¶ Steganographic Execution Pattens

Joseph Hallett

October 30, 2011

Contents

1

1.1 Title

Platform Independent Programs—Steganographic Execution

1.2 Advisor

Dr. Dan Page

1.3 Motivation

Until recently it was believed that having a program that ran on multiple architectures was limited to virtual machines, and a few hand crafted examples of bytecode that would execute on multiple platforms. In 2010, however, a team of researchers from Carnegie Mellon University succeeded in developing a method for creating multi-architecture programs automatically for the x86, ARM and MIPS platforms, as well as between the Darwin, FreeBSD and Linux operating systems.

At the moment there are no publicly available tools to create the platform independent programs, and there has been little work on detecting a PIP from a program for a single architecture. There haven't been any attempts to extend the Carnegie Mellon teams approach to other architectures (such as XMOS's XS1 or the JVM), and there aren't any databases detailing what the gadget headers (which are the key to developing platform independent execution) are between platforms.

There are many applications of platform independent execution. It can be used as a keyed steganographic system for resisting attempts to steal software. Alternatively it can be used to smuggle programs past a warden in the classic Prisoners' Problem. It can be used to create generic shellcode that will run on multiple architectures, and it can be used to create viruses that target specific computers on a network similar to the Stuxnet worm. They also offer an alternative to the Universal Binary or FatELF formats for executables, and can be integrated with the pre-existing COFF executable format with no changes.

1.4 Objectives

- Implement the existing PIP-shellcode generation algorithm using a better-than-brute-force method to find the bytecode patterns that define a gadget header.
- Attempt to implement the single instruction PIP generation algorithm, by extending the PIP-shellcode algorithm from the whole program level to smaller subsections of code.
- Extend the PIP generation algorithm to a new architecture, by finding the intersection of semantic-NOPs and jumps between different instruction sets and using HPC resources.

- Investigate methods to detect PIP from non-PIP by analyzing instruction density, searching for possible gadget headers under certain architectures, and using static analysis techniques to analyze code coverage within an executable.

1.5 Plan

- Study the architecture manuals for any relevant architectures. Find all semantic-NOPs and semantic-JUMPs in each architecture and build a templating system to generate gadget headers. Run templating system to generate a database of gadget headers. (A long time)
- Implement the RG-generation algorithm for PIPs at the whole program level, and then the single instruction level. (Hopefully less time)
- Use static analysis to find distinguishing characteristics of PIPs and develop an oracle for distinguishing PIP from non-PIP with a success rate better than guessing. (Depends… Probably a fair amount of time).

2

2.1 Dynamic Binary Translation and Optimization (Ebciogluo1)

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

The aim of the paper is to describe a VLIW 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.

2.2 Binary Translation (Sites93)

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 $M_2(p) = b_2$.

2.3 English Shellcode

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.

2.4 Automatic Exploit Generation

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.

2.4.1 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.

2.4.2 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.

2.5 Design of a Resourcable and Retargetable Binary Translator

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.

2.6 An Information-Theoretic Model for Steganography