PLATFORM INDEPENDANT PROGRAMS

∾

The creation and detection of steganographic execution patterns

Joseph Hallett

January 7, 2012

Submitted in partial fulfillment of the requirements
for the degree of Master of Engineering to the Department of Computer Science

Faculty of Engineering
University of Bristol

Abstract

My thesis! Ooo err!

Contents

Introduction

In 2010 a team of researchers developed a generalized method for creating platform independent programs (PIPs)[3]. A PIP is a special sort of program which can be run on multiple different computer architectures without modification[1] . Unlike shell scripts or programs written for a portable interpreter a PIP doesn't require another program to run or compile it; rather it runs as a native program on multiple architectures with potentially different behaviour on each.

To find a PIP you would have to analyze the architecture manuals for each architecture you wanted and find the instructions in each which compiled to identical patterns of bytecode and use them to construct your PIP. The approach taken by the authors was to find small PIPs with a very specific form: do nothing then jump[2] . By ensuring each architecture jumped to a different point and that each architecture didn't accidentally run into a region anouther architecture jumped into they could construct PIPs for any arbitrary program by splitting them up into blocks of instructions specific to each architecture and connecting them with the small PIPs.

They go on to give in the paper a generalized algorithm for constructing these PIPs, and say that they have a working implementation of it for creating PIPs for the x86, ARM, and MIPS platforms, as well as the Windows, Mac, and Linux operating systems.

Aim of the Project

For this thesis I have implemented a small section of the PIP generation algorithm—finding the *gadget headers*; the PIPs that link the specific code sections together. To generate the PIPs a list of *semantic NOPS*[3] and potential branch instructions has been found for each architecture in the original paper and to extend the work of the original paper I have analyzed two new platforms: the Java Virtual Machine, and XMOS's XS1.

Unfortunately there does not seem to be a public database of these instructions available for any architectures. Semantic NOPs have been used in areas other than creating PIPs, for example malware classification[2][6], but there still appears to be no exhaustive list exists documenting them. Part of the work required is to create one.

Why is this interesting?

PIPs can be used for a variety of applications. One potential application suggested by Brumley et. al.[3] is for sneaking programs past dynamic execution checkers. Suppose two prisoners *Alice & Bob*, wish to send a malicious program between themselves. To send the message they have to send it through a *Warden* who checks first that there communications don't contain anything malicious and only delivers the message if its believed to be harmless.

To sneak the program Alice and Bob use a form of *keyed-steganography*[4] . The program which they wish to communicate becomes their *steg-text*, and they construct a *cover-text* by writing some other program which doesn't contain anything malicious. They either prearrange a shared secret which is the platform that their programs should really be run on: this forms the key. With the cover-text and steg-text created they create their message by generating them into a PIP where on one platform (x86 say) it appears to run the innocuous program and on the secret key platform (ARM for the sake of argument) it runs the program they really wish to communicate. Alice hands the warden the program and tells him that it is for the innocuous architecture[5] . The warden runs the program and sees it isn't something he would want to restrict and delivers it. In fact unless he is aware that it has been in constructed in this way he may not even check any other architectures as for most platforms it will appear to be garbage just like any normal executable.

Another application is *exfiltration protection*[6] . Here the idea is that to protect its software from theft a secret agency could make a modification to an existing platform (the JVM or

another virtual machine would be a good choice here) and compile their program for this modified platform. They then create another program for the unmodified platform which does something else; maybe it phones home, maybe it destroys itself from the computer it's running on. They create a PIP out of these two programs and now if the program is stolen and the exfiltrator isn't aware of the PIP nature (or exactly what modifications were made to the architecture) they're not going to be able to run the program they removed.

Microcode offers another neat way to use PIPs. Suppose an attacker manages to compromise a system in such a way that they can alter the microcode of the processor, such as the recent HP printer attack[4]. Now suppose that as well as the microcode update they also modify an existing program[7] so that on the compromised system it gives a backdoor or acts maliciously, but on another (say one which is trying to forensically work out what is wrong with the printer) it acts normally. Brumley et. al. go on to point out[3] that if this was done by Intel and the PIP was a preexisting and digitally signed application then it is a particularly scary prospect. Merely signing the program would be insufficient protect a user it would not check if the machine it was executing on had been modified.

Other applications of PIPs include shellcode and viruses. For shellcode the idea is that you can write it once and use it anywhere. For viruses the idea is that if you could get the virus on a disk that is mounted on multiple architectures (say an NTFS share or USB key) then you can attack any platform you're plugged into.

The problem here is that although PIPs could be used to write architecture independant programs there are more elegant solutions available than relying on the intersection of instruction sets between architectures. There are a couple of preexisting systems for doing this such as Apple's *Universal Binary* or the *FatELF*[5] format.

Challenge

Summary

7.   In the PIP paper[3] they suggest ls.

Chapter 1

Technical Basis

Chapter 2

Components

Chapter 3

Execution

3.1 IMPLEMENTATION

3.2

Chapter 4

Conclusion

Bibliography

[1] AT&T. Elf header. In *SYSTEM V APPLICATION BINARY INTERFACE*. The Santa Cruz Operation, Inc., 1997. Cited on p. 1.

[2] Daniel Bilar. Fingerprinting malicious code through statistical opcode analysis. 2007. Cited on p. 1.

[3] Sang Kil Cha, Brian Pak, David Brumley, and Richard Jay Lipton. Platform-independent programs. 2010. Cited on pp. 1 and 2.

[4] Ang Cui and Jonathan Voris. Print me if you dare. In *28th Chaos Communication Congress Behind Enemy Lines*, 2011. Cited on p. 2.

[5] Icculus. Fatelf: Universal binaries for linux. Cited on p. 2.

[6] M.D. Preda, M. Christodorescu, S. Jha, and S Debray. A semantics-based approach to malware detection. 2007. Cited on p. 1.