Bibliography

[1] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*, February 2011. Cited on p. 4.

[2] Gogul Balakrishnan, Radu Gruian, and Thomas Reps. Codesurfer x86—a platform for analyzing x86 executables. 2005. Cited on p. 5.

[3] Daniel Bilar. Fingerprinting malicious code through statistical opcode analysis. 2007. Cited on p. 4.

[4] Christian Cachin. An information theoretci model for steganography. 1998. Cited on p. 5.

[5] Christina Cifuentes, Mike Van Emmerik, and Norman Ramsey. The design of a resourcable and retargetable binary translator. 1999. Cited on p. 4.

[6] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, The University of Auckland, 1997. No citations.

[7] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic birnary translation and optimization. 2001. Cited on p. 3.

[8] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. English shellcode. 2009. Cited on p. 3.

[9] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. 1993. Cited on p. 3.

[10] Yves Younan, Pieter Philippaerts, Frank Piessens, Wouter Joosen, Sven Lachmund, and Thomas Walter. Filter-resistant code injection on arm. 2009. Cited on p. 5.

Chapter 1

Literature Reviews

Dynamic Binary Translation and Optimization[7]

1.1

The aim of the paper is to describe a VLIW[1] architecture that is compatible with an existing architecture (PowerPC in this case). The new architecture, DAISY, can execute existing programs through the use of dynamic compilation.

Unlike the PIP aproach they design their architecture to be binary compatible (or at least with an architecture level VM) with PPC, rather than by relying on overlaps in the IS and platform specific behaviour.

Again, self modifying code is an issue; but less so on the PPC arch. They admit it'd be a pain on x86.

1. VLIW (Very Long Instruction Word) is a CPU architecture designed such that it can take advantage of instruction level parallelism.

Binary Translation[9]

1.2

Paper describes various methods of translating programs from one architecture to another without using virtualization. They discuss the various methods for translating between the VAX and OpenVMS architectures.

They point out that certain things can make programs untranslatable namely: exception handling differences, undocumented features, and system specific behavior (VAX memory management or program-image formats).

They do analysis on their programs. They note on the MIPS architecture:

only 10 instructions account for about 85% of all code.

This will probably be useful when trying to detect my PIPs from non-PIPs. They also add that certain sequences of instructions are idiomatic on a given architecture: again this can probably be exploited for distinguishing PIPs.

This seems to be more to do with the $\prod$-translation step of the original PIP paper, rather than the actual design of steganographic execution. However it might be interesting with the dual problem: given a program p for a machine M_1 such that $M_1(p) = b_1$ find M_2 and b_2 such that $M_2(p) = b_2$.

English Shellcode[8]

1.3

The paper describes a method of hiding shellcode such that it is hidden inside English text—making the detection of it harder. They take advantage of grammar and NL techniques to obscure the code.

Currently shellcode attacks can be spotted at compile time with linters (e.g. RATS. See also AEG paper), and through an emulation layer, taint analysis, or by checking inputs against known shellcodes.

User input or network traffic is considered suspicious when it is executable or anomalous

This is *insanely* cool. Hide a PIP gadget inside a buffer such that it looks like its holding some text. Looks completely (well relatively) inoccuous but then you're executing it.

1.4 Automatic Exploit Generation[1]

The AEG challenge: given a program find vulnerabilities and generate exploits automatically. Managed to find two new zero-days. Uses heuristics to rank potentially executable areas & to search them; to look first at HTTP get requests for example.

Requires the source code (its a linter). They have a nice video showing it working on a bug in `iwconfig` involving a `strcpy()` call.

An extension/reworking of the LLVM projects KLEE symbolic execution engine. Works better for this purpose but most exploits found usually benefit from a little hand tweaking first.

Can prioritise search areas based on different statistics (i.e. if a programmer is known to make off-by-one errors they can prioritize buffer error searching). Currently able to produce *return-to-stack* and *return-to-libc* attacks.

Return-to-stack attack

Change the return address of a function so that we return to somewhere on the stack where our shellcode has been placed.

Return-to-libc

Find a '`/bin/sh`' string in an executable (difficult!), then overwrite a return address to jump into an '`execve /bin/sh`' call.

Both these attacks still need to hijack control flow though. Neither attack describes how to do this, but once you have managed to get a shell it is effectively game over.

Very cool, but not what I'm really trying to do.

1.5 Design of a Resourcable and Retargetable Binary Translator[5]

Automatic translation of executables from one format to another without the source code (and by the *NoWeb* guy!). Essentially they do it by reverse engineering and decompiling before recompiling it.

Only works well on EXE formats at the moment. Uses an intermediate language, but it seems very early. Small programs with negligable overhead.

1.6 Finger Printing Malicious Code Through Statistical Opcode Analysis[3]

Looks at the number of different opcodes in various different programs. Finds that malware tends to have slightly different set of opcodes to regular programs, but it isn't clear how significant his results are: one or two percent is not a lot. With the more unusual opcodes the

results are a little more interesting. The frequency of rare opcodes is significantly higher in malware (it seems again no ranges to compare against). He concludes saying that opcodes are a relatively week predictor of malware in the most part but for rarer opcodes there a bit better.

Interesting. Suspect if you had a list of semantic nops available it'd work better though (and I bet malware would have a *lot* of them). Its an interesting theory anyway. I'd like to try it with some programs of my own.

CodeSurfer/x86---A Platform for Analyzing x86 Executables[2]

1.7

Describes an analysis package for x86 built upon IDA for doing *value-set analysis*. Value-set analysis is a method of recovering an intermediate representation of a program written in a high-level language. One of the big problems in doing this is knowing what is going on with memory and pointers. This paper attempts to describe a new tool for dealing with this for the purpose of analysizing the behaviour of malicious code.

Would being able to see additional NOPS help? Possibly not we're looking at memory use here, but its an attempt to reverse some of the obfuscating transforms.

Filter-resistant Code Injection on ARM[10]

1.8

About writing the first purely alphanumeric shellcode for the ARM platform.

Apparently they use the `PL` and `MI` conditionals throughout (presumably they intersect nicely with ASCII? 00 or 01 then?). No trivial `MOV` instruction: instead you have to do weird things using an immediate in a `SUB` expression and then load it using the PC with an offset. Very clever stuff. They can even enter and exit thumb mode.

Show that they can write a *BrainFuck* compiler (hence their alpha-numeric shellcode is Turing Complete).

An Information Theoretic Model for Steganography[4]

1.9

Gives information theoretic model for steganography.

Passive Adversary ε-Security

Given message m: you can decide correctly whether it was covertext C or stegotext S with probability less than or equal to [ˆperfect]:

$$D(P_C||P_S)\leq$$

[ˆperfect] if $= 0$ then the system is perfectly secure.

And so it goes on giving more mathematical definitions. A bit dry, but at least it gives the formal definitions.

A Taxonomy of Obfuscating Transformation[6]

1.10

Describes several techniques for obfuscating code in (e.g. to Java bytecode) in terms of three characteristics:

Potency how much does this confuse a human?

Resilience how easy is it de-obfuscate automatically?

Cost how much overhead does this add?

Obfuscating transform

A transform producing P' from P is an *obfuscating transform* if P and P' have the same observable behaviour and they terminate (or don't) in the same manner with the same output given the same input.

$$P \implies P'$$

Measures of Complexity

1. Program length
2. Cyclomatic complexity
3. Nesting complexity
4. Data flow complexity
5. Fan-in/out complexity
6. OO Metric (how much of it isn't well written object code?)

Transformation Potency

A transform 's potency is measuered as the ratio between the complexity of the transformed program and the original program, minus one. A transform is *potent* if this ratio is greater than zero.

$$_{\text{pot}}(P) \leftarrow \frac{E(P')}{E(P)} - 1$$

Measures of Resilience

Programmer effort $\sim$ how much time would it take to write a de-obfuscator to reduce the potency of τ.

Deobfusctor effort $\sim$ how efficient is the deobfuscator at deobfuscating τ.

Measures of Cost

Essentially how badly does τ effect the time and space complexity of the program.

Control Transforms

Opaque predicates $\sim$ add tests that you know will pass or fail a priori, but that will be difficult for a deobfuscator to deduce.

Computational Transforms

Dead Code $\sim$ add code that does nothing…or does something to irrelevant objects.

Extend Loops $\sim$ make the termination conditions really contrived.

Convert to a non-reducible flow-graph $\sim$ add features which don't exist in the language compiling from (eg. `GOTO`)

Inline function calls $\sim$ no library calls for you!

Embed an interpreter $\sim$ everything from a different language!

Add redundant operands $\sim$ multiply non-obviously by one! Add zero! Square then square root!

Parallelize it $\sim$ concurrency is hard

Aggregation

Inlining and outlining ∼ make random things one time function calls
Use coroutines ∼ nb: this won't foil an attack by Don Knuth
Clone methods ∼ duplicate functions and pick randomly between them
Make loops more complex ∼ loops within loops ∼ unroll loops and alter only some of the iterations
Reduce locality ∼ data loading and use are far apart

Data transformations

Use unusual structures ∼ bit packing is fun ∼ multiple bits for one bit of data ∼ encryption
Change encoding ∼ trinary is fun
Promote variables ∼ Use the most generic class of data
Split variables ∼ half the char is in this int and the other half is over here…
Generate constants ∼ no need to hardcode constants
Merge scalars ∼ these bits are for this… and the rest are for something else