Logical Fuzzing

Richard Johnson | switech@microsoft.com

Welcome

- Introduction
- Agenda
  - The Business of Fuzzing
  - Fuzzing Technology
  - Architecting a Framework
  - Bennu Concept Tool

Fuzzing As We Know It

- Fuzzing is a method of software testing
- A high volume of *exceptional data* is sent to various interfaces of a target to locate faulty program logic
- Simple in concept, complex in practice
  - Hundreds of fuzzers have been written
- Fuzzing has held up in practical testing
  - Many thousands of bugs have been identified

From a Business Perspective

Identifying flaws in software is critical to the reliability and security of our information systems

Security critical bugs are very expensive to fix in deployed products

Fuzzers produce repeatable results useful for regression testing

Fuzz testing is part of the SDL best practices

Fuzzers are very cheap and very

Fuzzers are responsible for 70% of the bugs Microsoft patched in 2006

Fuzzers are responsible for the majority of the "month of" bugs

Fuzzers are responsible for the IFRAME bug, the .printer bug, etc

Fuzzers are responsible for
.printer bug, etc

Comparing Methodologies

- Manual Data Flow Analysis
  - Can be performed on any form of code
  - Produces an undefined number of bugs
  - Manual efforts are not repeatable or scalable
  - Very expensive and limited source of engineers
- Static Data Flow Analysis
  - Can target classes of bugs
  - Automated and repeatable
  - High false positive rate
  - Lacking effective algorithms
- Dynamic Data Flow Analysis
  - Can target classes of bugs
  - Automated and repeatable
  - Solves some problems with static analysis
  - Lacking effective algorithms*

```
int main ( int argc, char
**argv )
{
   FOO_STRUCT foo;
...
   foo.val = strdup(argv[1]);
   foo.sz  = strlen(foo.val);
...
   vuln(&foo);
}

void vuln ( struct *foo )
{
   char buf[STATIC_SIZE];
...
   strncpy(buf, foo->val, foo-
>sz);

}
```

Fuzzing Technology

Initial Public Offering

- Barton Miller, et al "An Empirical Study of the Reliability of UNIX Utilities", 1990
- Introduced "fuzz", the first dumb fuzzer
- Fuzzed with unstructured, rar
- Targeted command line argu utilities in 7 UNIX varieties
- Results: 25% – 33% of the utilities tested crashed, depending on the version of UNIX

> "Our approach is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability."

Initial Public Offering

- Miller tried again in 1995 with improvements
  - X Windows clients
  - Network ports
  - Memory exhaustion simulation
- Crashed as many as 40% of the console utilities and 25% X windows clients
- None of the network facing code faulted

> "Our 1995 study surprised us ... the continued prevalence of bugs in the basic UNIX utilities seems a bit disturbing. The simplicity of performing random testing and its demonstrated effectiveness would seem to be irresistible to corporate testing groups."

Valuable Input

- Miller, inspired by the storm, used random input data
- *Mutation* based input performs transformations on existing protocol data
- Static lists of values are used to target common implementation defects and known classes of bugs

Smarter Fuzzing

- Fuzzing interfaces with unstructured inputs will yield limited results
- Structured inputs allow for more effective traversal of program states
- This is where the art of fuzzing begins

You be the Smart, I'll be the Fuzz

- SPIKE, Dave Aitel, 2002
  - C language API for data generation and rapid network client development
  - Structured data dynamically defined as blocks
  - Relation model for size fields

- Peach Fuzzer Framework, Michael Eddington, 2004
  - Object oriented python API
  - Improved block based analysis with an abstracted fuzzing model

You be the Smart, I’ll be the Fuzz

- Peach Fuzzer Components
  - Generators
    - Primitive or complex block data generators
  - Transformers
    - Static encoders or decoders associated with a generator
  - Protocols
    - State logic is implemented using generators
  - Publishers
    - Provide a transport for the target protocol

Meanwhile in Academia

- PROTOS, 2002

Functional fuzzing using behavior models

Master Specification

- *BNF* notation utilized to describe *interaction models* and *syntax models*

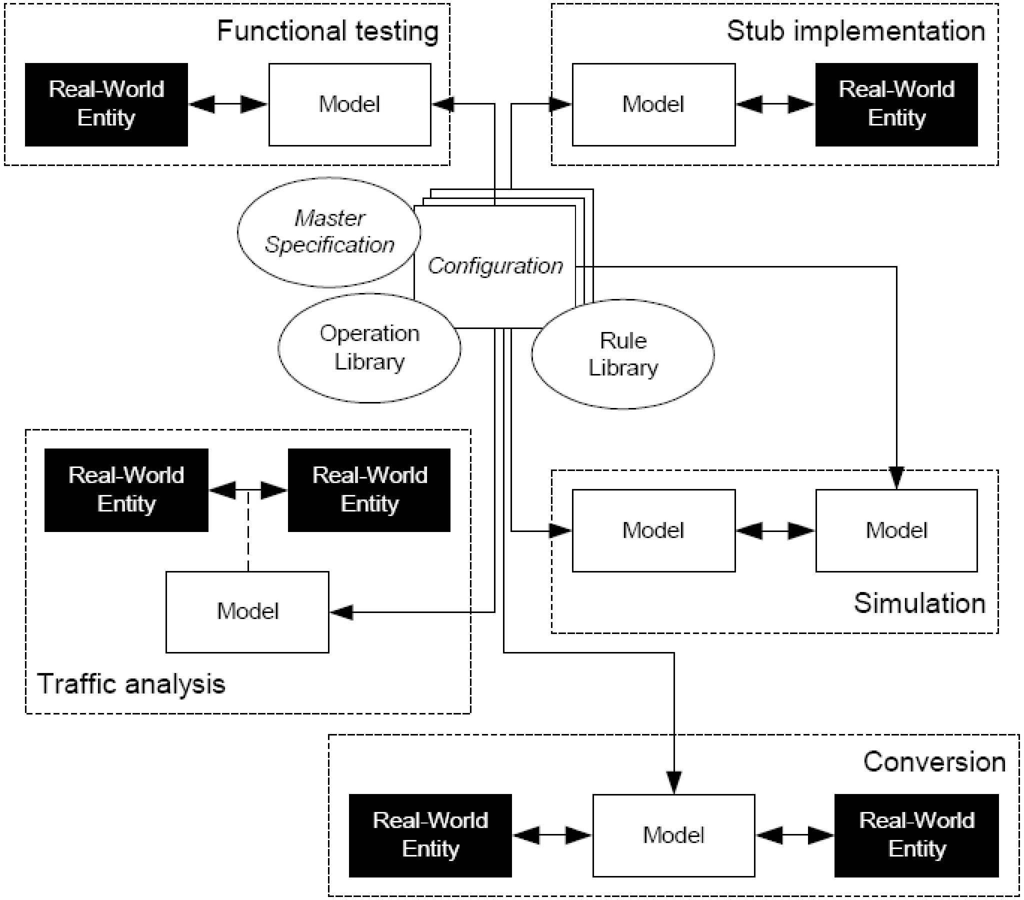
Configuration

- Performs *operations* on the master specification to derive a Mini-Simulation model

Communication Rules

- Connect the model to execution environment

PROTOS Mini-Simulation Concept



"A Functional Method for Assessing Protocol Implementation Security", Rauli Kaksonen

Meanwhile in Academia

- Entity Modeling
  - Describes internal behavior of an entity
  - Standards
    - Specification and Description Language (SDL)
    - Unified Modeling Language (UML)
- Interaction Modeling
  - Describes behavior between two entities
  - Standards
    - Unified Modeling Language (UML)
    - Tree and Tabular Combined Notation (TTCN)
    - Message Sequence Chart (MSC)
- Syntax Modeling
  - Describes the structure of data exchanged by entities
  - Standards
    - Abstract Syntax Notation One (ASN.1)
    - Extensible Markup Language (XML)

Behavior Modeling

PROTOS Mini-Simulation Behavior Grammar (TFT

Backus-Naur Form (BNF)

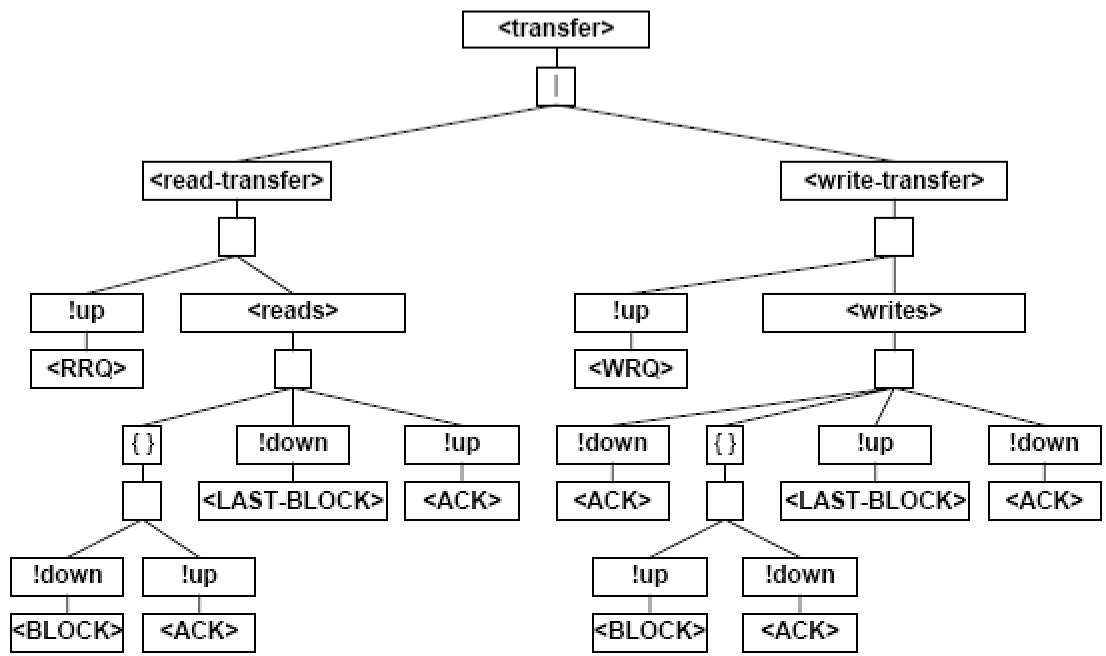
- Flexible context-free grammar extension to regular expressions
- Lacking standard notation

```
<transfer> = <read-transfer> |<write-transfer>
<read-transfer> = !up<RRQ> <reads>
<write-transfer> = !up<WRQ> <writes>
<reads> = {!down<BLOCK> !up<ACK>} !down<LAST-BLOCK> !up<ACK>
<writes> = !down<ACK> {!up<BLOCK> !down<ACK>} !up<LAST-BLOCK> !down<ACK>
```

Simulation Grammar

- Attribute grammar using modified BNF notation
- Tree-based *Data Productions*
- *Tags* represent callbacks such as *input triggers*

PROTOS Mini-Simulation Behavior Tree (TFTP)



Syntax Modeling

PROTOS Mini-Simulation Syntax Grammar (TFTP)

Syntax Grammar

- Also uses modified BNF
- Tree-based *Type Productions*

Evaluation

- Transforms input grammar to output grammar
- Engine traverses input tree, executing *rules* on subtrees
- *Semantic Rules* evaluate data
- *Communication Rules* implement I/O

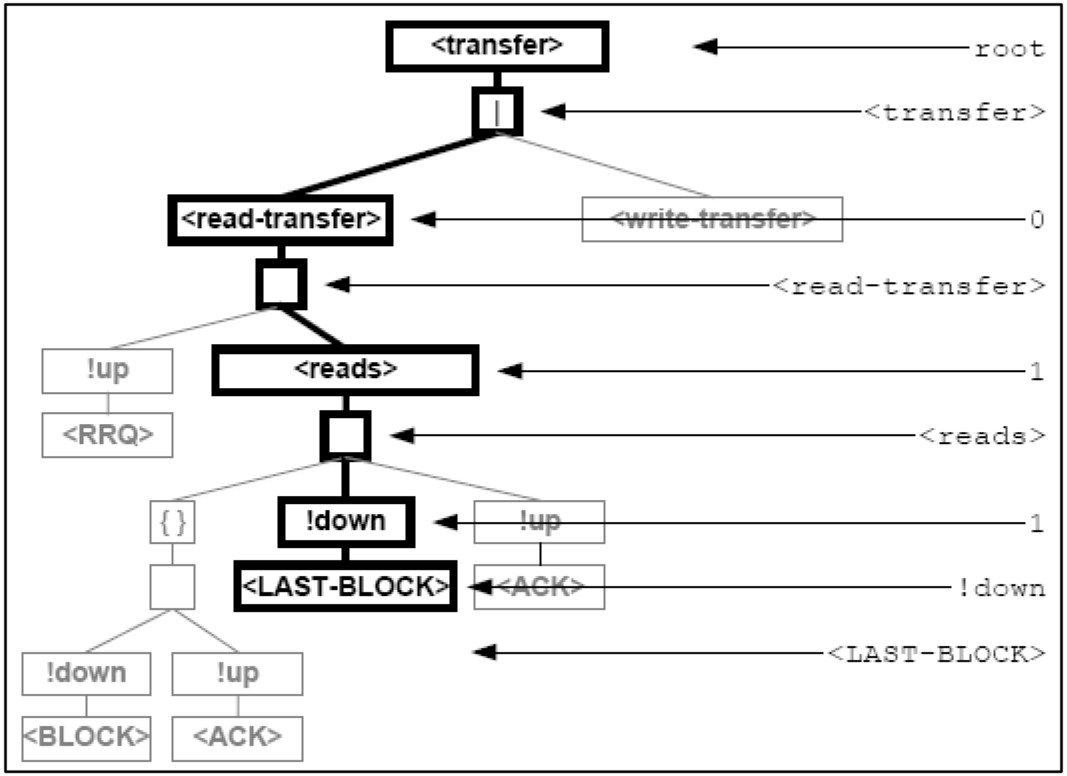
```
# Request PDUs
<RRQ> ::= (0x00 0x01) <FILE-NAME> <MODE>
<WRQ> ::= (0x00 0x02) <FILE-NAME> <MODE>

# Subsequent PDUs
<BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 512 x <OCTET>
<LAST-BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 0..511 { <OCTET> }
<ACK>  ::= (0x00 0x04) <BLOCK-NUMBER>
<ERROR> ::= (0x00 0x05) <ERROR-CODE> <ERROR-MESSAGE>

# Miscellaneous productions
<MODE> ::= "octet" 0x00 |"netascii" 0x00
<FILE-NAME> ::= { <CHARACTER> } 0x00
<BLOCK-NUMBER> ::= <OCTET> <OCTET>
<ERROR-CODE> ::= <OCTET> <OCTET>
<ERROR-MESSAGE> ::= { <CHARACTER> } 0x00
<CHARACTER> ::= 0x01 - 0x7f
<OCTET> ::= 0x00 - 0xff
```

State Traversal

PROTOS Mini-Simulation Path Representation

Path Finding

- Paths are used to access elements of the grammar
- Masks can be used as an optimized path representation



`<transfer>.0.<read transfer>.1.<reads>.1.!down.<LAST-BLOCK>`

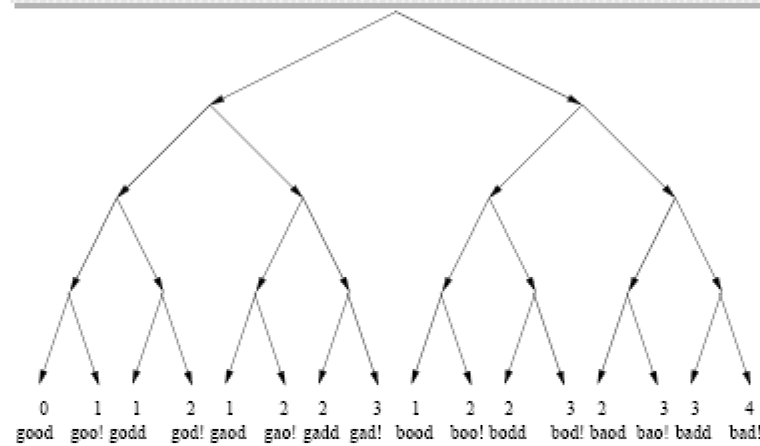
Dynamic Whiteboxing

- Scalable, Automated, Graph Executution (SAGE)

"Automated Whitebox Fuzz Testing", 2006

  - Runtime state of a recorded session is stored for analysis
  - Symbolic execution gathers input constraints from conditional statements
  - Solution given by known-good input data is negated and solved again
  - *Generational* vs *Depth-First Search* (DFS) algorithms

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == ‘b’) cnt++;
   if (input[1] == ‘a’) cnt++;
   if (input[2] == ‘d’) cnt++;
   if (input[3] == ‘!’) cnt++;
   if (cnt >= 3) abort();
}
```



What’s Missing?

- Abstraction
  - Existing behavior model research is not being utilized
- Automation
  - Current technology not fit for production use
  - Manual processes introduce inconsistent results
- Unification
  - Commonalities in desired functionality have not been assessed
  - Lack of a common platform prevents useful integration of existing research tools

Architecting a Fuzzing Framework

Fuzzer Engines

- *Fuzzer Engines* can be classified by features:
  - Input Generation
    - Random or Mutation or Static
  - Data Model
    - Unstructured or Structured
  - Behavior Model
    - Stateless or Stateful
- The desired platform should support the creation of both simple and complex fuzzers

A Note About Input Generation

- Reproducibility is crucial
- Multiple passes of data generation is ideal to target known classes of bugs first
- Fuzzers should be able to run for an infinite time but cover the critical space quickly
- Extended model for generation sequencing would be ideal

Fuzzer Development Phases

Target Profiling

- Manual Analysis
  - Protocol Specifications

- Static Analysis
  - Type and Symbolic Debug information
  - Execution Flow Graphs
  - Data Flow Graphs

- Dynamic Instrumentation
  - Interface discovery
  - Indirect execution and data flow

- Sample input data
  - File harvesting
  - Traffic Analysis

Data Modeling

- Notation for behavior modeling should be abstract enough to represent both data and behavior
- ASN.1 is cumbersome and not human readable, and cannot model behavior.
- PROTOS's modified BNF grammar looks highly capable
- XML serialization is widely supported making it a good option

Behavior Modeling

- PROTOS interaction model is robust and useful
- New research is on-going in using XML to represent state models
  - “XML Graphs in Program Analysis”, Anders Møller, et al
  - GXL Schema

Testing and Analysis

- Target Instrumentation
  - Debugger Engine
- Logging
  - Callbacks and Exception Handling
- Result Analysis
  - Analysis using standard debugging Tools
  - Visualization for manual analysis

Bennu: A Concept Tool

id=13824

Bennu Goals

- State of the Art
  - Identify and use the best research concepts available for fuzz testing
- Flexible & Reusable
  - Framework should be able to be used to create any of the types of fuzzers in common use today
  - New fuzzers should have access to previous models
- Intelligent
  - Use profiling information when present
  - Do not *require* any special information to execute

Bennu Goals

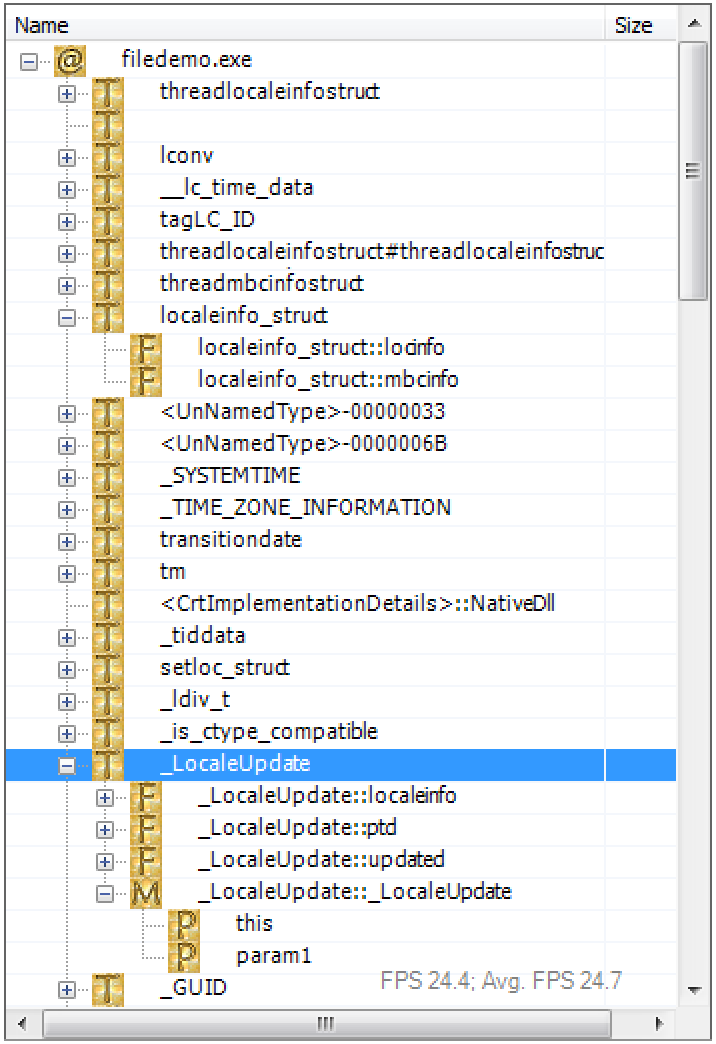
- Approachable
  - Users should not need to write much code or understand how internal models work
- Customizable
  - Target Profiling and Testing Analysis should be pluggable
- Scalable
  - Distributed testing should be possible

Assisted Target Profiling

- Static analysis engine powered by Phoenix*
  - Symbols
  - Types
  - Imports
  - Control Flow
  - Data Flow

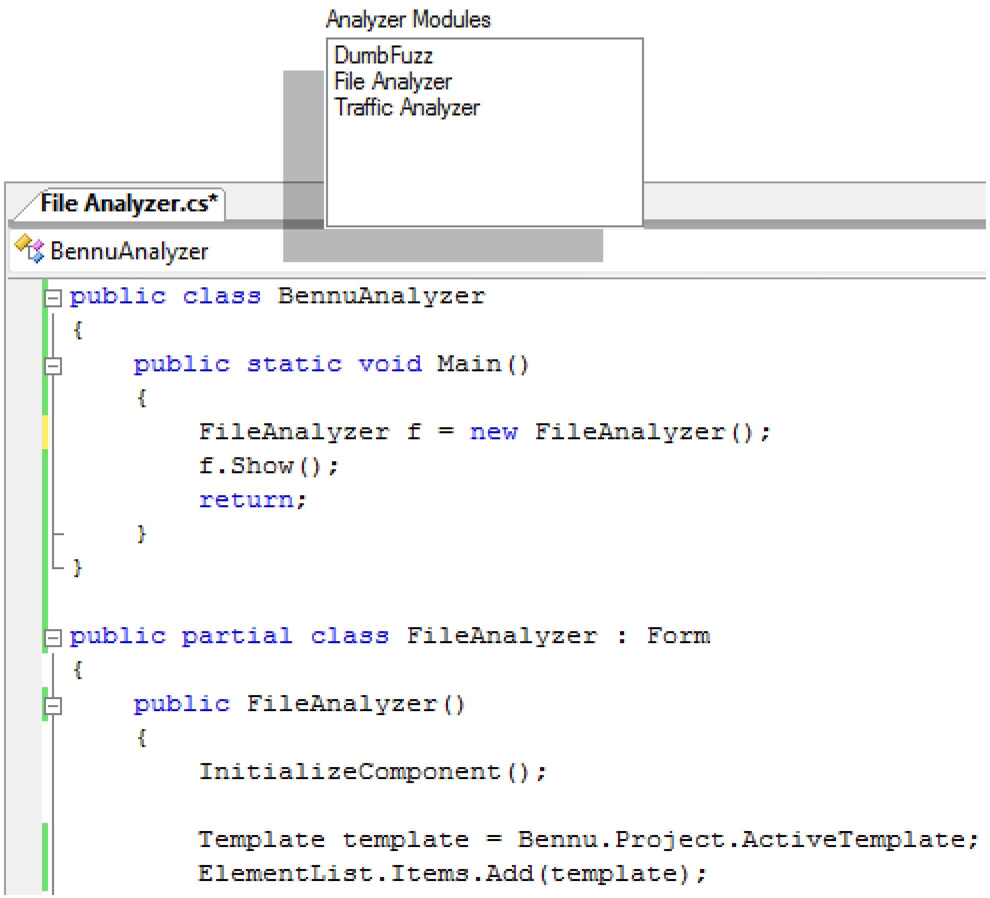
- Dynamic analysis engine powered by Microsoft Debug Engine (dbgeng.dll)

- Run-time compiled Target Analyzers written in C# perform analysis functions with the static and dynamic engines



Assisted Target Profiling

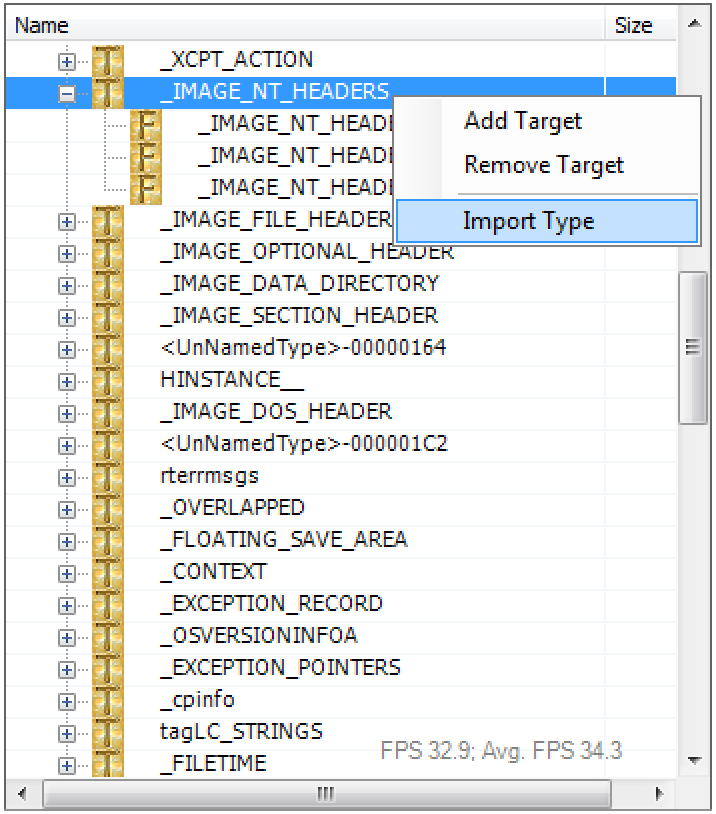
- Static analysis engine powered by Phoenix*
  - Symbols
  - Types
  - Imports
  - Control Flow
  - Data Flow
- Dynamic analysis engine powered by Microsoft Debug Engine (dbgeng.dll)
- Run-time compiled Target Analyzers written in C# perform analysis functions with the static and dynamic engines



Assisted Data Modeling

- XML Data Model
  - Structured template definitions
  - Type specification
  - Extended relationship model

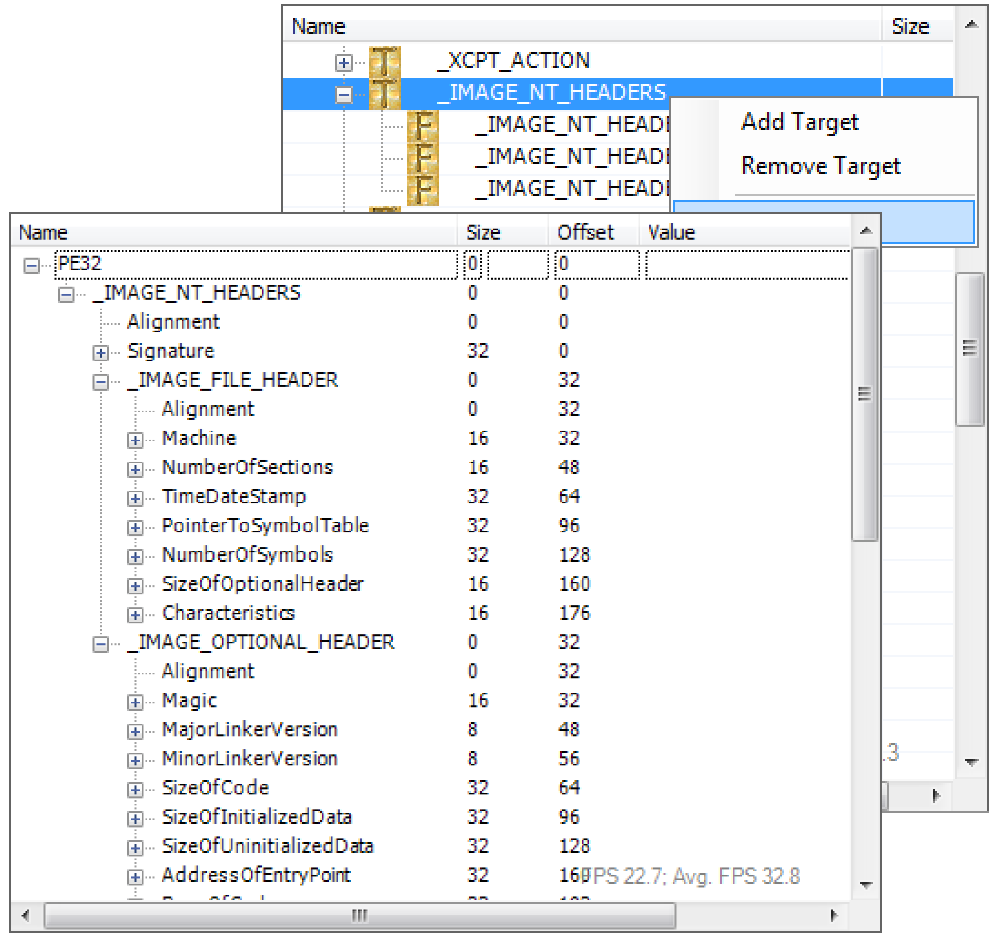
- Developed in cooperation with Mike Eddington, supported by Peach 2.0



Assisted Data Modeling

- XML Data Model
  - Structured template definitions
  - Type specification
  - Extended relationship model

- Developed in cooperation with Mike Eddington, supported by Peach 2.0



Assisted Behavior Modeling

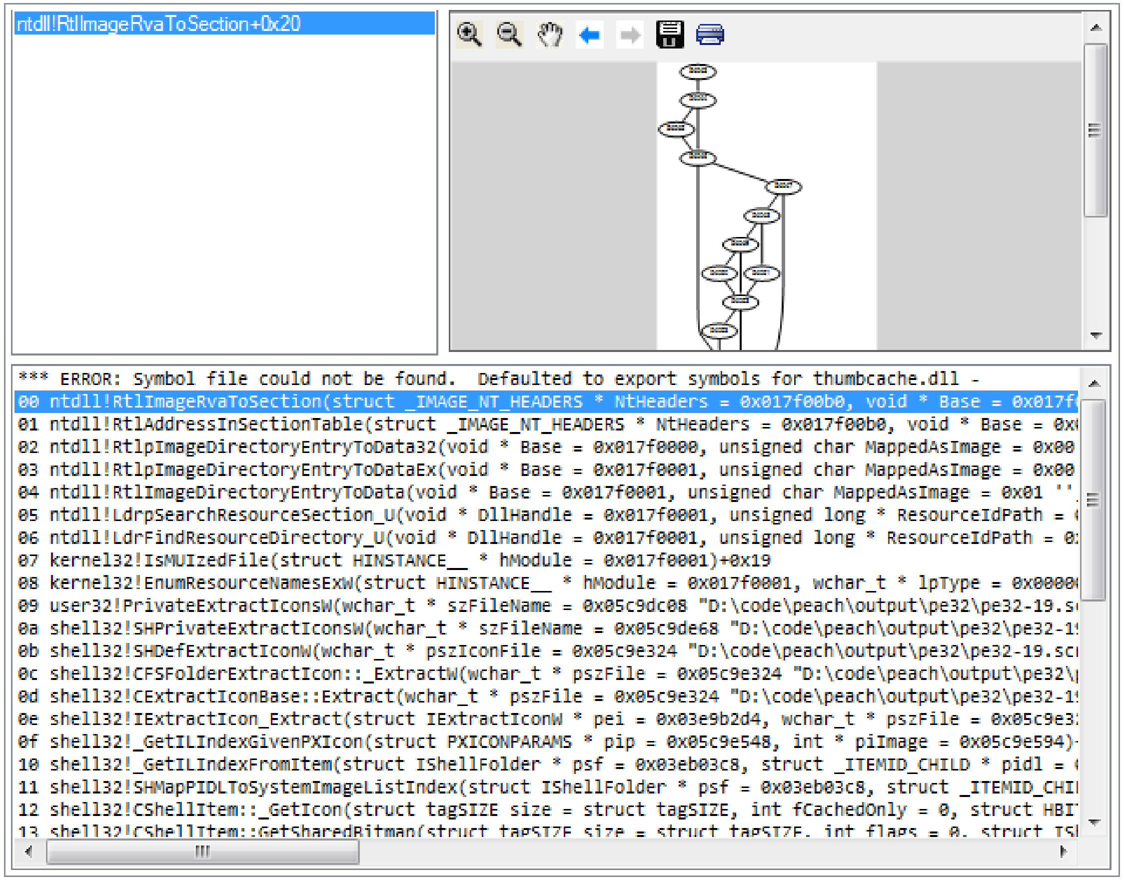
- XML Model
- Evaluations use callbacks
- State model abstraction currently being developed

UNDER DEVELOPMENT

- Developed in cooperation with Mike Eddington, supported by Peach 2.0

Automated Testing and Analysis

- Tests executed by Peach 2.0 running on an embedded Python engine
- Exception handling and post-run analysis using the Dynamic Analysis Engine
- Quickly inspect minidump contents
- View visited code blocks
- Register callbacks for automated post-run analysis



Conclusions

- Fuzzing is an increasingly powerful approach to software security
- Available support libraries are sufficiently robust to build complex analysis frameworks
- Academic research has revealed technology possibilities that have yet to be fully realized
- Automating the abstraction of behavior models provide an ideal area of research for security engineers