Multiprocess Malware

Marco Ramilli
DEIS, Univ. of Bologna
Via Venezia, 52 - 47023
Cesena, ITALY
marco.ramilli@unibo.it

Matt Bishop
Department of Computer Science
Univ. of California, Davis
Davis, CA 95616-8562, USA
bishop@cs.ucdavis.edu

Shining Sun
Computer Science
Univ. of Hong Kong
Hong Kong
snsun@cs.hku.hk

Abstract

Malware behavior detectors observe the behavior of suspected malware by emulating its execution or executing it in a sandbox or other restrictive, instrumented environment. This assumes that the process, or process family, being monitored will exhibit the targeted behavior if it contains malware. We describe a technique for evading such detection by distributing the malware over multiple processes. We then present a method for countering this technique, and present results of tests that validate our claims.

1 Introduction

Today selling malware defenses (called "anti-virus" even though they deal with malware in general) is a multi-billion dollar business. As the anti-virus researchers and industry develop new defenses, new malware evades them. Defense mechanisms have grown in complexity from simple signature scanning to a combination of signature scanning, behavior analysis, and emulation. The suspect data is either scanned looking for suspicious patterns of bits (static *signature scanning*) or is run in a restricted environment and its behavior analyzed for suspicious patterns of actions (dynamic *behavior analysis*). This type of analysis makes two critical assumptions. First, all signature-based anti-virus mechanisms assume that malware exhibits the signature in a form that the anti-virus mechanism can detect.

Second, anti-virus mechanisms assume that signatures are valid representations of malware. This is necessary to reduce the number of false positives.

Consider what happens when the first assumption is incorrect. In a system that uses static signatures, break the data object into parts such that the signature is also broken into components, each part of the data object having a different component of the signature. The individual parts can be sent onto the target system, reassembled, and executed. As signature scanners scan data objects entering the system or being loaded into memory, and the complete malware only exists as a result of the components being assembled in memory, the anti-virus engine will never scan an entity containing the complete signature. This approach evades static signature detection [13]. The program to assemble the components looks like a standard Windows .NET engine, and so flagging it as malware would cause many standard Windows programs to be flagged as malware also.

Now look at the first assumption in light of behavioral analysis, which typically involves examining patterns of application programmer interface (API) calls. A signature exploits two relationships among these calls. The *temporal* relationship reflects the temporal ordering of the events The *spatial* relationship is that the signature occurs in the "view" of the anti-virus mechanism. In practice, this means they occur in a single process, or a related family of processes, usually started from a downloaded data object. Breaking either of these relationships inhibits detection. For example, if the API calls occur in a different order, the anti-virus mechanism will not recognize the signature. In this case, the different temporal ordering will probably render the "malware" harmless.

But the spatial relationship is a different matter. An attacker divides the malware into multiple coordinated processes such that no sequence of API calls executed by one process matches any of the behavioral signatures. The entities that will be executed to create the processes do not match any of the static signatures, and none of the resulting processes performs any actions that the anti-virus tool will flag as suspicious. This negates the spatial assumption because no signature occurs in any one "view," or process. Then the anti-virus tools would not detect that malware has been injected onto the system.

Such an attack requires two steps. The first step is to place the malware components onto the system in such a way that each component can be executed to create a process that co-ordinates with one or more of the other component processes. The second step is to run each component individually. They must co-ordinate in such a way that their combined actions are equivalent to the single malware. This requires that we partition our malware into components none of which contains a static signature that causes an anti-virus tool to raise an alert. The efficacy of this approach was discussed elsewhere [13]; suffice it to say that the components need only avoid detection by the active anti-virus tools on the system (which may use static signatures, behavioral analysis, or a combination of the two). The components are then placed onto the system individually, and in such a way that they will be executed either immediately or in the future. Because many anti-virus tools look for patterns of behavior among processes and their descendants, we must ensure each of the components starts as a sibling process of the other components rather than as a descendant. Note that, in many cases, they may be executed sequentially, and over a period of time—that is, temporally far apart.

This paper reports on our design and experiments with one piece of malware rewritten as separate components. First, we examine related work. Then, we discuss our design of this multi-process malware. We experiment with one particular instance to demonstrate the results of this design. We conclude with a discussion of future directions and consider ways to ameliorate this threat.

2 Related Work

Tasks are often distributed across multiple processes that co-operate to achieve a common goal. In computer security, many previous attacks have done this. For example, the Internet worm [6] placed an executable "grappling hook" on the target system. The hook pulled over the rest of the worm from the other system. This is similar in concept to the way the computer viruses Dichotomy [7] and RMNS [8] worked. They had two separate parts, one of which intercepted the relevant calls, and invoked the second part that performed the malicious action. However, the Internet worm was two different processes, and the second (the worm proper) was constructed by linking an object file with libraries already resident on the target.

Other attacks, often in the guise of malware [1,2,4,9] are "multi-stage" in their activation or execution. Models [5, 12, 15] and interpretative methods such as visualization [10] help analysts understand how multi-stage attacks work and how they spread. Many existing worms work this way, exchanging messages and copies of the worm with other hosts to propagate and to control their spread.

Unlike these attack methods, we construct multiple co-ordinated processes that perform the same actions as malware. Thus, there is no single process that performs the malicious actions, so any attempt to monitor individual processes for malicious behavior will fail. Each process is independent of the others, save for the need to co-ordinate their actions (which may involve the use of, for example, covert channels rather than conventional interprocess communication calls). Further, the processes begin independently; they need not pull over programs from other systems. Indeed, some of our processes may be implemented using gadgets [11, 14] to reduce the amount of data that must be placed on the system.

3 Design of Multi-Process Malware

We define *spatial locality* to be the view of the system over which anti-virus software looks for known or anomalous patterns of behavior in the suspected programs. For example, most existing anti-virus systems look at events either in the same process or in descendants of processes executed by the same user. So, let events a and b occur in processes p_a and p_b, respectively. We define $user(P)$ as the user/owner of process P. Then a and b are in the same spatial locality if $user(p_a) == user(p_b)$ and any of $p_a == p_b$, p_a is an ancestor of p_b, or p_b is an ancestor of p_a, is true.

To inhibit detection, we simply disrupt the spatial locality of events, so the events matching the patterns are spread over multiple localities. Consider a piece of malware M implementing a sequence of events $R = [a_1, \ldots, a_n]$, where $[\ldots]$ indicates a partial ordering of $a_1, \ldots, a_n$, and producing a resulting action R. In the usual case, the spatial relationship of all n events is within the same process. They are in the same spatial locality as defined by most current anti-virus software. So, we distribute the events across multiple processes in such a way that the events are in different spatial localities. Given our definition, the simplest way to do this is to put each events a_i into a separate process that is unrelated by ancestry to the processes $a_j, j \neq i$, performing the other actions. Then, the events are in the different

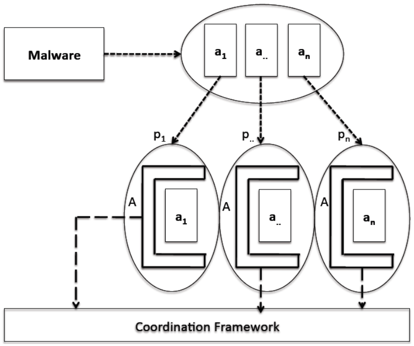


Figure 1. Multi-processes Malware.

spatial localities of most current anti-virus software, and so they will not detect that the events are co-ordinated.

In other words, we spread the events that the malware exhibits over several processes, turning the single malware process into a set of processes that, individually, do not exhibit the events of the malware but, taken as a whole, do. As a spatial locality defines a particular memory context, the multi-process malware must be able to run different actions from different memory contexts. Let $f(M) = R$ be a function that produces the result R given executing process M (the process containing the malware). We then define processes $p_1, \dots, p_n$ such that p_i executes a_i. Then, as expected:

$$\begin{aligned} R = f(M) &= f([p_1, \dots, p_n]) \\ &= [f(p_1), \dots, f(p_n)] = [a_1, \dots, a_n] \end{aligned}$$

M consists of a partial ordering of processes producing in turn a partial ordering of events. The processes must co-ordinate with each other to ensure the required temporal relationships of the events are preserved. This can be done through IPC among the processes themselves, or by communicating with a co-ordinating process A that performs none of the events a_i itself. Figure 1 depicts this process pictorially.

As an example, consider the original (single process) version of the malware Zeus (M = Zeus), which infects consumer PCs, waits for a user to log into a financial institution on the list that Zeus targets, and then steals the user's credentials and sends them to a remote server. Zeus also injects HTML into the browsed page so that its own content is displayed with (or instead of) the genuine pages from the bank's web server. In this way, it is able to ask the user to divulge more information, such as a payment card number and PIN, one-time password, and Transaction Authentication Numbers used by some banking services to authorize financial transactions. For Zeus, we define R as "providing a user's credentials to another user without authorization". The original Zeus malware accomplishes this in three steps: (a) It injects itself onto the system (a_1); (b) It steals the credentials (a_2); and (c) It sends them to remote storage (a_3). Applying the above analysis to Zeus, we obtain three single and coordinated processes p_1, p_2, and p_3, each performing the actions a_1, a_2, and a_3 respectively, in that order. This gives us the same result as running M directly.

In our experiments, detailed below, we used a fourth process A to co-ordinate the other three. Had we chosen, we could have embedded a co-ordination wrapper into each of the processes p_i, as shown in Figure 1. The lack of ancestor/descendent relationships among the processes is critical because most anti-virus mechanisms define spatial locality as that relationship. They look for for sequences of events in the same process, or they look at events in process families. This means that if p_1 is the parent of p_2 and p_3, the anti-virus mechanism would detect the sequential execution of a_1, a_2, and a_3. But if p_1, p_2, and p_3 are unrelated (for example, each was executed by a different user) or are siblings, under the usual definition of "spatial locality", the anti-virus mechanism would not correlate the events as indicating malware. Thus, the original malware M performs the same actions in the same sequence as do the distinct processes $[p_1, \dots, p_k]$ when executed in a way that satisfies the constraints of the partial ordering. This does not satisfy the spatial locality assumption, and hence will evade detection. As the p_i processes do not themselves contain the malware signature, they will not be detected individually either.

To summarize, the preconditions for this attack to work are that neither dynamic nor static anti-virus mechanisms must flag as suspicious the processes $p_1, \dots, p_n$; processes $p_1, \dots, p_n$ must be coordinated to guarantee an execution order of $a_1, \dots, a_n$ that satisfies the partial order constraints; and each process p_i must be executed.

4 Experiments

In order to show that fragmenting malware as described above evades anti-virus detectors, we need to show that partitioning the malware into separate processes allows us to put the parts onto the system without the static or behavioral detectors flagging any part as suspicious. Then, we need to show that the separate processes can perform the malicious action without a behavioral anti-virus detector detecting the attack.

For the first step, we follow Ramilli's and Bishop's approach [13]. Define $AV(x)$ to be an anti-malware detection mechanism that returns true if the input to AV, namely x, is malware and false if not. Our anti-virus function AV for the first stage is the set of anti-virus detectors at Virus Total, which includes most commercial anti-virus programs as well as open-source ones. In the second stage, we use behavior-based anti-virus programs to which we had access as our AV. For ease of construction, we selected malware for which source code is available. By monitoring the actions of the malware, one can partition the malware into a sequence of actions, and from those derive the component processes.

4.1 First Experimental Stage: Static Analysis

We assume that Virus Total uses well-configured and up-to-date anti-virus engines. We also assume the anti-virus tools there perform a static signature analysis on the given files. The target malware is BullMoose [3]. From the source code, we see that BullMoose takes 3 actions to compromise the system:

a_1 It saves an exploited HTML page onto the local hard drive.

a_2 It changes the Microsoft Windows registry key to make Internet Explorer the default browser program.

a_3 It then causes IExplorer.exe (the executable for Internet Explorer) to be opened with the default page being the exploited HTML page from point 4.1, above.

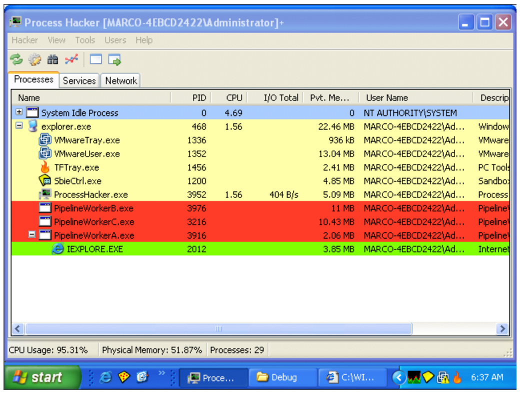
Table 1 shows that all but 4 anti-virus tools found the BullMoose virus. We next apply the transformation process shown in Figure 1 by building three different executables p_1, p_2, and p_3, each one wrapping the respective action a_1, a_2, a_3, respectively. The three processes might be run in different orders and at different times, because the coordination framework ensures the timing and sequence of actions matches those of the original BullMoose malware. When all three processes have completed, they have performed the same actions as BullMoose. Table 2 shows that none of the static anti-virus tools flagged the three executables as suspicious. The static analysis engines cannot detect the malware because the malware's signature, which for BullMoose is the code that performs the sequence of actions (a_1, a_2, a_3), has been broken into different files so that each file contains $\frac{1}{3}$ of the original signature. Detecting this attack using static analysis would require the detectors to flag any executable containing *any* of a_1, a_2, or a_3 as suspicious. This would cause a large number of false positives.

Table 1. Static Analysis Results: BullMoose.

Antivirus	Version	Last Update	Result
AhnLab-V3	2011.01.08.00	2011.01.07	Win-Trojan/[..]
AntiVir	7.11.1.57	2011.01.07	TR/Malex.6656f
Antiy-AVL	2.0.3.7	2011.01.07	Trojan/Win32.[..]
Avast	4.8.1351.0	2011.01.07	Win32:M[..]
Avast5	5.0.677.0	2011.01.07	Win32:M[..]
AVG	9.0.0.851	2011.01.07	Generic15.BZXE
BitDefender	7.2	2011.01.07	Trojan.G[..]
CAT-QuickHeal	11.00	2011.01.07	Trojan.S[..]
ClamAV	0.96.4.0	2011.01.07	Trojan.A[..]
Command	5.2.11.5	2011.01.07	W32/M[..]
Comodo	7331	2011.01.07	UnclassifiedM[..]
DrWeb	5.0.2.03300	2011.01.07	Trojan.S[..]
Emsisoft	5.1.0.1	2011.01.07	Trojan.W[..]
eSafe	7.0.17.0	2011.01.06	Win32.Agent
eTrust-Vet	36.1.8087	2011.01.07	-
F-Prot	4.6.2.117	2011.01.07	W32/M[..]
F-Secure	9.0.16160.0	2011.01.07	Trojan.G[..]
Fortinet	4.2.254.0	2011.01.07	-
GData	21	2011.01.07	Trojan.G[..]
Ikarus	T3.1.1.90.0	2011.01.07	Trojan.Win32.S[..]
Jiangmin	13.0.900	2011.01.07	Trojan/Small.hts
K7AntiVirus	9.75.3472	2011.01.07	Trojan
Kaspersky	7.0.0.125	2011.01.07	Trojan.W[..]
McAfee	5.400.0.1158	2011.01.07	Generic.dx!hyb
McAfee-GW-Edition	2010.1C	2011.01.07	Generic.dx!hyb
Microsoft	1.6402	2011.01.07	Trojan:Win32/[..]
NOD32	5768	2011.01.07	Win32/Agent.RCX
Norman	6.06.12	2011.01.07	W32/M[..]
nProtect	2011-01-07.01	2011.01.07	Trojan/W[..]
Panda	10.0.2.7	2011.01.07	Trj/CI.A
PCTools	7.0.3.5	2011.01.07	Trojan.Generic
Prevx	3.0	2011.01.08	-
Rising	22.81.04.04	2011.01.07	Trojan.W[..]
Sophos	4.61.0	2011.01.07	Mal/Generic-L
SUPERAnti Spyware	4.40.0.1006	2011.01.07	-
Symatec	20101.3.0.103	2011.01.07	Trojan Horse
TheHacker	6.7.0.1.112	2011.01.07	Trojan/S[..]
TrendMicro	9.120.0.1004	2011.01.07	TROJ-G[..]
TrendMicro-HouseCall	9.120.0.1004	2011.01.07	TOJ-G[..]
VBA32	3.12.14.2	2011.01.06	Trojan.W[..]
VIPRE	7991	2011.01.07	Behaves[..]
ViRobot	2011.1.7.4242	2011.01.07	-
VirusBuster	13.6.134.0	2011.01.07	Trojan.S[..]

Table 2. Static Analysis Results: Multi-Process BullMoose.

Antivirus	Version	Last Update	Result
AhnLab-V3	2011.01.08.00	2011.01.07	-
AntiVir	7.11.1.57	2011.01.07	-
Antiy-AVL	2.0.3.7	2011.01.07	-
Avast	4.8.1351.0	2011.01.07	-
Avast5	5.0.677.0	2011.01.07	-
AVG	9.0.0.851	2011.01.07	-
BitDefender	7.2	2011.01.07	-
CAT-QuickHeal	11.00	2011.01.07	-
ClamAV	0.96.4.0	2011.01.07	-
Command	5.2.11.5	2011.01.07	-
Comodo	7331	2011.01.07	-
DrWeb	5.0.2.03300	2011.01.07	-
Emsisoft	5.1.0.1	2011.01.07	-
eSafe	7.0.17.0	2011.01.06	-
eTrust-Vet	36.1.8087	2011.01.07	-
F-Prot	4.6.2.117	2011.01.07	-
F-Secure	9.0.16160.0	2011.01.07	-
Fortinet	4.2.254.0	2011.01.07	-
GData	21	2011.01.07	-
Ikarus	T3.1.1.90.0	2011.01.07	-
Jiangmin	13.0.900	2011.01.07	-
K7AntiVirus	9.75.3472	2011.01.07	-
Kaspersky	7.0.0.125	2011.01.07	-
McAfee	5.400.0.1158	2011.01.07	-
McAfee-GW-Edition	2010.1C	2011.01.07	-
Microsoft	1.6402	2011.01.07	-
NOD32	5768	2011.01.07	-
Norman	6.06.12	2011.01.07	-
nProtect	2011-01-07.01	2011.01.07	-
Panda	10.0.2.7	2011.01.07	-
PCTools	7.0.3.5	2011.01.07	-
Prevx	3.0	2011.01.08	-
Rising	22.81.04.04	2011.01.07	-
Sophos	4.61.0	2011.01.07	-
SUPERAnti Spyware	4.40.0.1006	2011.01.07	-
Symatec	20101.3.0.103	2011.01.07	-
TheHacker	6.7.0.1.112	2011.01.07	-
TrendMicro	9.120.0.1004	2011.01.07	-
TrendMicro-HouseCall	9.120.0.1004	2011.01.07	-
VBA32	3.12.14.2	2011.01.06	-
VIPRE	7991	2011.01.07	-
ViRobot	2011.1.7.4242	2011.01.07	-
VirusBuster	13.6.134.0	2011.01.07	-



Name	PID	CPU	I/O Total	Pvt. Me...	User Name	Descrip
System Idle Process	0	4.69		0	NT AUTHORITY\SYSTEM	
explorer.exe	468	1.56		22.46 MB	MARCO-4EBCD2422\Ad...	Window
VMwareTray.exe	1336			936 kB	MARCO-4EBCD2422\Ad...	VMware
VMwareUser.exe	1352			13.04 MB	MARCO-4EBCD2422\Ad...	VMware
TFTray.exe	1456			2.41 MB	MARCO-4EBCD2422\Ad...	PC Tool
SbieCtrl.exe	1200			4.85 MB	MARCO-4EBCD2422\Ad...	Sandbo
ProcessHacker.exe	3952	1.56	404 B/s	5.09 MB	MARCO-4EBCD2422\Ad...	Process
PipelineWorkerB.exe	3976			11 MB	MARCO-4EBCD2422\Ad...	Pipeline
PipelineWorkerC.exe	3216			10.43 MB	MARCO-4EBCD2422\Ad...	Pipeline
PipelineWorkerA.exe	3916			2.06 MB	MARCO-4EBCD2422\Ad...	Pipeline
IEXPLORE.EXE	2012			3.85 MB	MARCO-4EBCD2422\Ad...	Internet

CPU Usage: 95.31% | Physical Memory: 51.87% | Processes: 29

Figure 2. Dynamic Analysis Results

4.2 Second Experimental Stage: Dynamic Analysis

We next considered a set of anti-virus tools that performed behavioral (dynamic) analysis. We put our anti-virus tools[1] on a well-configured and up-to-date version of Microsoft Windows XP and, for each one, we performed the following tests:

1. We ran the original BullMoose malware to test if the anti-virus tool gave an alert.
2. We then ran the multi-process version of the malware to test if the anti-virus tool gave an alert.
3. When no alert occurred, we checked the real execution of the malware by running Internet Explorer to see if it opened the crafted HTML page, which contained a malicious Javascript program.

Figure 2 shows the results of one test using ThreatFire. In a previous test, ThreatFire detected the original malware, blocked it from executing, and moved it into the designed quarantine folder. The figure presents the output, viewed using Process Hacker, showing that the three processes (highlighted in red) ran without triggering ThreatFire. The process below, in green, is Internet Explorer running as a child of the third process, with the crafted HTML page as the default page.

This demonstrates that preconditions 1 and 3, at the end of Section 3, hold. The co-ordination framework that ensures precondition 2 holds is straightforward to write, and for space reasons we omit the details.

[1] The specific anti-virus tools we use were Anubis, JoeBox, Norman, Sophos AV, Avira AnntiVir, ThreatFire, and AVG.

5 Conclusion and Future Work

An open question is how to automate the division of an arbitrary malware so it achieves the same results as the original, and yet none of the component processes will be flagged as suspicious. Determining the conditions under which it can be done raises questions about program analysis.

An obvious counter to the attack methodology described here is to broaden the notion of "spatial locality" to include all events on the system. Then the anti-virus system can perform the correlation over multiple *unrelated* processes, closing the gap exploited here. This is similar to how many host-based intrusion detection systems work, but they are too heavy-weight for many environments such as home computers because, if the data is analyzed on the resident system, in addition to security considerations, the performance degradation due to the continuous checking may affect the usability of these systems.

This suggests an alternative design for anti-virus systems. In addition to looking for signatures (whether static or behavioral), focus upon the *effects* of the attack. The multi-process version of BullMoose would be (and, in fact, was) detected by an anti-virus system that looked for alterations to the Registry that affected Internet Explorer's start-up page. The essence of this approach is to look for the effects of compromise and not for things that may cause compromise. It is the distinction between misuse-based intrusion detection (which looks for signatures of known attacks) and specification-based intrusion detection (which detects programs acting incorrectly). The program may act incorrectly for a variety of reasons; what matters is it acts incorrectly. Similarly, here, what matters is that the start-up page changed. Why it changes is for forensic analysis. That it *did* change (or something tried to change it) is what causes the breach.

Acknowledgements: Thanks to Richard Yeh and Harvey Chan for hepful discussions.

References

[1] M. Abu Rajab, F. Monrose, and A. Terzis. On the impact of dynamic addressing on malware propagation. In *Proceedings of the 4th ACM workshop on Recurring malcode*, pages 51–56, New York, NY, USA, 2006. ACM.

[2] D. Bilar. Noisy defenses: Subverting malware's OODA loop [extended abstract]. In *Proceedings of the 4th Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '08, pages 9:1–9:3, New York, NY, USA, May 2008. ACM.

[3] BullMoose. Source code of computer viruses: Bullmoose, Nov. 2009.

[4] W. Cui, V. Paxson, and N. C. Weaver. GQ: Realizing a system to catch worms in a quarter million places. Technical Report TR-06-004, International Computer Science Institute, Berkeley, CA, USA, Sep. 2006.

[5] K. Daley, R. Larson, and J. Dawkins. A structural framework for modeling multi-stage network attacks. In *Proceedings of the 2002 International Conference on Parallel Processing Workshops*, pages 5–10, 2002.

[6] M. Eichin and J. Rochlis. With microscope and tweezers: An analysis of the internet virus of 1988. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 326–343, May 1989.

[7] E. Kaspersky. Dichotomy: Double trouble. *Virus Bulletin*, pages 8–9, May 1994.

[8] E. Kaspersky. RMNS—the perfect couple. *Virus Bulletin*, pages 8–9, May 1995.

[9] R. W. Lo, K. N. Levitt, and R. A. Olsson. MCF: a malicious code filter. *Computers & Security*, 14(6):541–566, Nov. 1995.

[10] S. Mathew, R. Giomundo, S. Upadhyaya, M. Sudit, and A. Stotz. Understanding multistage attacks by attack-track based visualization of heterogeneous event streams. In *Proceedings of the 3rd international workshop on Visualization for computer security*, pages 1–6, New York, NY, USA, 2006. ACM.

[11] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58, New York, NY, USA, Dec. 2010. ACM.

[12] D. Ourston, S. Matzner, W. Stump, and B. Hopkins. Applications of hidden markov models to detecting multistage network attacks. In *Proceedings of the 36th Hawaii International Conference on Systems Sciences*, Los Alamitos, CA, USA, 2003 2003. IEEE Comput. Soc. 36th Hawaii International Conference on Systems Sciences, 6-9 January 2003, Big Island, HI, USA.

[13] M. Ramilli and M. Bishop. Multi-stage delivery of malware. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software*, MALWARE 2010, pages 91–97, Piscataway, NJ, USA, Oct. 2010. IEEE.

[14] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Securityecurity*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[15] S. J. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 workshop on New security paradigms*, pages 31–38, New York, NY, USA, 2000. ACM.