Lurking in the Shadows: Identifying Systemic Threats to Kernel Data (Short Paper)

Arati Baliga, Pandurang Kamat and Liviu Iftode
Department of Computer Science, Rutgers University
Email: {aratib, pkamat, iftode}@cs.rutgers.edu

Abstract

The integrity of kernel code and data is fundamental to the integrity of the computer system. Tampering with the kernel data is an attractive venue for rootkit writers since malicious modifications in the kernel are harder to identify compared to their user-level counterparts. So far however, the pattern followed for tampering is limited to hiding malicious objects in user-space. This involves manipulating a subset of kernel data structures that are related to intercepting user requests or affecting the user's view of the system. Hence, defense techniques are built around detecting such hiding behavior. The contribution of this paper is to demonstrate a new class of stealthy attacks that only exist in kernel space and do not employ any hiding techniques traditionally used by rootkits. These attacks are stealthy because the damage done to the system is not apparent to the user or intrusion detection systems installed on the system and are symbolic of a more systemic problem present throughout the kernel. Our goal in building these attack prototypes was to show that such attacks are not only realistic, but worse; they cannot be detected by the current generation of kernel integrity monitors, without prior knowledge of the attack signature.

1. Introduction

Integrity of the operating system kernel is critical to the security and integrity of a computer system. Tampering with the kernel is traditionally performed by malware called rootkits. The attacker uses a rootkit to hide his presence on the compromised system. Other forms of malware such as worms, viruses and spyware successfully evade detection from the anti-virus software running on the system when bundled with a rootkit. Sophisticated rootkits achieve this hiding behavior by tampering with the kernel data. This may involve process hiding or hiding objects by tampering with the jump tables or file system handlers.

The challenge for the user and the intrusion detection system (IDS) is to detect the fact that the system is compromised as soon as the attack happens. The rootkit on the other hand, tries to hide this fact from the user for as long as possible. At this point, the attacker already has obtained root control on the system. While he has the ability to perform visibly damaging actions such as erase all files on the file system or reboot the system to install a new kernel image, these actions conflict with his goals of hiding his presence to retain long term control. Radical actions taken by the attacker are quickly detected with very high probability and the attacker loses control of the system as a result.

Monitoring the integrity of the kernel is achieved by isolating the detector from the system under surveillance. The detector either resides on a secure co-processor [18, 7] or uses virtual machine introspection [5]. While the mechanisms for kernel integrity monitoring have been successfully designed, the policies pertaining to the data that is to be monitored needs to be inferred manually from known rootkit behaviors. This is an uphill battle, similar to the one faced by anti-virus software, where traditionally attackers hold an edge over defenders.

Petroni et al [8] developed such a specification architecture to manually specify constraints on kernel data and monitor for violations. While this architecture allows the specification of known attack profiles, newer and unknown attacks, which target data structures that are not monitored or attack monitored data structures in a different manner, escape detection. Besides being an external asynchronous monitor, it often finds data structures in an inconsistent state, which limits it's scalability and ultimately, the feasibility of such a solution. A methodology to comprehensively protect the integrity of all data structures in the kernel is still a topic of active research.

The only form of kernel data tampering performed by rootkits pertains to hiding the malicious objects. Such object hiding deceives the user into believing that he has a clean system. Very early rootkits achieved this goal by installing TROJANED system binaries and shared libraries that provided doctored responses. As file integrity tools such as Tripwire [10] and AIDE [1] were developed, attackers developed methods of intercepting user requests in

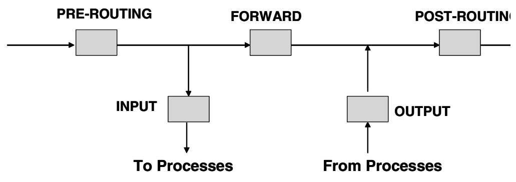

Figure 1. Hooks provided by the Linux netfilter framework

the kernel. This was achieved by hooking into the system call table and the interrupt descriptor table (IDT). Sophisticated rootkits modified machine instructions in the kernel code, which could redirect control to their own system call table without modifying the original one. As kernel integrity monitors started to monitor these immutable data tables, rootkits found ways of hiding by modifying the file system operation handlers and by exploiting discrepancies in the lists used by the scheduler and the process accounting data structures within the kernel.

While the data structures that are tampered have changed over the years, the intent of tampering is still the same, namely to hide the malicious files, process and network connections. Hence, the data structures tampered are all related to intercepting user-level requests and providing doctored responses or more generally, affecting the users view of the system. These rootkit attacks can be full detected if all hiding techniques can be completely explored. In fact, tools such as Strider Ghostbuster [4] detect the presence of rootkits, merely from their attempt to hide.

In this paper, we demonstrate a new class of stealthy attacks that do not try to hide but still evade detection from current generation of integrity monitors. These attacks are symbolic of a larger systemic problem that needs comprehensive analysis. *We also present a classification of kernel data tampering methods based on techniques that we have designed as well as those already known.* To the best of our knowledge, this is the first paper that presents this new class of attacks on kernel data and provides a classification of the techniques used, which is a step towards developing a comprehensive solution.

```
Chain INPUT (policy ACCEPT)
target prot opt source    destination
ACCEPT tcp  --  anywhere  anywhere tcp dpt:ssh
ACCEPT tcp  --  anywhere  anywhere tcp dpt:telnet
ACCEPT tcp  --  anywhere  anywhere tcp dpt:24
REJECT tcp  --  anywhere  anywhere tcp dpt:http reject-with
                                      icmp-port-unreachable

Chain FORWARD (policy ACCEPT)
target     prot opt source    destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source    destination
```

Figure 2. Firewall rules deny admission to web server port

2. Attacks

In this section, we present four stealth attacks that solely change kernel data but do not exhibit hiding behavior. None of these attacks can be detected by the currently known integrity monitoring approaches without prior knowledge of the attack signatures.

2.1. Disable Firewall

This attack hooks into the *netfilter* framework of the Linux kernel and stealthily disables the firewall installed on the system. The user cannot determine this fact by inspecting the system using *iptables*. The rules still appear to be valid and the firewall appears to be in effect. In designing this attack, the goal of the attacker is to disable the network defense mechanisms employed by the target systems, thereby making them vulnerable to other attacks over the network.

Background: *Netfilter* is a packet filtering framework in the Linux kernel. It provides hooks at different points in the networking stack. This was designed for kernel modules to hook into and provide different functionality such as packet filtering, packet mangling and network address translation. These hooks are provided for each protocol supported by the system. The *netfilter* hooks for the IP protocol are shown in Figure 1. Each of the hooks, *Pre-routing, Input, Forward, Output* and *Post-routing*, are hooks at different points in the packets traversal. *Iptables* is a firewall management command line tool available on Linux. *Iptables* can be used to set the firewall rules for incoming and outgoing packets. *Iptables* uses the netfilter framework to enforce the firewall rules. Packets are filtered according to the rules provided by the firewall.

Attack Description: The pointers to the netfilter hooks are stored in a global table called *nf_hooks*. This is an array of pointers that point to the handlers registered by kernel modules to handle different protocol hooks. This data structure is exported even by the latest 2.6 Linux kernel. We modified the hook corresponding to the IP protocol and redirected it to our dummy code, effectively disabling the firewall. The firewall rules that we used during this experiment are shown in Figure 2. The INPUT rules deny admission for incoming traffic to the web server running on the system. Before the attack, we were unable to access this web server externally. After we inserted the attack module, we could access the web content hosted by the web server running on http port (port 80). Running *iptables* command to list the firewall rules still shows that the same rules are in effect (as shown in Figure 2). The user has no way of knowing that the firewall is disabled as the rules appear to be in effect.

Impact: A stealthy attack such as the one described cannot be detected by the existing set of tools. Since our attack module is able to filter all packets without passing it to the

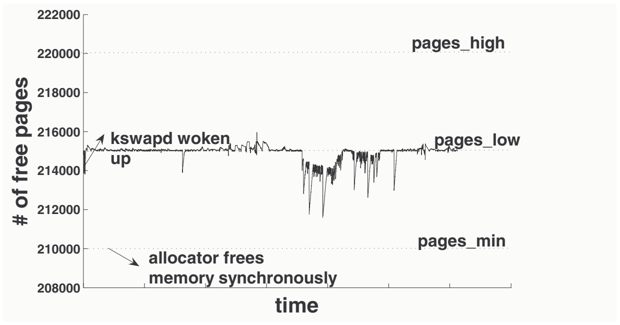

Figure 3. Kernel Memory Allocation: Zone balancing logic and usage of zone watermarks

firewall, it can run other commands upon receipt of a specially crafted packet sent by the remote attacker.

2.2. Resource wastage attack

This attack causes resource wastage and performance degradation on applications by generating artificial memory pressure. The goal of this attack is to show that it is possible to stealthily influence the kernel algorithms by simply manipulating data values. This attack targets the zone balancing logic, which ensures that there are always enough free pages available in the system memory

Background: Linux divides the total physical memory installed on a machine into nodes. Each node corresponds to one memory bank. A node is further divided into three zones: *zone_dma*, *zone_normal* and *zone_highmem*. *Zone_dma* is the first 16MB reserved for direct memory access (DMA) transfers. *Zone_normal* spans from 16MB to 896MB. This is the zone that is used by user applications and dynamic data requests within the kernel. This zone and *zone_dma* are linearly mapped in the kernel virtual address space. *Zone_highmem* is memory beyond 896MB. This zone is not linearly mapped and is used for allocations that require a large amount of contiguous memory in the virtual address space.

Each zone is always kept balanced by the kernel memory allocator called the *buddy allocator* and the page swapper *kswapd*. The balance is achieved using zone watermarks, which are basically indicators for gauging memory pressure in the particular zone. The zone watermarks have different values for all the three zones. These are initialized on startup depending on the number of pages present in the zones. These three watermarks are called *pages_min*, *page_low* and *pages_high* respectively as shown in Figure 3. When the number of free pages in the zones, drops below *pages_low* pages, *kswapd* is woken up. *kswapd* tries to free pages by swapping unused pages to the swap store. It continues this process until the number of pages reaches *pages_high* and then goes back to sleep. When the number of pages reaches *pages_min*, the buddy allocator tries to synchronously free pages. Note that sometimes the number of free pages can go below the *pages_min*, due to atomic allocations requested by the kernel.

Attack Description: The zone watermarks for each zone are stored in a global data structure called *zone_table*. *Zone_table* is an array of *zone_t* data structures that correspond to each zone. Zone watermarks are stored inside this data structure. This symbol is exported even by the 2.6 kernel. The location of this table can be found by referring to the System.map file. We wrote a simple kernel module to corrupt the zone watermarks for the *zone_normal* memory zone. The original and new values for these watermarks are shown in Table 1. We push the *pages_min* and the *pages_low* watermarks very close to the *pages_high* watermark. We also make the *pages_high* watermark very close to the total number of pages in that zone. This forces the zone balancing logic to maintain the number of free pages close to the total number of pages in that zone, essentially wasting a big chunk of the physical memory. Table 1 shows that 210065 (820.56 MB) pages are maintained in the free pool. This attack can be similarly carried out for other zones as well, wasting almost all memory installed on the system. The table indicates that only about 60MB is used and the rest is maintained in the free pool, causing applications to constantly swap to disk. This attack also imposes a performance overhead on applications as shown in Table 2. The three tasks that we used to measure the performance overhead are file copy of a large number of files, compilation of the Linux kernel and file compression of a directory. The table shows the time taken when these tasks were carried out on a clean kernel and after the kernel was tampered. The performance degradation imposed by this attack is consid-

Watermark	Original Value	Modified Value
pages_min	255	210000
pages_low	510	215000
pages_high	765	220000
total free pages	144681	210065
Total number of pages in zone: 225280		

Table 1. Watermark values and free page count before and after the resource wastage attack for the normal zone

Application	Before Attack	After Attack	Degradation (%)
file copy	49s	1m, 3s	28.57
compilation	2m, 33s	2m, 56s	15.03
file compression	8s	23s	187.5

Table 2. Performance degradation exhibited by applications after the resource wastage attack

File #	bday	operm	binrnk6x8	cnt1s	parkinglot	mindist	sphere	squeeze	osum	craps
1	0.765454	0.497607	0.197306	0.000000	0.159241	0.000000	0.893287	0.423572	0.641313	0.147407
2	0.044118	0.180747	0.143452	0.000000	0.012559	0.000000	0.055361	0.769919	0.002603	0.066102
3	0.079672	0.999996	0.467953	0.000000	0.132155	0.000000	0.001550	0.190808	0.032007	0.468605
4	0.009391	0.000334	0.010857	0.000000	0.400118	0.000000	0.000258	0.573443	0.051299	0.057709
5	0.059726	0.996908	0.754544	0.000000	0.065416	0.000000	0.212797	0.276961	0.009343	0.389614
6	0.384023	0.975071	0.003450	0.000000	0.004431	0.000000	0.021339	0.047575	0.139662	0.082087
7	0.002450	0.458676	0.014060	0.000000	0.002061	0.000000	0.000010	0.044232	0.068223	0.836221
8	0.001195	0.840548	0.115478	0.000000	0.192544	0.000000	0.001535	0.024058	0.000078	0.214631
9	0.427721	0.553566	0.138635	0.000000	0.311526	0.000000	0.071177	0.296367	0.003107	0.679244
10	0.654884	0.106287	0.212463	0.000000	0.072483	0.000000	0.212785	0.338967	0.122016	0.710536

Table 3. Results of running the Diehard battery of tests after contamination of the entropy pool

erable.

Impact: This attack resembles a stealthier version of the resource exhaustion attack, which traditionally has been carried out over the network [17, 12, 11]. We try to achieve a similar goal i.e to overwhelm the compromised system subtly by creating artificial memory pressure. This leads to a considerable performance overhead on the system. This also causes a large amount of memory to be unused all the time to maintain the high number of pages in the free pool, leading to resource wastage. The attacker could keep the degradation subtle enough to escape detection over extended periods.

2.3. Entropy Pool Contamination

This attack contaminates the entropy pool and the polynomials used by the Pseudo-Random Number Generator (PRNG) to stir the pools. The goal of this attack is to degrade the quality of the pseudo random numbers that are generated by the PRNG. The kernel depends on the PRNG to supply good quality pseudo random numbers, which are used by all security functions in the kernel as well as by applications for key generation, generating secure session id's, etc. All applications and kernel functions that depend on the PRNG are in turn open to attack.

Background: The PRNG provides two interfaces to user applications namely */dev/random* and */dev/urandom*. The PRNG depends on three pools for its entropy requirements: the primary pool, the secondary pool and the urandom pool. The */dev/random* is a blocking interface and is used for very secure applications. The device maintains an entropy count and blocks if there is insufficient entropy available. Entropy is added to the primary pool from external events such as keystrokes, mouse movements, disk activity and network activity. When a request is made for random bytes, bytes are moved from the primary pool to the secondary and the urandom pools. The */dev/urandom* interface on the other hand is non-blocking. The contents of the pool are stirred when the bytes are extracted from the pools. A detailed analysis of the Linux random number generator is available in [6].

Attack Description: This attack constantly contaminates the entropy pool by writing zeroes into all the pools. This is done by loading an attack module that consists of a kernel thread. The thread constantly wakes up and writes zeroes into the entropy pools. It also attacks the polynomials that are used to stir the pool. Zeroing out these polynomials nullifies a part of the extraction algorithm used by the PRNG. The location of the entropy pool is not exported by the Linux kernel. We can find the location by simply scanning kernel memory. Entropy pool has the cryptographic property of being completely random [15]. Since we know the size of the entropy pools, this can be found by running a sliding window of the same sizes through memory and calculating the entropy of the data within the window. Kernel code and data regions are more ordered than the entropy pools and have a lower entropy value. The pool locations can therefore be successfully located.

We measured the quality of the random numbers generated by using the diehard battery of tests [2]. The results are summarized in Table 3. Diehard is the suite of tests used to measure the quality of random numbers generated. Any test that generates a value extremely close to 0 or 1 represents a failing sequence. More about the details of these tests can be found in [2]. We run the tests over ten different 10MB files that were generated by reading from the */dev/random* device. The table shows that the sequence that is generated after attack, fails miserably in two of the tests: *cnt1s* and *mindist* and partially in the others. A failure in any one of the tests means that the PRNG is no longer cryptographically secure.

Impact: After the attack, the generated pseudo random numbers are of poor quality, leaving the system and applications vulnerable to cryptanalysis attacks.

2.4. Disable Pseudo-Random Number Generator (PRNG)

This attack overwrites the addresses of the device functions registered by the PRNG with the function addresses of the attack code. The original functions are never invoked. These functions always return a zero when random bytes are requested from the */dev/random* or */dev/urandom* devices. Note that though this appears similar to the attack by traditional rootkits that hook into function pointers, there is a subtle difference. Since this particular device does not affect user-level view of objects, this is not a target for achieving hiding behavior and hence, not monitored by kernel integrity monitors.

Attack Description: The kernel provides functions for reading and writing to the */dev/random* and */dev/urandom* devices. The data structures used to register the device functions are called *random_state_ops* and *urandom_state_ops* for the devices */dev/random* and */dev/urandom* respectively. These symbols are exported by the 2.4 kernel but are not exported by the 2.6 kernel. We could find this data structure by first scanning for function opcodes of functions present within *random_state_ops* and *urandom_state_ops*. We then used the function addresses in the correct order to find the data structure in memory. Once these data structures are located in memory, the attack module replaces the genuine function provided by the character devices with the attack function. The attack function for reading from the device simply returns a zero when bytes are requested. After the attack, every read from the device returns a zero.

Impact: All security functions within the kernel and other security applications rely on the PRNG to supply pseudo random numbers. This attack stealthily compromises the security of the system, without raising any suspicions from the user.

3. Categorizing Attacks

We have identified several attack categories based on the tampering techniques employed. The categories are derived from the techniques used by rootkits in existence as well as the new class of attacks that we have designed. These attacks span across static as well as dynamic data in the kernel. The first two categories described, namely *control hijacking* and *control interception*, involve directly changing the control path in the kernel by manipulating jump tables or function pointers. The last three categories, namely *control tapping*, *data value manipulation* and *inconsistent data structures*, work solely by manipulating non-control data.

Our motivation behind creating these categories is to identify the broader systemic problem in the kernel, rather than the individual attacks themselves. This helps in building defense techniques that are generic and that can be applied comprehensively throughout the kernel to protect all data structures that are vulnerable to a given category of attacks.

3.1. Control Hijacking

Control hijacking attack is a form of manipulating the control flow within a kernel control path. This attack redirects the control flow to the attack code and the original code is never actually invoked. The attacks that we designed in this category are (a) disable firewall attack and the (b) disable PRNG attack. All layers in the kernel, originally put in for extensibility and to provide a common interface, can be abused to perform such an attack. All jump tables and function pointers are also susceptible to this form of attack.

3.2. Control Interception

Control interception was the technique used by most traditional rootkits that changed the system call table, IDT and the kernel code. These attacks intercept the kernel control path in such a way that control first flows to the attack code. The attack code then calls the original code. This way, the attacker is able to filter requests to and responses from the original code. Control interception is typically used for hiding the attacker's files, processes and network connections. All layers, jump tables and function pointers are susceptible to this form of attack as well.

3.3. Control Tapping

Control tapping ensures that the attack code is invoked en route to the original function. In other words, the interception takes place in such a way that the attack code is not able to manipulate the arguments and results of the original function being called. The only assurance for this type of interception is that the attack code will be invoked on every call to the original function. One example of this form of attack is the attack hooking to the *execve* system call done by registering a new binary format. This attack is discussed in [3].

3.4. Data Value Manipulation

These attacks rely on manipulating values of critical variables, which in turn directly or indirectly influence the algorithms used by the kernel. Defending from such an attack requires a close analysis of data structure values and some form of value based monitoring. The attacks that we developed that fall in this category are (a) resource wastage attack and (b) entropy pool contamination attack.

3.5. Inconsistent Data Structures

This class of attacks makes kernel data structures inconsistent, which are otherwise supposed to be consistent during normal operation. Two common known methods used are process hiding and module hiding. Process hiding is achieved by the fact that the kernel uses separate lists for scheduling and accounting. The malicious process is removed from the accounting list but not from the scheduler list, so the process is still scheduled. Module hiding is done by removing the module entry from the module list after the module is loaded in memory, making it invisible accounting commands.

4. Related Work

Garfinkel et al [5] proposed to use virtual machine based introspection. Zhang et al [18] proposed the use of a secure coprocessor that can verify the integrity of the kernel. Petroni et al [7] demonstrated a prototype that could successfully monitor the integrity of kernel code and static tables from a secure coprocessor. In another recent work [8], they also built a specification based compiler that could compile high-level manually specified constraints and monitor for those constraints within kernel dynamic data. This work however requires the user to either know the attack signature or anticipate it and generate a specification. Several attestation based approaches have been proposed to verify the integrity of running code [14, 9, 16, 13]. These approaches use a secure chip as the trusted computing base to bootstrap trust. The verification procedure is based on comparing hashes with known good values or timing calculations. While these approaches work well for checking the integrity of code, they cannot check the integrity of data.

5. Conclusion and Future Work

In this paper, we have demonstrated a new class of stealth attacks that do not employ the traditional hiding behavior used by rootkits. We have designed attack prototypes to demonstrate that such attacks are realistic and indicative of a more systemic problem in the kernel. Furthermore, they cannot be detected by currently known monitoring approaches without prior knowledge of the attack signatures. We have also classified the data tampering techniques used by all known kernel tampering malware. As part of future work, our goal is to design an automated comprehensive integrity monitor for kernel data.

Acknowledgments

We would like to thank Joe Kilian for his insightful comments and discussions on this work and early drafts of the paper. We would also like to thank Trent Jaegar (shepherd) and the anonymous reviewers for their feedback in refining the paper. This work has been supported in part by the NSF CAREER grant CCR-0133366.

References

[1] Advanced intrusion detection environment. http://sourceforge.net/projects/aide.

[2] The marsaglia random number cdrom including the diehard battery of tests of randomness. G Marsaglia - See http://stat.fsu.edu/pub/diehard, 1996.

[3] Registration weakness in linux kernel's binary formats. Shellcode - See http://www.packetstormsecurity.org/papers/general/binfmt-en.pdf, 2006.

[4] D. Beck, B. Vo, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*.

[5] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, 2003.

[6] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006.

[7] N. L. P. Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.

[8] N. L. P. Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *USENIX Security Symposium*, 2006.

[9] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. 12th USENIX Security Symposium, Washington DC, 09 2003.

[10] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, 1994.

[11] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 24(2):115–139, 2006.

[12] C. L. Schuba, I. V. Krsul, M. G. Kuhn, E. H. spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on tcp. *sp*, 00:0208, 1997.

[13] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. K. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP*, pages 1–16, 2005.

[14] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. *sp*, 00:272, 2004.

[15] A. Shamir and N. van Someren. Playing "hide and seek" with stored keys. In *FC '99: Proceedings of the Third International Conference on Financial Cryptography*, pages 118–124, London, UK, 1999. Springer-Verlag.

[16] E. Shi, A. Perrig, and L. van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *IEEE Symposium on Security and Privacy*, pages 154–168, 2005.

[17] H. Wang, D. Zhang, and K. Shin. Detecting syn flooding attacks. 2002.

[18] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, 2002.