Data Structure Reverse Engineering

Digging for Data Structures
Polymorphic Software with DSLR

Scott Hand

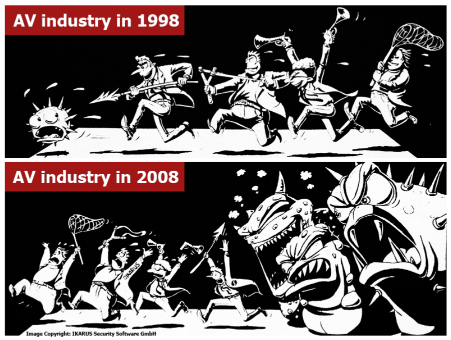
October 28th, 2011

Outline

Outline

Outline

The Current Situation



The Current Situation

What's Happening?

- Effectiveness of AV solutions not what it used to be
- Some are calling for dissolution of AV industry (Source)
- Lots of botnets

Why?

- Signature checking just greps for patterns
- Weak against obfuscation
  - Packing
  - Code polymorphism
  - Junk bytes

The Current Situation

Example of AV Weakness

- Whitepaper published by SANS institute examined efficacy of AV apps in detecting Metasploit payloads.
- Obfuscation on payload that was detected by 14 out of 32 AV engines led to its detection by only 4 out of 32 engines.
- This was only on the Windows platform. Linux AV tools failed 100% of the time.

Motivation for New Approach

Why Data Structures?

- Previous weaknesses focused on problems with code matching approaches
- Such obfuscation attempts change code but maintain abstractions
- Maybe we can find a way to look for patterns in those abstract structures...

Outline

Description of Data Structure Approach

Some Properties

- All programs use data structures
- These data structures are abstractions of implementation details
- The data structures used tend to be very similar between programmers

Approach:

We can try to look for general compound data structures.

Laika Overview

Key Challenges

- Identify position and size of objects
  - Use potential pointers in image to estimate object positions and sizes
- Determine which objects are similar
  - Convert objects from sequences of raw bytes into sequences of semantically valued blocks
  - "Probably pointer blocks", "probably string blocks", etc
  - Cluster objects with similar sequences of blocks using Bayesian unsupervised learning

Laika Overview

Empirical Approach

1. Built a virus checker on top of Laika
2. Check against conventional scanners
3. Results
   - Laika has 99% accuracy
   - ClamAV has 85% accuracy

Outline

Classification

- Input is a set of unknown objects
- Identifies distinguishing features (feature selection problem)
- Train a classifier
- Make inferences about the class of each object
- Output is set of objects with tagged classes

Types of Classification - Data Tagging

- Supervised Learning
  - Inference engine is trained on labeled data with a set of given classes. Easier, more effective, simpler to validate. Labeled data not always possible.
- Unsupervised Learning
  - Inference engine is given data set and asked to generate a set of classes. Engine finds number of distinct classes and tags items accordingly

Types of Classification - Underlying Learning Method

- Generative - Learning machine attempts to learn an underlying probability distribution. This is helpful because probabilistic methods such as *expectation maximization* (or its Bayesian counterpart *maximum a posteriori*) become available to use.
- Discriminative - Learning machine attempts to learn the best way to determine class boundaries. This is often more specialized and data efficient at the cost of flexibility.

Feature Selection

Nature of the Problem

- Feature selection is the most important part of designing any classifier
- Often independent of classification method
- Especially hard for this problem, as objects from same class will still often have completely different byte values

Block Types

- Convert each machine word into a *block type*
- Basic types:
  - Address
  - Zero
  - Char
  - Data

Feature Selection

Atom Types

- Classes are represented as vector of atoms
- Atoms are a collection of blocks, so we need to identify atoms from block streams
- Basic atom types:
  - Pointer
  - Zero
  - String
  - Integer
- Looks like there's some relation between atom and block types...
- A block type is an atomic type with some error. This can be observed by examining $P(\textit{blocktype}|\textit{atomictype})$

Finding Data Structures

Basic Process

1. Scan through memory and identify pointers
2. Tentatively estimate the start position of objects using locations from pointers
3. Find the end position using estimation done during clustering
4. The rest of the block past the end of the object is classified as random noise
5. Introduce a random atomic type to handle this noise

Heuristics

Exploiting Malloc

- Example: Using the Lea allocator in GNU libc leaks chunk size information.
- More general: Most malloc algorithms keep similar objects in the same area of memory.
- Extremely effective, but does not improve Laika's accuracy. Why?
- Laika's similar size estimations leads to it already knowing that nearby objects are similar

Bayesian Model

- The lth machine word of memory image M is notated M_l.
- The kth atomic type of class j is ω_{jk}.
- X is the input list, with X_i indicating the position ith object in X.
- We want to maximize the most likely objects and classes given a memory image. An equation for this can be obtained with the following steps:
  1. Bayesian approach means MAP. We can get this from Bayes' rule as $P(\Theta|X) = \frac{P(X|\Theta)P(\Theta)}{P(X)}$.
  2. Plugging in the values specific to this problem we get $P(\omega, X|M) = \frac{P(M|\omega,X)P(\omega,X)}{P(M)}$.
  3. Applying the chain rule to the class and object joint distribution, we obtain $P(\omega, X|M) = \frac{P(M|\omega,X)P(X|\omega)P(\omega)}{P(M)}$.

Bayesian Model

- Our normalizing constant $P(M)$ can be dropped as we only care about the likelihood, not the probability
- We assume independence both between and within classes
- This lets us calculate the prior distribution easily as $P(\omega) = \prod_j \prod_k P(\omega_{jk})$.
- $P(X|\omega)$ represents the probability of locations and sizes of the list of objects based on our class model. The term is 0 for illegal solutions and 1 otherwise. $P(M|\omega, X)$ represents the model's fitness for the data. This can be calculated as $P(M|\omega, X) = \prod_l P(M_l|\omega, X)$
- The previous method of calculating the likelihood equation makes a Naive Bayes assumption; it assumes data is conditionally independent to other data.

Bayesian Model - Putting It All Together

- As stated, the probability being sought is $P(\omega, X|M) \propto P(M|\omega, X)P(X|\omega)P(\omega)$
- Substituting the prior distribution yields $P(M|\omega, X)P(X|\omega)\prod_j \prod_k P(\omega_{jk})$
- Taking the function $\delta : X \times \omega \to \{0, 1\}$ returning 0 for illegal solutions and 1 otherwise, adding in the list suitability factor yields $\delta(X, \omega)P(M|\omega, X)\prod_j \prod_k P(\omega_{jk})$
- Finally, adding in the model fitness factor yields $P(\omega, X|M) \propto \delta(X, \omega)\prod_l P(M_l|\omega, X)\prod_j \prod_k P(\omega_{jk})$

Bayesian Model - Final Equation

Maximize:

$P(\omega, X|M) \propto \delta(X, \omega)\prod_l P(M_l|\omega, X)\prod_j \prod_k P(\omega_{jk})$

Intuition

- First term does sanity checking
- Second term penalizes Laika for putting an object into an unlikely class and makes sure the solution reflects the particular memory image
- Third term encourages simple solutions by penalizing approaches with many classes

More Optimizations

Typed Pointers

- Simple pointer/integer classifications produce reasonable results, but we can further optimize by introducing typed pointers
- If all instances of class have a pointer at the same offset, it's likely that the targets of those pointers share a class
- Good for small classes and objects with no pointers
- Increases computational complexity. Breaks our previous independence assumptions
- Will cause small errors to propagate

One More Consideration...

Dynamically-Sized Arrays

Not all classes have feature vectors of the same size. We will allow objects to wrap around modulo the size of a class. This means an object can be classified as a contiguous set of instantiations of a given class - an array.

Implementation

Code Details

- Done in Lisp
- Unsupervised learning is difficult
- Use approximation scheme based on computing $P(\omega, X|M)$ incrementally
- Uses typed pointers as a guiding heuristic

Empirical Method

- Used Gentoo Linux to build applications and libraries with minimal optimizations, debugging symbols
- Wrote a wrapper for *malloc* to track allocations and evaluate Laika's ability to identify them

Results

Data Structure Detection

- Mostly correct
- Some difficulties
  - Heap is extremely noisy
  - Only 30% of objects contained a pointer, remaining 70% classified by objects pointing to them
  - Poor software practices such as *tail accumulator array* in X Window data structure. Solution? Send X Window developers a dirty sock.
- Percentage of success was around 65% without malloc info, around 0.78 with malloc info.

Results

Program Classification

- **Agobot** - 99.4% (83% ClamAV)
- **Kraken** - 99.8% (85% ClamAV)
- **Storm** - 99.9% (100% ClamAV)

Outline

Analysis of Laika

Good

- At worst, it is defense in depth by posting malware authors different challenges
- At best, it can synergize with code analysis

Bad

- Won't work with large class of simple malware
- Much more resource intensive

Outline

Outline

How Can We Beat Laika?

Polymorphic Data Structures

- Randomize data structure layout
- Done during compilation
- Can evade Laika-style detection
- Can also help foil rootkit attacks

More about Data Structures

They are ubiquitous

- Network protocol reverse engineering (guided fuzzing)
- Buffer overflow attacks
- Kernel rootkits require knowledge of OS data structures
- Attack signatures (Laika)

A Few Things about Randomization...

Examples where it doesn't work

1. Not as helpful in network communication because other parties are involved
2. Public definitions cannot be randomized
3. Tail accumulator arrays rely on the zero-length array to be at the end of the data structure
4. Programmers may use direct data offsets to access some fields
5. When data structure order is used during value initialization

How to address this?

A Few Things about Randomization...

Similar issues

- Monoculture leads to large-scale reproductive attacks
- We should aim to embrace randomization
- Similar solutions:
  - Address space randomization
  - Instruction set randomization
  - Data randomization
- In a similar spirit, let's examine data structure layout randomization (DSLR)

Randomizable Data Structures

Mitigation steps

- Data structures are randomizable if and only if it is not exposed to external programs or does not violate *gcc* syntax or programmer intention
- We ask the programmer to indicate when a data structure is randomizable

Outline

A Simple Approach

Reorder Data Structure Layout

- Pretty straightforward
- Given n structures each with m fields, this gives $(m!)^n$ program combinations.
- Is it actually that simple?

Identical Layouts

- This occurs when reordering the data structure produces an isomorphic data structure
- Example: a data structure containing `int` followed by `int` is not changed by reordering
- Solution? Insert junk data into the data structure

Implementation with *gcc*

Possible places to do randomization

1. Abstract Syntax Tree (AST)
2. GIMPLE (representation with at most three operands)
3. Static single assignment (SSA) tree representation
4. Register-transfer language (RTL) tree

Reasons for choosing AST

- AST retains a lot of program source code information
- AST is easier to understand and more convenient to modify
- AST occurs before *gcc* has determined layout of data structures, so we can reorder data structure members without computing specific memory addresses.

Which Data Structures to Randomize

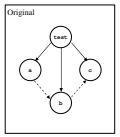
Our choices

- `struct`
- `class`
- Function stack variables

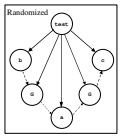
Example

```
struct test
{
    int a;
    char b;
    int *c;
};
```

(a)



(b)



(c)

DSLR Implementation in GCC

Four Key Components

1. Keyword Recognizer
2. Re-orderer
3. Padder
4. Randomization Driver

Keyword Recognizer

A few new keywords are introduced:

1. `__obfuscate__` - This lets *gcc* know that a data structure may be randomized. It is followed by some specific options.
2. `__reorder__` - This tells *gcc* to reorder the elements in a structure.
3. `__garbage__` - This tells *gcc* to insert garbage into the data structure.

Implementation Details

Reorder

- When generating AST for a program, *gcc* chains members of a data structure to a list.
- When finished, the reorder keyword is encountered and it uses a seed from the randomization driver to reorders the chain.

Padder

- Carried out in the process as reordering
- Size of garbage items is determined by randomization driver

Implementation Details

Randomization Driver

- We need to be able to ensure randomization consistency across a single project build
- Stores a random value (either from project build file or from the `glibc` function `random` then stored) and a count of the number of randomized fields.
- Reordering is done with a recursive Knuth shuffle
- Padding selects fields from sizes in the set $1, 2, 4, 8$.

Outline

Empirical Analyses

Effectiveness

- Apply to goodware and malware
- Goodware includes programs such as openssh
- Malware includes programs from offensive computing and VX Heavens
- Achieves a code difference between 3 and 17%.

Rootkit Defense

- Used DSLR to randomize the `task_struct` data structure in the version 2.6.8 Linux kernel
- Prevented 4 out of 6 rootkits tested

Empirical Analyses

Evaluation against Laika

- One small problem: Laika's released version only works on Windows binaries, DSLR uses *gcc*
- Had to manually execute the randomization methods
- Tried 3 Windows programs: agobot, 7-zip, and notepad. Laika could not process notepad, so proceeded with only the other two
- Worked well with agobot (used previously to demonstrate Laika)
- 7-zip was not quite as effective. Possible reasons include that 7-zip has lots of unrandomizable structures and that high library code usage. However, data structure analysis might not be a great idea when library usage is so high

Empirical Analyses

Performance Overhead

- Caused mainly by random value lookup, field count, and field reordering
- On average, only around 2% performance overhead to *gcc*
- Some applications were actually faster, possibly due to data locality improvements

Limitations and Future Work

- Does not support other languages such as Java, as it uses *gcc* at a language specific AST level.
- Randomizability of a data structure cannot be determined automatically
- Could use some other techniques such as struct and class splitting

Outline

Concluding Remarks

Any questions?

References

- Cozzie, Anthony and Stratton, Frank and Xue, Hui and King, Samuel T.
  Digging for data structures.
  Proceedings of the 8th USENIX conference on Operating systems design and implementation
- Lin, Zhiqiang and Riley, Ryan D. and Xu, Dongyan
  Polymorphing Software by Randomizing Data Structure Layout.
  Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.