Apate - A Linux Kernel Module for High Interaction Honeypots

Christoph Pohl
MuSe - Munich IT Security Research Group
Munich University of Applied Sciences
Munich, Germany
Email: christoph.pohl0@hm.edu

Michael Meier
Fraunhofer FKIE Cyber Defense
University of Bonn
Bonn, Germany
Email: michael.meier@fkie.fraunhofer.de

Hans-Joachim Hof
MuSe - Munich IT Security Research Group
Munich University of Applied Sciences
Munich, Germany
Email: hof@hm.edu

***Abstract*—Honeypots are used in IT Security to detect and gather information about ongoing intrusions, e.g., by documenting the approach of an attacker. Honeypots do so by presenting an interactive system that seems just like a valid application to an attacker. One of the main design goals of honeypots is to stay unnoticed by attackers as long as possible. The longer the intruder interacts with the honeypot, the more valuable information about the attack can be collected. Of course, another main goal of honeypots is to not open new vulnerabilities that attackers can exploit. Thus, it is necessary to harden the honeypot and the surrounding environment. This paper presents Apate, a Linux Kernel Module (LKM) that is able to log, block and manipulate system calls based on preconfigurable conditions like Process ID (PID), User Id (UID), and many more. Apate can be used to build and harden High Interaction Honeypots. Apate can be configured using an integrated high level language. Thus, Apate is an important and easy to use building block for upcoming High Interaction Honeypots.**

Keywords–Honeypot; Intrusion Detection; Linux Kernel; Rule Engine

I. INTRODUCTION

Honeypots are well known tools for Intrusion Detection and IT Security research. Usually, honeypots fall into one of two classes: Low Interaction Honeypots and High Interaction Honeypots. A Low Interaction Honeypot simulates attackable services, systems, or environments whereas a High Interaction Honeypot [1][2] offers a real exploitable service, system, or environment. As in most cases a honeypot is not a productive system, every activity on a honeypot is either unintended use or an attack.

When deploying a High Interaction Honeypot, it is necessary to harden the honeypot to avoid attackers gaining unintended control of the system running the honeypot. A High Interaction Honeypot should by definition be exploitable, but it should prevent annoying or harmful operations on the honeypot system. Another important requirement for High Interaction Honeypot is to log as much information as possible about the state of the system and about ongoing intrusions. Therefore, a High Interaction Honeypot needs a highly flexible way to decide which information should be logged and which should not. Apate offers such a flexible way, using a high-level language for configuration. Also, it should be possible to log information on a as fine granular level as possible. Apate offers a logging on system call level. Manipulation of system calls, depending on user interaction or the system environment, is necessary to provide High Interaction Honeypot functionalities. This allows the honeypot provider to present different environments depending on PID, UID (and many more), or system call parameters. For example, the High Interaction Honeypot provider is able to present one file structure to PID 42 and a completely different file structure to PID 43. This manipulation can be used to decoy an attacker. Furthermore, it can also be used to suppress harmful actions. The honeypot admin is able to prevent execution of system call. Blocking a system call can be done by really blocking (not calling the real system call), or in a more sophisticated way. At last, it is necessary that High Interaction Honeypot components (like the proposed LKM) should be hard to detect for intruders. This requirement calls for sophisticated technologies, already known from rootkits. For productive use, it is necessary that a High Interaction Honeypot module has only low computational overhead. An attacker should not be able to detect a High Interaction Honeypot by observing performance leaks.

Apate is a Linux Kernel module that fulfills all requirements mentioned above. Hence, it is an important building block for High Interaction Honeypots.

The rest of this paper is structured as follows: Section II provides an overview on related work. Section III describes the design and implementation of Apate in detail. Section IV shows the evaluation of Apate. Section V concludes the paper.

II. RELATED WORK

A well known honeypot tool, based on LKM for 2.6 Linux Kernel, is Sebek [3][4]. Sebek is primarly used for logging purposes in High Interaction Honeypot. Thus, it provides several methods for detailed logging (like logging via network or GUI). In [5][6], ways to detect Sebek are described. Sebek does not provide the possibility to manipulate system calls, hence it does not offer such a fine-grain information logging as provided by Apate.

Another approach for monitoring systems is to use virtual machine introspection and system view reconstruction. For example, [7], [8], and [9] use this approach. Introspection realized on hardware level of the virtual machines offers a stealthier approach then Apate. However, Apate provides additional means to manipulate the behavior of system calls, which are not supported by [7], [8], and [9], hence Apate is superior to these approaches.

SELinux [10] is a well known tool for inserting hooks at different locations inside the kernel. Such an approach provides

 ISBN: 978-1-61208-427-5

access control for critical kernel routines. SELinux can be controlled on a very fine granular level with an embedded configuration language. Although SELinux is very useful in hardening a kernel, it is not designed for honeypot purposes. Especially, it lacks in the possibility to decoy the attacker using "wrong" information.

Grsecurity [11] with PAX [12] is similar to Apate. However, it greatly differs in ease of deployment and ease of configuration [13]. It also lacks in the possibility to decoy the attacker with "wrong" informations.

In conclusion, non of the mentioned related work fulfills all requirements listed in Section I. Apate fulfills all requirements, hence is a useful building block for upcoming High Interaction Honeypots.

III. Design and implementation

Apate intercepts system calls and allows to execute custom code in these calls. Figure 1 shows the interception strategy of Apate.

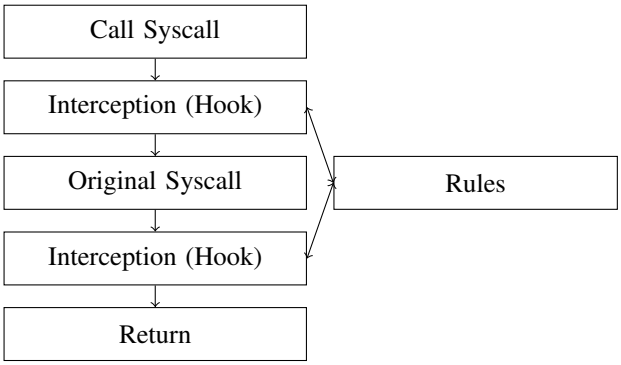

Figure 1. interception strategy of Apate

Apate does not manipulate the syscall table to prevent detection (see Subsection III-D for details). Apate intercepts the syscall within the syscall target, i.e., the real syscall address is called but Apate jumps immediately to the interception routine (after consuming some decoy assembler code). The hook decides on the action to invoke, based on the rules for this system call. Within this action, it is able to manipulate, block and/or log a system call. The following hooks are implemented in Apate :

- sys_open, sys_close, *sys_open*
- sys_read, sys_write, sys_unlink
- sys_execve
- sys_getpid, sys_getuid
- sys_mkdir, sys_rmdir
- sys_getdents

This paper focuses on the usage of system calls that are related to File IO and execution control as these system calls are usefull for hardening High Interaction Honeypots.

A. Configuration

Apate can be configured in a very flexible way as can be seen in Figure 2. The configuration file *rules.apate*, written in a high level language (see sectionIII-B for details), gets compiled by the Apate compiler, resulting in the file *apaterules.c*. Together with the original source code, the compiler generates the Apate LKM. The resulting LKM can be loaded into kernel

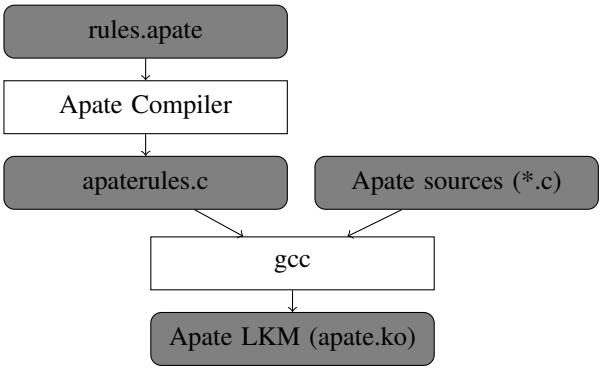

Figure 2. Configuration workflow of Apate

with common insmod util. Once loaded, the ruleset is active.

The configuration consists of rulesets. A ruleset is an ordered list of rules. A system call gets intercepted when one or more rules match. One system call can have more than one matching rule with different decision parameters. There are three major types of decision parameters:

- Parameters that are system call independent like PID, UID, SSID (in fact every variable from *struct task_struct* [14] could be used for conditions).
- Parameters that are dependent to the specified system call. Often, these are function parameters like paths.
- Parameters that are defined by functions. This decision parameter allows to build reactive systems. For example, one can define a condition which reads some file, whenever the file contains a keyword the condition could be true.

Rules are defined as stated below:

Let be $true = 1$ and $false = 0$. Let $c(a, b)$ be a condition such that:

$$\begin{aligned} C &: A \times B \to \{0, 1\} \\ (a, b) &\mapsto c(a, b) \end{aligned} \tag{1}$$

, where $a \in A$ and $b \in B$ are two parameters, which are used by $c()$ for the calculation of the condition. The parameters are further called decision parameters. For example, a decision parameter could be the path of the system call sys_open*()* and the related condition is *if (param[0] == "/etc/passwd") ? 1 : 0* where $a = param[0]$ and $b = "/etc/passwd"$. This rule matches system calls trying to get access to the file $"/etc/passwd"$

Let $cb(d, e, f)$ be a condition block. A condition block calculates the result of conditions or other conditionblocks with *AND* or *OR*. A condition block uses two parameters d and e and an operator f. d and e can be the result of any $c(a, b)$ or another $cb(d, e, f)$.

CB is the set of all possible condition blocks:

$$\begin{aligned} CB &: \{0, 1\} \times \{0, 1\} \times \{AND, OR\} \to \{0, 1\} \\ (d, e, f) &\mapsto cb(d, e, f) \end{aligned} \tag{2}$$

Further, the set of all conditions and condition blocks is AC such that

$$AC = C \cup CB \tag{3}$$

For $cb(d, e, f)$ let $d, e \in AC$ and $f \in \{2, 4\}$. When $f = 2$ the operator AND will be used. When $f = 4$ the operator OR will be used. $c^*(a, b)$ is a condition that returns always true. The second parameter in the conditionblock can be neutralised with $cb(d, c^*(0, 0), 2)$

This leads to the definition

$$cb(d, e, f) = \begin{cases} 1 & \text{if } (d + e) * f \geq 4 \\ 0 & \text{other} \end{cases} \tag{4}$$

This definition makes it possible to group different conditions and to be aware of precedences.

Let A be the set of atomic actions. An atomic action is a function that provides only one single functionality. For example, an atomic action can be the redirection of a system call. An action $a \in A$ falls in one of three groups:

- Manipulating actions
- Logging actions
- Blocking or emergency exit actions

Let AS be an orderd list of actions. The index function $i(x)$ assigns an index to each element $x \in AS$, hence

$$AS = \{x \in AS | 0 < i(x - 1) < i(x)\} \tag{5}$$

Let AAS be the lists of all actions. A rule $r^{g,h}$ consists of one condition block $g \in CB$ and an action set $h \in ASS$. Let R be the set of all rules. Whenever the condition block returns 1, the action set h is started.

Let RS be a list of sorted rules ($RS \in R$). Each element of RS has a flag fl. A flag is defined as

$$fl \in \{exit = 1, \neg exit = 0\} \tag{6}$$

When a system call gets called, all rules in RS are calculated beginning with the first rule in RS and until a rule is in state $true$ and $fl = 1$.

Using the definitions above, a highly configurable system could be build. Including some basic predefined conditions enhances convenience, e.g., equality checks for integer, floats or strings.

B. Configuration High Level Language

The configuration language implements two main requirements: first, the configuration language should be flexible, including the ability to reuse patterns, store variables, calculate with operators, embed external functions, define functions, and use decision statements. This allows to use the language to describe even very complex scenarios. Second, the configuration language should provide a transparent way to define rules, related to honeypots (or in scope of this paper to control and manipulate system calls). To deal with these requirements, the Apate language combines concepts known from functional programming (in this case Haskell [15]) with a concept well known from packet filter configuration (in this case pf [16]).

This Section gives a brief introduction to the important parts of the language. For the sake of clarity, some convenience features of the Apate language (e.g., embedded C, self defined functions, loops) are omitted.

Listing 3 shows some example source code for the Apate language.

```
define c1,c2,c3 as condition
define r1,r2 as rule
define a1,a2 as action
define cb1 as conditionblock
define rc1 as rulechain
define sy1 as syscall

let c1 be testforpname
let c2 be testforparam
let c3 be testforuid
let a1 be manipulateparam
let a2 be log
let sy1 be sys_open

let cb1 be {(c1("mysql") && c2(0;"/var/\
    lib/mysql/*"))}

let r1 be {cb1->a1(0;"/var/lib/mysql/*" \
  ;"/honey/mysql/")}
let r2 be {{c3(">",0)}->a2()}
let rc1 be {r2,:r1} // :defines exit

bind rc1 to sy1
```

Figure 3. Example Sourcecode Apate language

The first block with the `define` statements binds variables to different types (like condition, rules, or functions). The code block with the `let` statements points these variables to values or functions. In this case, it defines 3 conditions (`c1,c2,c3`). `c1` will test the actual process name against another string. `c2` tests if a param of the actual syscall is equal to a given value. `c3` tests if the actual uid is equal to a given value. `a1,a2` are actions. `a1` manipulates a parameter of the actual system call. `a2` logs a system call. The variable `cb1` represents a condition block. Its `let` assignment also shows that it is possible to write nested variable assignements. In this case, the conditions `c1,c2` are combined with `&&` (*AND*). In the same line, the conditions `c1,c2` gets assigned with parameters. In this case the condition `c1` checks if the current parent process is the mysql-Process. `c2` checks if the first parameter (`0`) of the current system call is equal to */var/lib/mysql/** . The asterisk describes a wildcard function. The rule assignment for `let r1 be...` binds a conditionblock to an action. In this case, it means whenever the conditionblock returns true the action `a1` rewrites the first param of the current system call. It replaces */var/lib/mysql* with */honey/mysql* . The rule `r2` logs the current system call whenever the current UID is greater than 0. A ruleset (rulechain) `rc1` is assigned with `r2,r1`. The `r1` rule is also assigned as exit rule (`..:r1..`). When this rule fires, no further rules will be called. In the last line, the rule chain `rc1` is bound to the system call *sys_open*.

In conclusion, when the system call *sys_open* gets called,

the parent process is the mysql process and the system call parameter (in this case the path which should be opened) begins with */var/lib/mysql/**, this syscall gets manipulated and the syscall will open a file under */honey/mysql/...* . The second rule means that every call for *sys_open* will be logged, except when the root user calls this system call.

C. Manipulation of System Calls

If a rule matches, the corresponding action chain gets called to manipulate the original system call. An action chain has a length l with $1 \leq l < n$.

Figure 4 shows an example for the manipulation strategy. Functions prefixed with *f_* are actions.

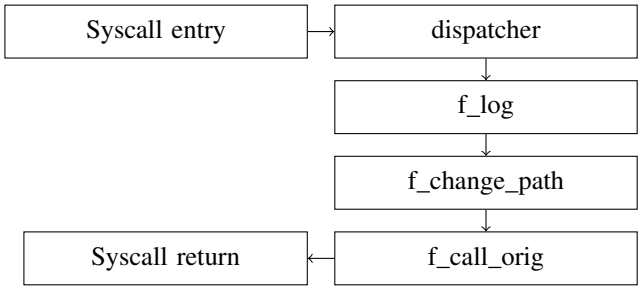

Figure 4. Conceptual Manipulation Strategy

The dispatcher represents the rule engine, deciding which action chain should be used. In this example the first action logs the system call. The next action manipulates some parameter like a path or anything else. The *f_call_orig* calls the original system call with the manipulated parameter. The result gets returned to the callee.

Technically, one action is a function that consumes all system call parameters, including the current *struct task_struct* and a pointer to the *syscall_result* variable. Each function returns an Integer, indicating wether the function call has been successful or not. Whenever a function returns an error, the action chain gets disrupted and an error routine is called. Finally, the hook returns the *syscall_result*. In case of an error the system call returns a system call dependent error.

D. Hiding Hooks and LKM

An attacker should not be able to detect Apate, otherwise Apate would not be suitable for High Interaction Honeypots. As hiding software in all use cases is very difficult, Apate must at least hide itself until the effort of detection of Apate is unreasonable high for an intruder. This requires to define which effort is unreasonable high for an attacker and which is not. The following actions are defined as reasonable for an attacker, hence should be prevented:

- Testing for module presence with standard utils like *lsmod,modinfo,...* or misleading errors when using *insmod* and similar tools
- Testing for presence of module in */proc/module* and */sys/module*
- Testing for presence of Apate related logfiles, configurations, and other artifacts

To hide Apate, it is necessary to remove the module from the module list. Simplified, all modules are represented in a global linked list. By using

```
list_del_init(&__this_module.list);
```

the module is removed and therefore invisible. To hide from the */sys/module* Apate uses

```
kobject_del(&THIS_MODULE->mkobj.kobj);
```

to remove itself from this representation. With these modification, the module is invisible to standard utils (they use */proc/module*) and in */sys/module* . These technologies are also well known rootkit technologies see for example [18][19].

Apate does not use any configuration files beyond the configured rules. The high level language should be deleted by the honeypot admin after its compilation into Apate. Hence, Apate cannot be identified by an attacker looking for configuration files.

Apate is used to cloak logfiles: predefined rules in Apate prevent all users to see, read, or write Apate logfiles. To gain access to the logfile, a system administrator need to restart the host system without the honeypot.

To detect a hook, an intruder needs to analyze physical memory. Apate makes it hard to load a new module into the kernel. It prevents to load another kernel module by overriding the flag that controls the module loading ability. Beyond the possibilities of Apate, the honeypot admin can harden the host system to ensure that this dumping has a high effort for an intruder.

Apate has different opportunities to insert hooks into system calls. By default, Apate changes the function pointer in the system call table. This is sufficient as long as the intruder has no possibility to compare the original table with the hooked table. If this is not enough protection, the admin can decide to harden the system with some anti-rootkit technologies. This makes it impossible for Apate to overwrite the jump points. Figure 5 shows the alternative hooking technology. This

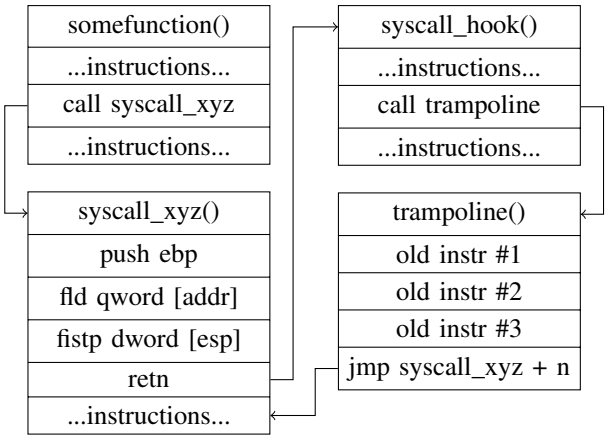

Figure 5. Hooking using a so-called trampoline

technology is well known from Windows and Linux rootkits. During the hooking process, Apate stores the first n bytes of the target system call function. The stored commands will be copied to a trampoline function. Instead of the original commands, Apate injects a jump operation. This lets the process jump into the hooking function immediately after entering the original system call function. Whenever the hooking function calls the original system call, it calls the trampoline function.

 ISBN: 978-1-61208-427-5

The original code is executed, then the trampoline function lets the process jump into the original function with an offset of n bytes. The trampoline is a feature to obfuscate the hook for rootkit detection tools and uses live patching technologies. Thus, it can be detected with core dump disassembling. This is out of scope of this paper as it is assumed that the effort to detect the honeypot with disassembling tools is too high.

IV. EVALUATION

There are three major goals for Apate. The first goal is to provide a highly flexible configuration. Although the proposed configuration system works well and is suitable for High Interaction Honeypots, it must be ensured that every possible combination of rulesets and actions can be described. This means that the configuration language must be turing complete. For this, it is determined that an action a is able to decide which rule from the ruleset should be invoked next. This means it can jump to any other rule from a given ruleset. It is also determined that Apate has an array (in this case an impossible array with infinite indices which can hold any other type (like actions, rules, other defined variables or anything else)). Technically, Apate has a register and a stack. Last, it is determined that an action is able to fill or read any index of this array. Together with actions for calculations and conditions for jump decisions, the system is turing complete. In fact any action is just a C-Function and the *define* statement creates variables.

The second goal is to provide a system which achieves a suitable level of stealthiness. As described in Subsection III-D, the system hides itself from common util tools like *lsmod, modinfo, modprobe, insmod* . Apate is also not available in */proc/module* or */sys/module* . To test for presence in any logfile a simple grep command with typical signatures for Apate (Simplified each log entry or configuration includes the string "apate", thus it is easy to detect it) is fired on the full system. However with proper rules these log entries are not visible by standard system commands.

The third goal is that Apate should be efficient. Performance tests should assure that Apate is able to serve under productive usage scenarios. The most important performance factor is the overhead of logging. To evaluate the performance of Apate in a productive scenario, the execution time of *sys_open* , *sys_write* ,*sys_read* , and *sys_close* are measured. The *sys_open* and *sys_close* get called just once a file is opened or closed. The *sys_write* and *sys_read* get called more often (under the condition that heavy writing will be done on the system). Thus, the test pattern concentrates on *sys_write* and *sys_read* . For the performance evaluation, data is copied from one file to another using increasing file lengths. This will be done for 100 times for each file size. The source file is generated on the fly from */dev/random* before each copy command. After each successful copy command the target file is deleted. A Gentoo 64 Bit system with 32 GB Ram and 16 Cores is used for all performance tests. The kernel is optimized by disabling unnecessary drivers and by enabling some debugging flags. One source file is generated for each size with random bits and a length of $l(file)$ bytes. Let the size of the file be:

$$0 < l < 1,000,000,000 \tag{7}$$

TABLE I. PERFORMANCE MEASUREMENT

Measurement	m_1	m_2	m_3	m_4
Measurements	110,800	110,800	110,800	110,800
Unique Filesizes	1,108	1,108	1,108	1,108
sd(runtime sec)	0.1066	0.2421		0.2452
var(runtime sec)	0.0114	0.0586		0.0601
iqr(runtime sec)	0.0010	0.0026		0.0023

and

$$l_n(file) = \begin{cases} l_{n-1} + 1 & \text{if } l_n < 1,000,000 \\ l_{n-1} + 1,000 & \text{if } 1,000,000 \leq l_n \\ & \wedge l_n < 100,000,000 \\ l_{n-2} + 1,000,000 & \text{if } 100,000,000 \leq l_n \\ & \wedge l_n \leq 1,000,000,000 \end{cases} \tag{8}$$

Four different settings were tested.

The first setting (m_1) is used as reference setting. It does not use any interception.

The second setting, m_2, uses only one rule which always returns true. The related action set calls the origin system call and logs this action. This is the shortest way in Apate to provide logging functionality. This testing is used to evaluate the logging overhead of Apate.

The third and fourth setting, m_3 and m_4, evaluate the influence of rules. Each rule consists of 50 conditions with $\{c_0, c_1, \ldots, c_{50}\}$ where each condition is combined with an *and* statement. The last condition returns false. Each test uses 50 rules. The last condition in rule number 50 (last rule) returns true. Overall, each system call passes 2500 conditions. This triggers an action set that will call the original system call (m_3 and m_4) and then logs this action (only m_4).

Table I shows the results of the performance evaluation. The *sd* -row shows the standard deviation, *var* shows the variance, and *Iqr* shows the interquartile range.

Figure 6 shows the correlation between file size and runtime. For every curve the median of the measured runtimes for each unique file size is connected with a line. The m_1 curve shows the reference setting. The m_2 curve shows that the logging component has a big influence on performance. Each syscall and its values were logged. Each log was sent to another server using UDP. Gentoo uses a buffer with 65,365 Byte. To copy a file with one Gigabyte it needs 32,720 syscalls. This explains the overhead of m_2 and m_4. The m_3 curve shows that the rule engine works with just a small overhead when only conditions get processed. For one measurement with a file size of one Gigabyte, the engine processed 81,800,000 conditions. However, to copy a file with less than 65,365 Byte only 4 syscalls are passed and therefore only 10,000 conditions gets processed.

In conclusion, these measurements shows that it is possible to build a syscall interception framework which is able to provide proper configuration with acceptable overhead. The evaluation does not show a single case that prevents a productive usage of Apate.

 ISBN: 978-1-61208-427-5

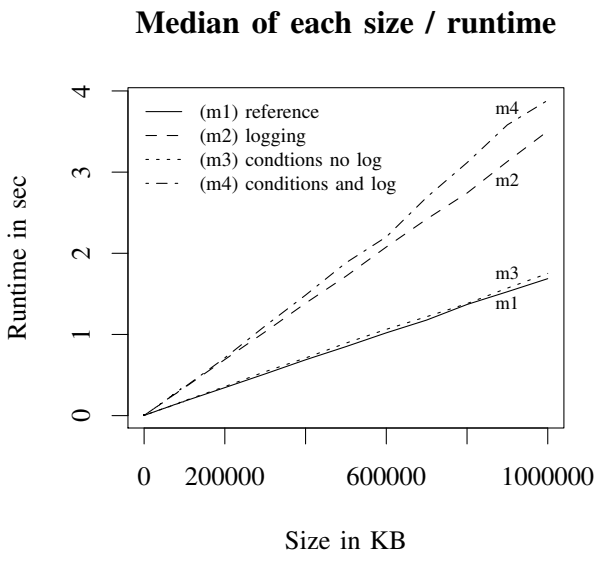

Figure 6. Relation between runtime/filesize/rules *sys_open*

V. CONCLUSION

This paper presented Apate, a Linux Kernel Module for hardening High Interaction Honeypots. Apate works on a system call level, is able to log, block and manipulate these calls, and uses an easy to use yet powerful configuration language. The evaluation shows that Apate has a moderate performance overhead and can be used in productive honeypot systems. Apate is also stealthy enough for most common usage scenarios. Overall, Apate is an ideal basis and important building block for upcoming High Interaction Honeypot Systems.

REFERENCES

[1] C. Pohl and H.-J. Hof, "The All-Seeing Eye: A Massive-Multi-Sensor Zero-Configuration Intrusion Detection System for Web Applications," in SECURWARE 2013, The Seventh International Conference on Emerging Security Information, Systems and Technologies, 2013, pp. 66–71.

[2] C. Pohl, A. Zugenmaier, M. Meier, and H.-J. Hof, "B.Hive: A Zero Configuration Forms Honeypot for Productive Web Applications," in International Conference on ICT Systems Security and Privacy Protection (IFIP SEC 2015), 2015.

[3] Honeynet Project, "Know Your Enemy: Sebek," 2003. [Online]. Available: http://old.honeynet.org/papers/sebek.pdf

[4] E. Balas, "Sebek: Covert Glass-Box Host Analysis," ;login: THE USENIX MAGAZINE, no. December 2003, Volume 28, Number 6, 2003. [Online]. Available: https://www.usenix.org/publications/login/december-2003-volume-28-number-6/sebek-covert-glass-box-host-analysis

[5] T. Holz and F. Raynal, "Detecting honeypots and other suspicious environments," in Proceedings from the Sixth Annual IEEE Systems, Man and Cybernetics (SMC) Information Assurance Workshop, 2005. IEEE, 2005, pp. 29–36. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1495930

[6] M. Dornseif, T. Holz, and C. Klein, "NoSEBrEaK - Attacking Honeynets," in Proceedings of the 2004 IEEE Workshop on Information Assurance and Security, Jun. 2004. [Online]. Available: http://arxiv.org/abs/cs/0406052

[7] C. Song, B. Ha, and J. Zhuge, "Know Your Tools: Qebek – Conceal the Monitoring — The Honeynet Project," http://www.honeynet.org/papers/KYT_qebek, visited 25.02.2015. [Online]. Available: http://www.honeynet.org/papers/KYT_qebek

[8] T. K. Lengyel, J. Neumann, S. Maresca, B. D. Payne, and A. Kiayias, "Virtual machine introspection in a hybrid honeypot architecture," in CSET'12: Proceedings of the 5th USENIX conference on Cyber Security Experimentation and Test. USENIX Association, Aug. 2012.

[9] X. Jiang and X. Wang, ""Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots," in Recent Advances in Intrusion Detection. Springer Berlin Heidelberg, 2007, pp. 198–218. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-74320-0_11

[10] S. Smalley, C. Vance, and W. Salamon, "Implementing selinux as a linux security module," NAI Labs Report, vol. 1, no. 43, 2001, p. 139.

[11] Open Source Security, "grsecurity," https://grsecurity.net, 2015, visited 25.02.2015.

[12] PAX Team, "Pax," https://pax.grsecurity.net, 2015, visited 25.02.2015.

[13] M. Fox, J. Giordano, L. Stotler, and A. Thomas, "Selinux and grsecurity: A case study comparing linux security kernel enhancements," 2009.

[14] L. Torvalds, "Linux kernel release 3.x source linux/sched.h," https://github.com/torvalds/linux/blob/master/include/linux/sched.h, 2015, visited 25.02.2015.

[15] S. Marlow, "Haskell 2010 language report," https://www.haskell.org/onlinereport/haskell2010/, 2010, visited 25.02.2015.

[16] OpenBSD, "Pf: The openbsd packet filter," http://www.openbsd.org/faq/pf/, 2015, visited 25.02.2015.

[17] C. Pohl, "Github apate sourcecode gpl2," https://github.com/c00clupea/apate, 2015, visited 25.02.2015.

[18] M. H. Ligh, A. Case, J. Levy, and A. Walters, The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. John Wiley & Sons, 2014.

[19] T. B. (TurboBorland), "Modern linux rootkits 101," http://turbochaos.blogspot.de/2013/09/linux-rootkits-101-1-of-3.html, 2013, visited 25.02.2015.

 ISBN: 978-1-61208-427-5