Chapter 6
Dynamic Analysis

Abstract In the previous chapters we have examined static extraction of program features for the purpose of birthmark construction. Dynamic analysis is examined in this chapter. It is an alternative approach to static analysis that can be used for birthmark construction. Dynamic analysis concerns itself with analysing a running program. The program being run is typically isolated in an environment which allows its behaviour to be inspected. Typical behaviours that are extracted are the API call sequence. Instruction sequences, basic block sequences and control flow are amongst other behaviours that can also be identified.

Keywords Dynamic analysis · Hooking · Dynamic binary instrumentation · Virtualization · Application level emulation · Whole system emulation

6.1 Relationship to Static Analysis

There are roughly two approaches to extract program features from software. In the static approach, the software is never executed and the features are extracted from a static view of the program. In dynamic analysis the software is executed, possibly in a virtual machine, and its run-time behaviour examined. The run-time behaviours exhibit the properties or features being extracted.

Static analysis is effective because it is able to examine to represent the set of all possible execution paths by approximating program behaviour. This is important because behaviours of specific programs may be hard to trigger dynamically. It is often difficult to trigger corner cases in programs and as a result a number of dynamic analysis testing methodologies exist to address this such as the use of analysing code coverage during execution. In the case of malicious code, malware authors actively change the behaviour of the code when under analysis.

S. Cesare and Y. Xiang, *Software Similarity and Classification*,
SpringerBriefs in Computer Science, DOI: 10.1007/978-1-4471-2909-7_6,

The main advantage of dynamic analysis is that the semantics of the program are exhibited, and obfuscations applied to the program have less effect on these exhibited semantics. Attempting to identify run-time behaviour properties for multiple paths of execution has been researched [1]. It is still a new area, but using symbolic execution to trigger different behaviours has had some success. The results of exploring these multiple paths can be accumulated into a final report to infer the intent or potential behaviour of a piece of software.

6.2 Environments

Dynamic analysis requires an environment in which to run and isolate the program being analysed. The environment in which to run a program can be categorized in the following list:

- Hooking
- Dynamic Binary Instrumentation
- Virtualization
- Application Level Emulation
- Whole System Emulation

6.3 Debugging

An operating system typically provides an API to debug processes. Debugging can allow for operations including single stepping through execution an instruction at a time, or setting a breakpoint at a particular code address. Debugging can be useful to monitor non malicious programs, however, most malware today implements anti-debugging functionality which can detect the presence of a debugger.

6.4 Hooking

Hooking is the process of intercepting API calls allowing for possible instrumentation. Hooks can be placed in user space or kernel space. Hooking is commonly used by commercial Antivirus software to monitor process behaviour and detect possible misuse. Detours [2] is an implementation of hooking for the Windows operating system. The basic mode of operation is to overwrite the function in memory with a trampoline to the intercept handling code. The intercept handling code performs any instrumentation or monitoring as necessary then restores control back to the original function. Another method of hooking is overwriting dispatch tables such as system call tables or import addresses. It is also

possible in Linux to natively intercept API calls to dynamic libraries by preloading another library. Malware today often can detect the presence of hooking by implementing checksums over their executable code.

6.5 Dynamic Binary Instrumentation

Dynamic binary instrumentation is an approach that instruments native code on the fly. The binary being executed is controlled from a dispatcher which analyses the code, instruments it, and then rewrites it for execution. Some examples of dynamic binary instrumentation include PIN [3], DynamoRIO [4], and Valgrind [5]. Dynamic binary instrumentation based on PIN has been used for malware unpacking and analysis in [6, 7].

6.6 Virtualization

Virtualization is a technique that supports native execution of a guest operating system by exploiting separation and isolation mechanisms implemented by the native hardware architecture or software. A number of methods are available to implement virtualization including paravirtualization which must be supported by both the host and the guest operating systems. The most important type of virtualization for providing an environment to perform feature extractions is hardware assisted virtualization. In the x86 architecture, hardware assisted virtualization was not always supported and detection of the virtualized environment was implemented by many strains of malware [6]. Hardware assisted virtualization has been used for malware analysis [8]. This type of analysis is harder to detect but attacks still exist to detect virtualization from a guest [9]. For example, it is known that memory caching between guests and hosts are different in the virtualized environment. However, as virtualization becomes a standard tool on the desktop, malware authors might no longer be able to associate virtualization with threat analysis.

6.7 Application Level Emulation

Application level emulation emulates the operating system and instruction set architecture for specific applications. This approach has been predominantly employed in Antivirus systems to perform real-time analysis of malware and automated unpacking [10]. Its main disadvantage is its inability to faithfully emulate the desired system which makes it susceptible to detection as has been the case with modern malware.

The typical features emulated in an application level emulator on the x86 Windows platform for the purposes of malware detection include:

- Instruction Set Architecture (ISA).
- Virtual Memory.
- Windows API emulation.
- Linking and Loading.
- Thread and Process Management.
- OS Specific Structures.

The instruction set architecture (ISA) must be faithfully emulated. In practice, most deployed emulators only simulate part of the complete x86 ISA. Malware authors have responded by using uncommon instructions such as those associated with MMX and FPU to detect and thwart the emulation process.

Virtual memory must be emulated. 32-bit x86 employs a segmented memory architecture. In Windows the segment registers are utilised to reference thread specific data. This data is additionally used by Windows Structured Exception Handling (SEH). SEH is used to gracefully handle abnormal conditions such as division by zero and is routinely used by packers and malware to obfuscate control flow.

The Windows API is the official system call interface provided by Windows. There are too many Windows API functions to full emulate in a typical environment so only the most common APIs are implemented. This also presents a method for malware to detect and thwart an emulator using uncommon API calls.

Linking and loading must be implemented by an emulator. Program loading entails allocating the appropriate virtual memory, loading the program text, data and dynamic libraries. Relocations must be performed and run-time linking performed.

Threads and process management must be performed. Malware can sometimes try to detect and thwart a debugger or emulator by being multi-process or multi-threaded.

OS specific structures must also be simulated. Windows has a number of these including the Process Environment Block, the Thread Environment Block and the Loader Module. These structures are visible to applications and can be used by malware.

6.8 Whole System Emulation

A whole system emulator emulates the hardware of a PC. This allows an operating system to be installed as a guest. There are roughly two approaches to implement a whole system emulator or any emulator in general:

- Interpretation
- Dynamic Binary Instrumentation

An example of whole system emulators includes QEMU [11] which is based on dynamic binary translation. Bochs is another whole system emulator that uses

interpretation instead of dynamic binary translation. Bochs has been used for malware unpacking and analysis [12]. Interpretation is slower than dynamic binary translation which makes QEMU a popular choice.

Interpretation works by implementing a fetch, decode and execute loop inside the emulator. Dynamic binary translation translates sequences of code from the guest into native code on the host. It can perform optimisations on these blocks of code which improves efficiency. The blocks are also cached reducing the costs of translation. In general, dynamic binary translation offers significant performance improvements over an interpretation based emulator.

It is possible to modify a whole system emulator to monitor or instrument guest execution [13]. The BitBlaze project [14] is a project for binary analysis that makes heavy use of whole system emulation to perform tasks including malware analysis. Whole system emulation is effective for behavioural analysis of code but attacks exist to detect its presence from the guest [9].

References

1. Brumley D, Hartwig C, Kang MG, Liang Z, Newsome J, Song D, Yin H (2007) BitScope: automatically dissecting malicious binaries. Technical report CMU-CS-07-133, School of Computer Science, Carnegie Mellon University
2. Hunt G, Brubacher D (1999) Detours: binary interception of win32 functions. Paper presented at the proceedings of the 3rd conference on USENIX Windows NT symposium, vol 3. Seattle, Washington
3. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. Paper presented at the proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation
4. Bala V, Duesterwald E, Banerjia S (2000) Dynamo: a transparent dynamic optimization system. Paper presented at the proceedings of the ACM SIGPLAN 2000 conference on programming language design and implementation
5. Nethercote N, Seward J (2003) Valgrind a program supervision framework. Electron Notes Theor Comput Sci 89(2):44–66
6. Guizani W, Marion JY, Reynaud-Plantey D (2009) Server-side dynamic code analysis. In: Malicious and unwanted software (MALWARE), 2009 4th international conference on, 2009, pp 55–62
7. Quist D (2007) Valsmith covert debugging circumventing software armoring techniques. In: Black hat briefings USA
8. Dinaburg A, Royal P, Sharif M, Lee W Ether (2008) Malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on computer and communications security 2008. ACM, New York, USA, pp 51–62
9. Raffetseder T, Kruegel C, Kirda E (2007) Detecting system emulators. In: Lecture notes in computer science, vol 4779, p 1
10. Cesare S, Xiang Y (2010) Classification of malware using structured control flow. In: 8th Australasian symposium on parallel and distributed computing (AusPDC 2010
11. Bellard F (2005) QEMU, a fast and portable dynamic translator. In: USENIX annual technical conference 2005, pp 41–46
12. Boehne L (2008) Pandora's bochs: automatic unpacking of malware. University of Mannheim

13. Bayer U, Kruegel C, Kirda E (2006) TTAnalyze: a tool for analyzing malware. In: European Institute for Computer Antivirus Research (EICAR), 2006
14. Song D, Brumley D, Yin H, Caballero J, Jager I, Kang M, Liang Z, Newsome J, Poosankam P, Saxena P (2008) BitBlaze: a new approach to computer security via binary analysis. In: Information systems security