

# Blind Security Testing

## An Evolutionary Approach

Black Hat USA 2007

Scott Stender

Vice President, iSEC Partners



**iSEC Partners**

<https://www.isecpartners.com>

**iSEC**  
PARTNERS

# Blind Security Testing

## *An Evolutionary Approach*

- Who are you?
  - Co-Founder and Vice President of iSEC Partners
  - Security consultant and researcher
  - Based in Seattle, WA
- Why listen to this talk?
  - Security, especially software security, is tied to testing
  - As software security improves, our testing methods must improve as well
  - This talk will be of interest for those who are involved in creating security test tools
  - No high-profile bugs today, but you will be better able to find your own!

# Blind Security Testing

## *An Evolutionary Approach*

- **Blind Security Testing**
  - Blind testing is useful in several areas: baseline testing, audit, closed systems...
  - The techniques here need only to interact with the system to be successful
  - More information is always better, grey/white box analysis techniques can make tests more efficient
- **An Evolutionary Approach**
  - Testing is difficult, and security testing is especially so
  - One problem I have encountered is test suite optimization
  - This talk proposes a method of competition between test classes and cases within those classes to optimize test suites

# Blind Security Testing

## *An Evolutionary Approach*

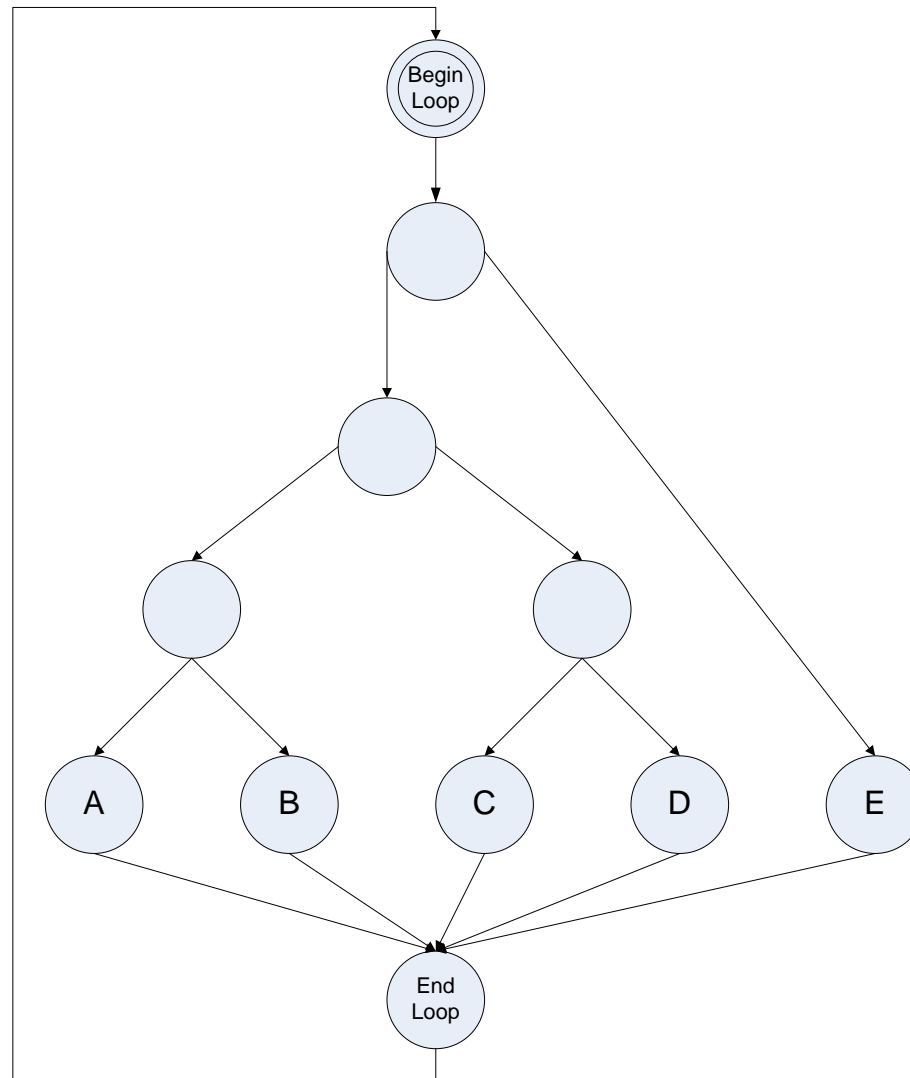
- **Background**
  - Problems Testing Software
  - The Need for Optimized Test Sets
- **Current Approaches**
  - Flaw-Specific Testing
  - Random Testing
  - Improved Heuristics
- **The Evolutionary Approach**
  - Test Cases as Populations
  - Test Case Organization and Competition

# Background

- Problems Testing Software
  - Even trivial applications can generate near-infinite test cases
- One Classic Example\*
  - Consider a program with 5 logic paths that is wrapped in a do...while loop
  - The loop is executed up to 20 times

\*Myers, Glenford. *The Art of Software Testing*

# Background



# Background

- Intractability of Testing
  - This simple example can be represented in about 10-20 lines of C code (or one of perl)
  - On each loop iteration, output will depend on the five output states
  - So...
    - Test with one iteration has  $5^1 = 5$  outputs
    - Up to two iterations has  $5^1 + 5^2 = 30$  potential outputs
    - Up to twenty iterations has  $5^1 + 5^2 + 5^3 + \dots + 5^{20}$  potential outputs
- This is approximately 100 trillion test cases!
  - At one test / sec, they would take 3.2 million years to run
  - Great coverage if we wait that long!
  - Even then, one still cannot say that the program is “correct”

# Background

- Security testing is even harder!
  - Myers' example was exercising functionality, something that has a chance of being finite (though large)
  - Security testing does not have that luxury
- Functional Security Testing
  - Verify authentication and authorization behavior
  - Verify proper use of cryptography for data protection
- Non-Functional Security Testing
  - Verify system cannot be compromised
  - Check for presence of current and as-yet-unknown classes of flaws

# Background

- Test For Buffer Overflows
  - Supply long strings: 128 bytes, 256 bytes, 65536 bytes...
  - Magic lengths:  $2^{32} - 1$ ,  $2^{32} - 2$ , ...
  - Off the wall: Off by one that happens to occur in 436 bytes
  - Pattern centric: First byte must be 0x1E, substring must match...
- And those others...
  - SQL Injection, XML injection, XSS, attacks against custom serialization...
- Don't forget the random fuzzing!
  - A truly infinite test set

# Background

- The Need for Optimized Test Suites
  - Based on testing only non-functional cases we have generated an infinite number of test cases
  - Let's just accept it now, comprehensive testing is impossible
  - A better goal: Optimized Test Suites
- Experienced security testers do this today
  - Consider testing a web application
  - First thing to try: type in that apostrophe
  - Second thing: see if “ZZZZZ” gets reflected in input
  - Why these over random data? They work
  - Let's see if we can automate the decision-making process

# Blind Security Testing

## *An Evolutionary Approach*

- Background
  - Problems in Testing Software
  - The Need for Optimized Test Sets
- Current Approaches
  - Flaw-Specific Testing
  - Random Testing
  - Improved Heuristics
- The Evolutionary Approach
  - Test Cases as Populations
  - Test Case Organization and Competition

# Current Approaches

- The Goal: Optimized Test Cases
  - We cannot execute everything
  - Let us execute what is most likely to cause flaws in the time available
- Most security testing tools pull from a similar pool of test cases:
  - Flaw-Specific Testing
  - Random Testing

# Current Approaches

- Flaw-Specific Testing
  - The goal is to identify specific, known classes of flaws
  - The approach: identify test data and expected results for security tests
- Consider the test suites for the following:
  - Buffer Overflow
  - Format String
  - Integer Overflow / Boundary Conditions
  - SQL Injection
  - Cross-Site Scripting
  - XML Injection
  - Command Injection
  - Encoding Attacks

# Test Classes

Buffer Overflow	A x 8	... 128	... 256	...65536		
Format Strings	%n	%s	%p	%x		
Integer Overflow	0	255*	65535*	2 <sup>x</sup> -1*		
XSS	“	‘	<	=	;	\
SQL Injection	‘	“	\	(	,	
XML Injection	<	&	“	=	[	
Command Injection	_	;		“	>	
Encoding	URL	UTF-*	B64	I18N		

# Current Approaches

- **Flaw-Specific Testing - Benefits**
  - They are surprisingly effective
    - Just consider the number of SQL Exceptions and EIP=41414141s you have seen!
  - They are easily prioritized over random input
    - If I know it is a managed web app, no test for buffer overflow
    - If I know they use dynamic SQL strings everywhere, test for SQL injection
- **Flaw-Specific Tests – Drawbacks**
  - They cannot find flaws other than those expected
    - Put another way, one could consider them a “local optima”
  - Even simple flaw-specific tests can take a prohibitively long time to execute (and still not test everything)

# Current Approaches

- Random Testing
  - The goal is to see how the system acts when subjected to random input
  - The approach: profile an application that is processing random input and watch for unexpected behavior
- Consider the sets of test cases for random testing:
  - Pure random data
  - Parameter-specific random data
  - Random mutations of legitimate data
    - Bit-flipping
    - Bitstream “sliding”

# Current Approaches

Random

Content

Length

Parameter-  
Specific

Content

Length

Mutation

Bit filp

Bit slide

# Current Approaches

- Random Testing - Benefits
  - They are surprisingly effective\*
    - Whether 1990 or 2007, apps fall to random data
  - They avoid the problem of “local optima”
- Random Tests – Drawbacks
  - Purely random attacks are horribly inefficient
    - Instead of local optima, we choose no optima
    - We luck out when test cases are cheap and apps are bad
  - Test results are hard to define
    - Application crashes are bad, but what about the variety of other errors that could indicate a problem?

\*B.P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities"

# Current Approaches

- Improved Heuristics
  - Simple heuristics and other evolutionary approaches can go a long way towards improvement
- Flaw-Specific Testing Improvements
  - Removing test cases based on equivalence classes
  - Advanced algorithms for test case verification
    - See Blind Exploitation Techniques for some great work here!
- Random Testing Improvements
  - Use of evolutionary algorithms with feedback based on debugging and/or code coverage
    - Sidewinder from BlackHat 2006
    - Evolutionary Fuzzing System from BlackHat 2007

# Blind Security Testing

## *An Evolutionary Approach*

- Background
  - Problems in Testing Software
  - The Need for Optimized Test Sets
- Current Approaches
  - Flaw-Specific Testing
  - Random Testing
  - Improved Heuristics
- An Evolutionary Approach
  - Test Cases as Populations
  - Test Case Organization and Competition

# An Evolutionary Approach

- Evolutionary algorithms work well in this problem space
  - They are best applied when trying to avoid local optima (as is the case with handcrafted Flaw-Specific Tests)
  - They can make sense of purely random data (as demonstrated by other researchers)
- First, a quick primer...
  - Evolutionary algorithms use biological selection as a model for computer systems
  - Potential solutions are considered from populations
  - Solutions are evaluated according to a fitness criteria
  - Better solutions are created based on the available populations and the fitness criteria

See Michalewicz, Z., and Fogel, D. *How to Solve It: Modern Heuristics*

# An Evolutionary Approach

- Evolution and Blind Security Testing
  - Instead of maximizing code coverage, optimize test sets
  - Use test case results as fitness criteria instead of code coverage or debugging
  - The goal: evolving an optimized test suite for a given request or application based purely on test feedback

# An Evolutionary Approach

- Test Case Organization and Competition
  - Need to define populations
  - Need to define fitness algorithms
  - Need to define next generation selection
- Population Design
  - An optimized test set is made up of several test cases, not just one case to rule them all
  - The problem breaks down according to two questions:
    - Which classes of test cases do we want to test?
    - Within those classes, which tests are most effective?
  - Think back to the manual optimization performed earlier:
    - Avoid buffer overflow testing for managed apps
    - Emphasize SQL injection testing when dynamic SQL used

# An Evolutionary Approach

- Populations as existing Test Sets
  - Start with populations for both Flaw-Based and Random test sets
  - Such populations can be created using traditional heuristics
  - Test sets (e.g. SQL Injection) and test cases (e.g. the apostrophe character) are evaluated for fitness
- Evolutionary competition between sets and cases
  - Test sets and test cases “compete” to be executed more often
  - One gets executed more often based on prior results



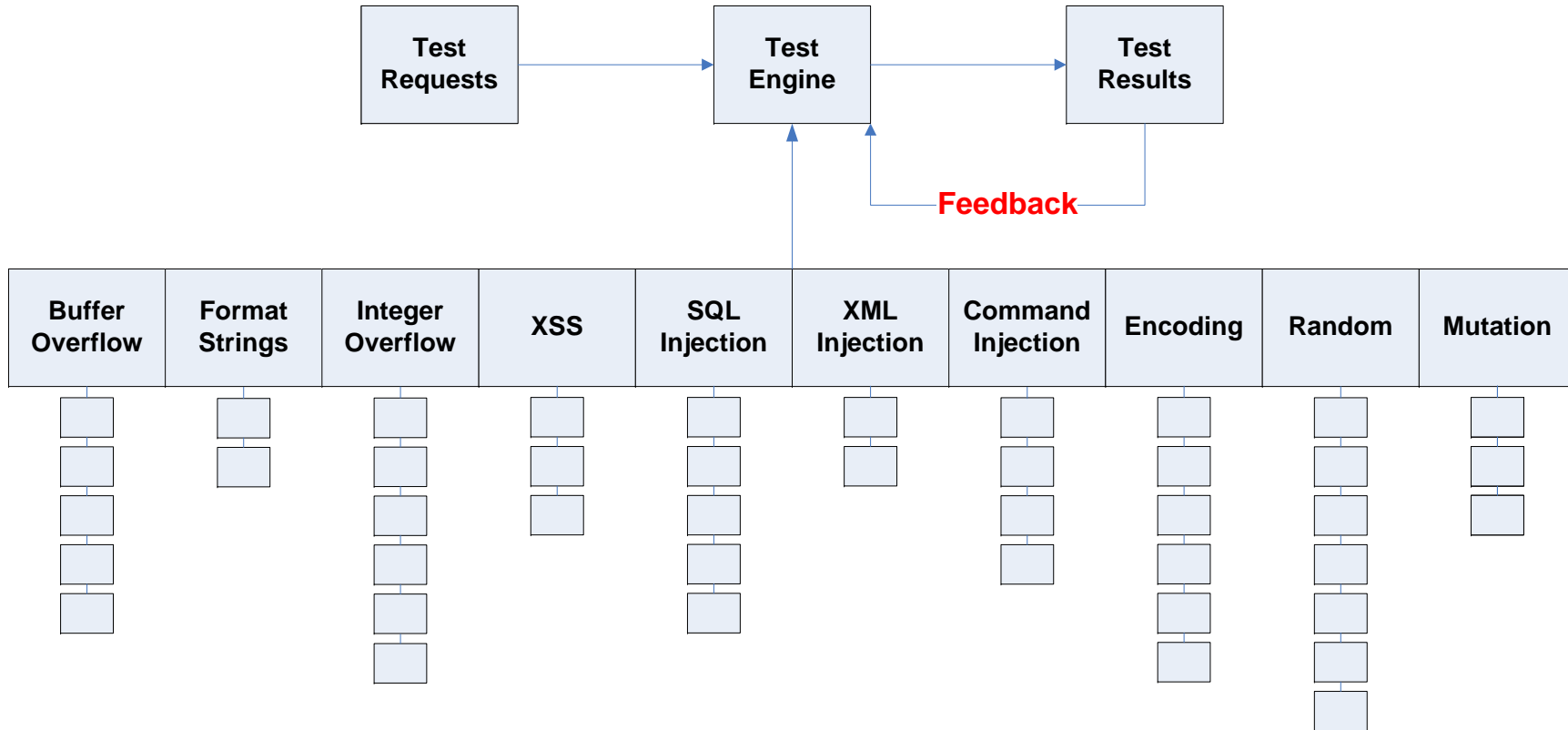
# An Evolutionary Approach

- Fitness Algorithm Design
  - The goal: make fitness accurate and determined by generally-available criteria
- Other approaches
  - System profiling via debugging or coverage is a natural choice
  - Code coverage and test set quality are often considered to be correlated
  - Downsides – not broadly available, and “code coverage = good test cases” is a controversial metric

# An Evolutionary Approach

- An alternative design
  - Use system feedback
  - Use natural properties of the test cases
- System Feedback Fitness Algorithms
  - Difference from control case offers meaningful feedback on the behavior of the test case
    - Magnitude of difference
    - Error detection within difference
  - For Flaw-Based test populations, sophisticated methods of error detection can be applied

# An Evolutionary Approach



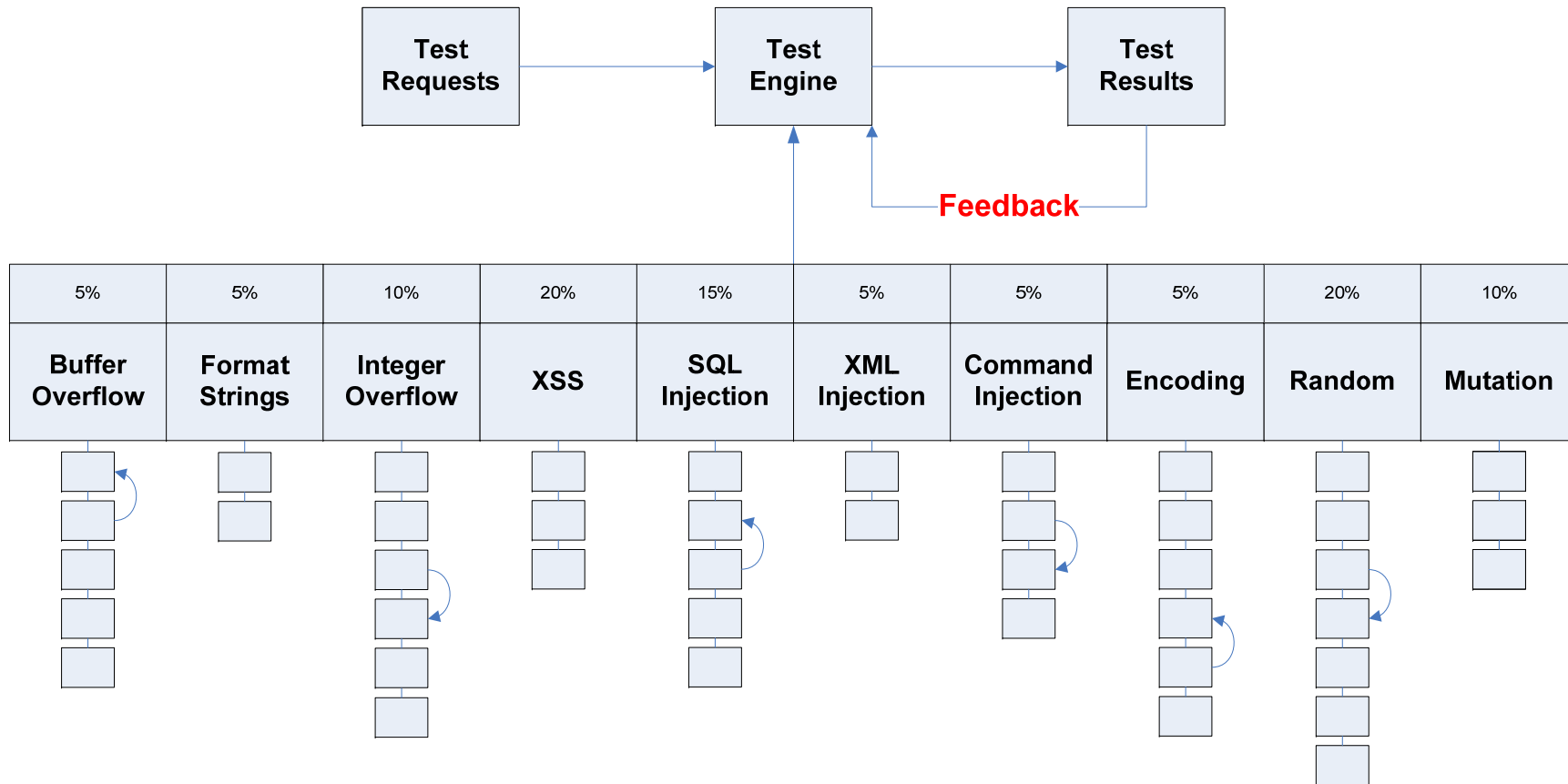
# An Evolutionary Approach

- Consider a web application
  - Capture a legitimate request without test data
  - Execute a test case against same request
  - Take a diff of the test vs. control
- Magnitude of change
  - Magnitude = sizeof(added) + sizeof(removed)
  - Effective and broadly applicable
    - System stops responding
    - System returns a stack trace
- Error detection
  - Check “added” portion of the diff for general and specific flaw evidence
    - Check for general error strings
    - Check for reflection of bad chars

# An Evolutionary Approach

- Choosing the next test
  - Fitness criteria adjusts the natural priority of test cases
    - Test class probability of execution is adjusted
    - Test case within the class is adjusted within the prioritized queue
  - Next test case execution takes this priority into account
- Note that we never remove a case or class
  - Remember one of the original goals – avoid local optima
  - If a case isn't initially successful, we want to leave the option open to come back
- The end result
  - Cases compete for execution time
  - Better cases move to the top

# An Evolutionary Approach



# An Evolutionary Approach

- Test Set Stability
  - This approach assumes that applications, as a whole, will share common programming styles and therefore failures
  - If this is not the case, you could “thrash” between test classes
- One option – reduce the temperature
  - Test classes are not assigned absolute probabilities, just “scores” that determine probability
  - One can, over the duration of the test run, reduce the probability of test case flux
    - Similar to “reducing the temperature” in Simulated Annealing
  - This allows enough data to make a reasonable test set, but avoid case-by-case thrashing

# An Evolutionary Approach

Using traditional test populations and fitness algorithms, we produce an optimized test set

- Benefits of this approach:
  - Broadly applicable to a number of systems
  - Does not require interactive control on the process being tested
- Drawbacks of this approach:
  - Code coverage and debugging are great sources of data
  - Using pure blind techniques will require significantly more test cases to make meaningful sense out of purely random test populations
  - Test cases are not optimal or comprehensive, just optimized

# An Evolutionary Approach

## Next Steps

- Improved fitness criteria
  - Using code coverage/debug data
  - Using log analysis
- Improved “breeding”
  - Smart optimization of pure random data cases
  - Splicing and joining of test set populations
- Stateful tests
  - Improve test execution ordering in addition to data

# Special Thanks

- For generously offering their time and feedback on the ideas presented in this presentation:
  - Brad Hill, iSEC Partners
  - Andreas Junestam, iSEC Partners
  - Tim Newsham, iSEC Partners

They are too many to list on one slide, but a special thanks to the researchers cited in this slide deck and in the associated handouts. One does not get far without standing on the shoulders of giants.

# Blind Security Testing

*An Evolutionary Approach*

## Questions and Answers

[scott@isecpartners.com](mailto:scott@isecpartners.com)