that this policy is properly enforced.*
- *[V14.2] Verify that security control interfaces are simple enough to use that developers are likely to use them correctly. (This is a level 4 requirement.)*

# Problems in automatic source code scan

- A static analyzer cannot step back and look at the big picture, e.g. architectural layers
- Evaluating non-functional security is almost impossible, e.g. robustness against DoS attack
- Logic flaws (e.g. in authorization) or missing security requirements cannot be detected
- Significant parts of the code may be missed completely, e.g. when in a different language or IoC/plugin code
- Configuration analysis may be problematic as well

# Mixing automation and manual work

- Manual code review on paper is pain!
- *Tools* are of great value, e.g.:
  - An IDE for traversing code (esp. jumping between caller and callee)
  - Grep or similar to quickly get pointers to interesting places and getting overview of technology used
  - Manual testing is good match for manual code review, and for that good tools (e.g. browser plugins) are essential

# From manual review to automation

- Build automated checks for manual findings
  - Doing this statically is not easy without proper tools
  - Dynamic approach may be easier, e.g. targeted automated scan or unit tests

# Conclusion

- When, where and how to use automated tools?
- Web portals implemented on a known and robust platform using systematic access control may be very good candidates:
  - Typical findings would be injection problems
  - Probably not much privacy or business assets to protect
- Complex business web application (e.g. extranet application) could be harder to verify:
  - Logical checks, privacy more delicate
  - Scalability and transactions to think of

# Conclusion

- Choice of verification methodology based on risk analysis
- Automatic code scan can give a rough measurement of a system even when run unconfigured
- Automatic code scanning is best combined with manual inspection, and/or as part of development build cycle
  - Scanner needs to be properly configured, though