Ubiquitous Redirection as Access Control Response

George Bakos
gbakos@ists.dartmouth.edu
Institute for Security Technology Studies
Dartmouth College
45 Lyme Road
Hanover, New Hampshire, 03755 USA

Sergey Bratus
sergey@cs.dartmouth.edu
Institute for Security Technology Studies
Dartmouth College
45 Lyme Road
Hanover, New Hampshire, 03755 USA

Abstract

Rule-based access control mechanisms, network firewalls and application input validation all serve to enforce security policy. When violating the acceptable conditions these defenses mandate, an unauthorized requester is generally turned away. We make an argument for a modification to traditional access limitation through redirection and deceptive completion across many layers of data communication. Ubiquitous redirection provides additional information on attacker behavior, consumes attacker resources, improving defender awareness and, ultimately, site security. We describe a variety of network-based techniques for deception implemented in our honeypots, and undertake a study of OS-level deception practiced by rootkit writers with the view towards prospective use of similar techniques for defensive applications.

Keywords: Honeypot, redirection, perimeter defense, deception systems, kernel instrumentation

1 Introduction

Traditional access control mechanisms enforce security policies by denying access to an information resource unless the accessing subject has been defined as an authorized user of that resource, and can successfully provide any required credentials. In addition, the resource being accessed must be present and available across the channel used. Should the resource not be available, a separate class of denial is encountered, with correspondingly different feedback to the requester. Valuable information can be gained by an unauthorized user in the measurement of this denial, and techniques such as *inverse scanning* are used to enumerate elements of a security architecture through these responses or the lack thereof [15, 12, 3].

In addition to network and transport layer information, most service requests include an appropriately encapsulated payload which will be validated by the client and server at the ends of the communication roadway for appropriate content and formatting. This validation generally is performed at the application layer, be it implemented in a service-specific proxy or the communication end-point. Responses may or may not be delivered upon inspection failure, determined by system capabilities and local policies. These responses can offer valuable information to an attacker.

We propose *Ubiquitous redirection*, or the perceived completion of *all* unauthorized access attempts, as a full or partial replacement for legacy failure responses. Through ubiquitous redirection, the perceived successes place strains on the attackers' result-parsing systems and provide the defender with a wealth of data. Such high-fidelity attack data is otherwise unavailable without first suffering system compromise. As with traditional limited-scope honeypots, this new data-set is almost entirely comprised of threat activity, as there is no otherwise legitimate use of the redirection target.

A honeypot [46, 47] is a security resource whose value lies in being probed, attacked, or compromised. Honeypots take many forms, including Honeytokens [46], or single files which appear to be of value, service emulators such as those included in DECEPTION TOOLKIT(DTK)[1] and TINY HONEYPOT(THP)[2], host emulators such as HONEYD[3], all the way up to Honeynets [46], entire networks of monitored computers and network devices. Research honeypots are generally deployed alone[4], with no obvious affiliation to any real organization, while production, or defensive, honeypots are located in close proximity to real-world resources considered to be of value. When employed in a defensive role one hope is that intruders will find the honeypot attractive enough to focus their attention there, rather than ex-

[1] http://www.all.net
[2] http://www.alpinista.org/thp
[3] http://www.citi.umich.edu/u/provos/honeyd/
[4] The ISTS *Distributed Honeypot System* project is one exception.

pend additional effort penetrating a more heavily guarded system. More aggressive defenders will place a deception system in the direct path of a known and present attack [9] in the hopes of gleaning enough information from the attacker's actions to fuel a fine tuning of production defenses, or a response if the attacker is coming from within [28]. Ubiquitous redirection provides opportunities for deception that are not limited to a single point in the attack path.

Moran [32] describes three different deployment schemes, Minefield, Shield and Zoo, whereby the deception systems are placed amongst, in front of, or separate from production computers, respectively. The Shield approach relies upon a router or firewall using Network Address Translation (NAT) to redirect un-permitted service connection attempts, while the Minefield scheme litters the perceived network with deception systems. In both cases, it is assumed that legitimate users possess both the correct destination address and service port information to construct packets that will not only pass router and firewall scrutiny, but also be accepted at the destination. In the absence of redirection, any deviation from this discrete set of values would be itself sufficient to warrant failure at the host. With redirection, however, such anomalous behavior is met with a false view of the target environment.

Deception in Depth [17] is a diverse array of deceptive measures at multiple layers of data access. In order for a requester of any resource to gain access to it, legitimate or otherwise, accept/reject decisions are made at numerous points including encapsulation, transit, decapsulation, authentication and parsing. Data format, content and context validation are examined before the request or response can be passed to the next layer. Each of these may be appropriate redirection triggers if the proper conditions for successful passage aren't met. We propose that by offering responses to those messages that fail to successfully negotiate these borders, rather than rejecting or denying them, we increase the opportunity to observe attack behavior of interest, including attacks against as of yet undisclosed vulnerabilities. These observations can be used to derive network intrusion detection system signatures [25] or the analysis of unique malware content and behavior. For example, low-interaction honeypots were instrumental in our analysis of SQLSnake[5], providing our first glimpse of the worm in the wild.

Throughout this paper we quote publications by authors who chose to publish them anonymously or under a pseudonym. We will will use *slanting* to highlight such pseudonyms.

[5] http://www.net-security.org/virus_news.php?id=20

2 Multilayer access control responses

We can view each traversal of a communication path, client $\leftrightarrow$ server, as a single complex decision tree evaluating features and attributes of the data and its packaging in transit. There is only one leaf node which is considered to be a success, that which is classified as a legitimate request, and will be honored by the recipient application. Internal nodes, and their associated edges, are specified by many different entities such as the IETF, software vendors, local security policy. The small set of edges at most internal nodes includes one which is preferred and leads to the *success* leaf, whereas others lead to various error handling leaves. Once the data has been fully classified as leading to the success state, or to one of the set of error states, the system in possession of that data can then respond according to those same specifications. At each of these nodes, we may have an opportunity for redirection without significantly impacting legitimate *success* processing. We will examine several such opportunities next.

2.1 Transit or Local Network Redirection

The TCP/IP [40] suite of protocols provides for end-to-end communication of data through multilayer encapsulation according to strict specifications. Many conditions are defined in the RFCs [6] as fatal and should (or may or must) terminate the routing, switching, transport or application handling of data. The Transmission Control Protocol, TCP [36], requires that data segments contain port numbers, which identify the TCP's customer applications, and state tracking information. Similarly, the Internet Protocol, IP [35], uses a discrete feature set to discriminate traffic which includes addresses, TTL values, identification numbers, and service differentiation flags. Others, including application-layer protocols such as HTTP [13], are similarly particular.

With deception redirection, additional nodes are inserted in the policy flow as in Figure 1, subjecting each packet to additional scrutiny *only* after it has diverged from the SUCCESS path. Care should be taken to avoid imposing additional "bumps in the wire", but to leverage those that already exist for the purpose of message validation, guided by the existing architecture.

Table 1 illustrates common results with non-success classifications. For each of the errors, we indicate one possible deceptive response; these are by no means the only opportunities for redirection or response. We recognize that these deceptive responses may, if triggered by a non-malicious user error, cause confusion, loss of productivity, or otherwise degrade the perceived usefulness of the system. In the case of the HTTP error message, input validation routines may need to differentiate between three different

Table 1. Features, related error classifications and real and deceptive responses

Feature	Node	Classification	Response	Deception
Destination address	Gateway	Host unreachable	Discard, return ICMP type 3 code 1	Honeyd redirect
Destination address	Gateway or host	Prohibited	Discard, return ICMP type 3 code 10	routing loop
Protocol	Host	Not active	Discard, return ICMP type 3 code 2	silence
Time To Live	Gateway	Expired	Discard, return ICMP type 11 code 0	ICMP type 3 code 3
TCP flags, dest. port	Host	SYN, port closed	Discard, Acknowledge, send Reset	LaBrea tarpit[6]
HTTP Method	Application	Not allowed	Discard, send error 405	False web page

states: valid input, benign deviation from valid input, and everything else. Input is only measured against the second and third states if it has failed the initial test, after which the appropriate response is selected and returned.

THP uses LINUX NETFILTER [42] modules to effect ubiquitous redirection based primarily on connection state tracking and exceptions to locally defined policy. All new connections are evaluated against conventional stateful packet filtering firewall rules, but policy violations are further evaluated to determine an appropriate deception response as shown in Figure 1. A small set of service emulation Perl scripts provide sufficient response to convince many automata, as well as many unsophisticated humans, while the default responder, a thinly simulated root shell, catches everything else.

A 24-bit network with 1.544Mbps connectivity serving approximately 25,000 web hits daily was monitored with an early version of the IDABench intrusion analysis system providing full fidelity packet capture of all network activity. As with all deception systems, an intrusion detection system is an essential part of the analysis of captured attacker activity. Results were promising, with the network tenants reporting no interruption or performance degradation, while snort logged an average of 100,000 alerts daily. At such low traffic rates, no degradation in performance was noticeable. The vast majority of alerts, as expected, were worm related, but many targeted attacks fell upon services not offered on this, or most, networks, and on systems that were mere redirections. Raw data and statistical breakdowns are available upon request.

One such captured session was an attempt to use an inetd managed root shell listening on port 1524. This is a well known backdoor opened by a publicly available attack tool that exploits a vulnerability in Solaris Common Desktop Environment -

```
uname -a;ls -l /core /var/dt/tmp/DTSPCD.log;\
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:\
/sbin:/usr/ccs/bin:/usr/gnu/bin;
export PATH;
echo "BD PID(s): "`ps -fed|\
grep ' -s /tmp/x'|grep -v grep| \
  awk '\{print \$2\}'`
adduser
cat /etc/issue
adduser 0
passwd 0
clear
ksh
su -l
exit
```

[6] http://www.hackbusters.net/

The attacker realized that his attempt failed, but not until after revealing, by attempting to delete `DTSPCD.log`, what particular service he thought he was exploiting. To be certain, there were no hosts on this network vulnerable to the DTSPCD buffer overflow vulnerability[7], nor any Solaris hosts at all.

While THP is a lightweight redirector and service emulator, Honeyd [38] provides a complete framework for deception systems, from individual services to complete honeynets. Using ARP reply spoofing, Honeyd can dynamically fill any gaps in an existing network, or with the netfilter `DNAT` and `REDIRECT` targets, as used in THP, can intercept based on static and stateful inspection policy. This brief mention does little justice to the rich capabilities of Honeyd; refer to [38] for a more complete discussion.

2.2 Application Layer Redirection

Although there are countless userspace applications which accept input from various, potentially untrusted, sources, we will discuss one here that is commonly used for content-based redirection and is readily modified to a *redirect all which is not expressly permitted* policy.

From RFC2616

> The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server.

[7] http://www.cert.org/advisories/CA-2001-31.html

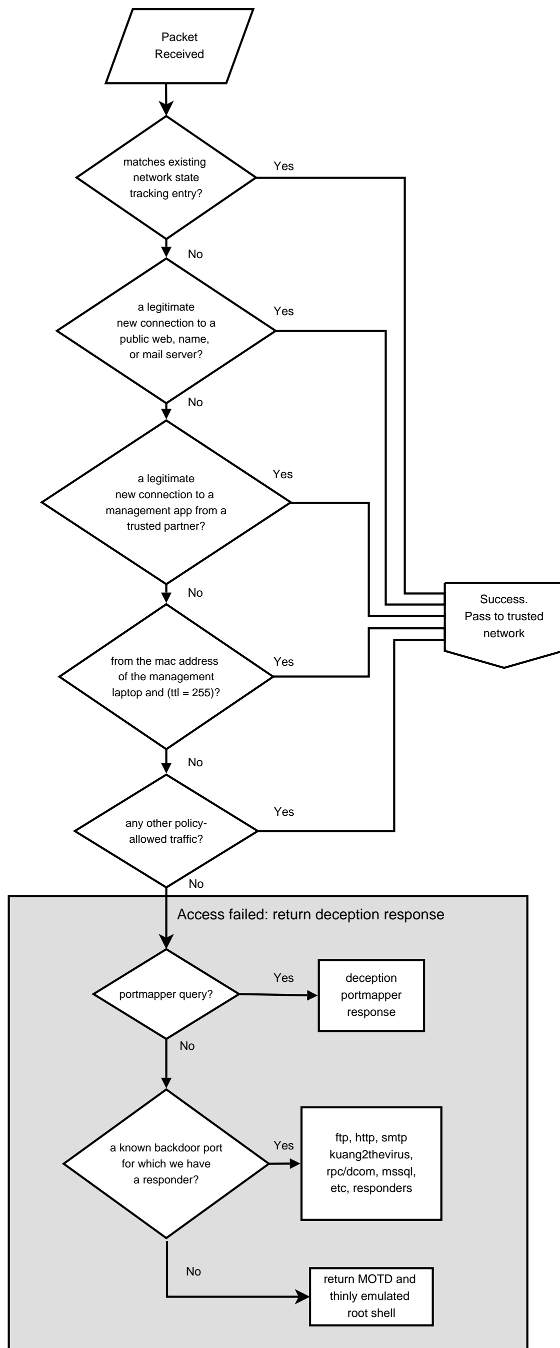

Figure 1. Tiny Honeypot using `netfilter` for redirection.

The Apache[8] HTTP server provides access control and redirection facilities commonly used for load balancing and content distribution. These capabilities can be used for transparently redirecting malformed requests to an alternate server. In the simplest case an Apache server can redirect all web requests for files that do not exist on the web server to a honeypot. If during a reconnaissance attempt the adversary searches for default.htm or viewcode.cgi they would be transparently redirected and receive a deceptive success response. One could enhance the illusion by mirroring the real web server and providing malformed CGI requests with contextually relevant, yet deceptive, responses. This concept is not unique to defensive deception - website administrators often use custom 404 handlers to increase the usability or profitability if a site[9] as do attackers for their own reasons[10].

As part of the defense infrastructure for a public web presence, we designed an enhanced httpd redirection capability using mostly native Apache server capabilities[11]. The Apache modules `mod_rewrite` and `mod_proxy` provide the web server with access control, web proxy and URI rewriting functionality needed to pull off this illusion. `mod_rewrite` uses a rule-based rewriting engine (based on a regular-expression parser) to rewrite requested URIs on the fly. The regular expression rule set processor, commonly used to manipulate URI layout, provides an effective means of access control. By representing all URIs, filenames and cgi inputs as regular expressions, `mod_rewrite` can be configured to launch handler code for URI manipulation and redirection. Let us consider the following `mod_rewrite` configuration syntax:

```
RewriteRule ^/(.*)    /var/www/$1

RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
...
```

The RewriteRule directive evaluates the regular expression, which in this case is all URI file requests. Without a preceding conditional requirement (RewriteCond), all requests will be rewritten to a simple fully-qualified file request. Next, the RewriteCond directives provide additional conditions that must be met for the *following* RewriteRule to be honored. The example statements check to see if the requested filename does not exist as a file (`!-f`) or as a directory (`!-d`) on the web server. If a request failed one of these checks, it would exit and not encounter the next

[8]http://www.apache.org

[9]http://www.byte-fbw.com/downloads/page-not-found/

[10]http://www.lurhq.com/ppc-hijack.html

[11]As external forces prevented this technique from being exercised, we have no measurements to offer here, but can speculate that any performance impact would be negligible

RewriteRule; the file exists and would be displayed to the user.

For our simple example we would like to transparently redirect all URL file requests that fail this access control chain to a honeypot. With `mod_proxy` installed on the web server, `mod_rewrite` will redirect and proxy the request to our honeypot. The following `mod_rewrite` rule, when placed immediately after the above RewriteCond statements, will provide such a proxy function to all URI requests that survived those conditions.

```
...
RewriteRule ^/var/www/(.*) \
        http://www2.yourdomain.com/$1
```

The honeypot at www2.yourdomain.com can be configured to provide some appropriate deceptive response such as a default web page, or it may hint at some known vulnerability. Before forwarding the honeypot's response to the requester, the proxy rewrites all of the packets to appear as if they are coming from the legitimate web server. This function helps maintain the illusion that the file exists on the server because upon inspection, all packet headers returned to the requester will be constructed by the legitimate web server. As the target web server is decrypting SSL/TLS data prior to HTTP parameter parsing, this technique is effective even after authentication and encryption negotiation.

Other candidates for similar application-header based evaluation and ubiquitous redirection include, but are not limited to:

- HTTP proxies
- SMTP servers
- authentication servers
- print servers
- SNMP servers

As mentioned earlier, any implementer of such three-tiered header validation must remain cognizant of user error within acceptable bounds. Profiling and/or user awareness is an essential part of this or any ubiquitous deception scheme.

2.3 Operating System

For quite some time, the black hat community has been exploring the potential of injecting code into operating system kernels to practice deception on legitimate users and administrators. This injected code intercepted and modified certain incoming and user data for deceptive purposes. The applications ranged from rootkits that merely concealed the presence of files, processes and connections to those that rendered the administrator's response actions ineffective, while creating the opposite impression. We are going to first give a brief survey of such techniques, then explore opportunities for their defensive application.

The lesson we draw from this study is that the instrumentation of the kernel for deception can be powerful, flexible, and yet relatively easy to accomplish. In particular, the necessary changes have good locality, and can be orthogonally introduced at many different points throughout the Linux and BSD kernels.

Intercepting system calls. Many kernel level rootkits target the so-called system call table, an array of pointers to functions implementing the functionality of privileged kernel mode operations, which is a common OS design feature[12]. This technique is known as *system call hooking* or *system call hijacking*. In its simplest form, it involves replacing a pointer in the system call table with the address of attacker-injected code that modifies the arguments of the system call, its resulting data or other kernel data structures. The injected code typically calls the actual function implementing the system call before or after making such adjustments. This technique was publicly described in 1997 by Runar Jensen on Bugtraq and by *halflife* in Phrack 50, and was probably known well before its publication.

An early tutorial on Linux loadable kernel module (LKM) rootkits [37] described hooking the `getdents` system call to conceal presence of files and processes[13] from user tools, the `open`, `write` and `read` system calls to completely conceal or fake the contents of a file, or of certain parts of a file. Furthermore, it described redirecting execution of files (by hooking `execve`) to introduce trojans and escape checksum-based file integrity checks, and the uses of the hooked `socket` call for operating a backdoor using specially crafted packets. Also described was hooking the module-related system calls to conceal the presence of rogue LKMs on the system.

The use of system call wrappers for protective purposes (hardening of systems and observation and confinement of processes at high risk of being attacked) has been well explored in research publications ([19, 16, 21]). Tools for logging systems calls and their arguments by means of hijacking them have since become available to the average system administrator through packages such as Snare[14] and Syscalltrack[15]. Since then, Linux Security Modules (LSM) has been proposed [48] and endorsed as the framework for

[12] In the Linux kernel this table is known as the `sys_call_table`, in Windows as the *system service dispatch table*. In this paper we limit the discussion of kernel deception techniques to Linux. Windows kernel rootkits use similar methods.

[13] When information about the kernel process table is obtained via the `/proc` filesystem, as is the case with `ps` and related utilities.

[14] http://www.intersectalliance.com/projects/Snare/

[15] http://syscalltrack.sourceforge.net

extending security functionality of the kernel. Other places, such as library interfaces, where code interposition could serve security purposes were also explored ([22, 27]).

The tutorial [37] was likely the first such publication to discuss the defensive deceptive uses of the system call hooking technique by the administrators, essentially by "getting there first", ahead of the attacker. The discussion included concealing the administrator's modules that provided hardening, extra logging and instrumentation, in particular, making certain processes invisible and untraceable. Further, it discussed examples of thwarting and logging the attacker's actions without his knowledge, such as redirecting reads and writes to sensitive files.

Kernel data structures manipulation Kernel level rootkits use various deception techniques to present the users with false or misleading information about the system's status. Unlike their older user space predecessors, these rootkits soon abandoned the naive substring matching approach[16] in favor of direct manipulation of the kernel data structures, in particular the process table. The challenge was twofold: the concealed records had to be hidden from the results of the kernel's standard traversal routines when presenting information to the user (e.g. the `/proc` filesystem operations) but at the same time appear normal to the vital kernel mechanisms such as scheduling and signal delivery (so that, e.g., hidden processes would get scheduled normally).

A number of rootkits used the `Knark` strategy of setting an unused bit in the process' `task_struct` flags to indicate that a process was hidden. Accordingly, the `fork` system call was modified to set this flag on the children of a concealed process right at their creation time. The entries of such processes obtained by the `/proc` filesystem pseudo-directory traversal implementation were erased from the output buffer, and the `kill` syscall's signal handling for them was modified. This approach did not interfere with the process table linked list and hash table structures (both implemented via pointer fields of the per-process `task_struct` record), and tools like `KSTAT` were written to follow these structures by interpreting the kernel memory directly via `/dev/kmem`, bypassing the possibly trojaned `/proc` traversal routines. However, these were soon followed by countermeasures that presented a false view of the process table to the examining process on the compromised machine, by producing a fresh doctored copy of the actual process table used by the scheduler for the sole benefit (and deception) of the examining process. Furthermore, an exploit called `Phantasmagoria` claimed to be able to patch the scheduler in the generic 2.4 kernel in such a way that the concealed processes were linked into the process table's linked list structure just in time for scheduling, and unlinked immediately thereafter, thus eliminating the need for modifying the output of the standard traversal routines.

Other kernel structures (e.g., the list of all loaded modules) were manipulated in a similar way, with the only difference that violating their integrity was often less critical to the normal OS operation.

Further in and further down. Countermeasures such as tools for verifying the integrity of the `sys_call_table` (e.g., `KSTAT`) and concealing its address in the kernel memory space soon lead to further explorations of other places where kernel behavior could be conveniently changed by inserting special case code, without modifying the system calls at all.

A number of convenient places for injection below the syscall level (which can be thought as the outward layer of the kernel, bordering on the user space) were soon identified. The Phrack 55 article [24] examined inserting hooks into the Linux TCP/IP stack right above the network device driver, causing each packet's `sk_buff` to be examined and its processing changed for special cases. In particular, the packet contents could be redirected, mangled and passed to the following stages of the stack, quietly dropped, or "stolen" by a special processing module such as a backdoor, and never passed to the rest of the stack.

The Virtual FS abstraction layer was shown to be another convenient place for placing deception and concealment code. The hijacking of `/proc` filesystem file operations dispatch table entries, with the purpose of process and connection hiding was described in Phrack 58 ([34, 29], and in Phrack 59 ([30]), an advanced version of execution redirection via interposing the wrapper code between the binary format handler for ELF executable files and the `execve` system call, restored immediately after the hooking was achieved. These and other VFS hacking techniques have been ported to the 2.6 kernels with a new version of the `adore` rootkit, `adore-ng`.

While the aforementioned hooking methods took advantage of the same design decisions that made Linux easily portable and extensible, other extremely architecture-specific ways of introducing code were explored, such as replacing interrupt handlers in the x86 Interrupt Descriptor Table (in particular, the floating-point exception handler and page-fault handler, [23, 7]) thus modifying the arguments of system calls even before they are dispatched through the Linux call gate handler.

At about the same time, more efficient TTY sniffing techniques based on hooking the internals[17] of the Linux

[16] Later user space rootkits, such as `lrk5`, stored the substrings that had to be suppressed in the output of standard tools in various configuration files hidden by the same tools.

[17] Several places for placing the sniffing code were considered, from the keyboard interrupt handler and scancode converter to the TTY buffer processing function, all well below the system call level considered by *halflife*

TTY driver were published (Phrack 59, [39]). The article [39] is interesting for its acknowledgment of the role played by TTY sniffers in honeypots[18], as well as for black hat purposes.

2.4 The defensive perspective

The above examples of code injected into the OS kernel for handling *special cases* follow the same essential scheme. The hooked kernel system call, function or handler is replaced with a *wrapper*, which applies a *filter* implementing the definition of the special case to the arguments or data passed to the hijacked procedure, and either passes them on to the original procedure (possibly doctoring its output before returning control to the caller of the wrapped fragment), or takes another non-standard action, like initiating a MOSDEF "remote stack swapping service"[19]. The filter may include complicated logic, and the non-standard side-effects may involve sophisticated lookups of context, taking advantage of the kernel's monolithic architecture and ready availability of data structures throughout the kernel space.

Our proposed way of applying these lessons to the practice of ubiquitous redirection is as follows. In keeping with the honeypot inverted view of the world, *let us consider the expected benign traffic or user data as a special case to be passed by the wrapper's filter for normal kernel processing, and redirect the rest into honeypotting handlers*. This presents the problems of defining (1) the normal traffic or data and (2) the honeypot's response to anomalies.

2.4.1 Defining the normal

The expected "normal" behavior of a system can be prescribed by a rule-based policy [44] (such as, in the order of increasing sophistication, the LIDS lists of allowed parameters, Grsecurity [20] ACLs or the fine grained BlueBox [8] approach), or learned from a period of guaranteed normal use, with techniques similar to those used in [14, 45, 43, 18].

In each case, a careful choice of features for data received by the hijacked kernel routine will be needed to allow for the smallest possible lists or ranges of normal values. The experience of the system call based approaches suggests that the amount of kernel memory required is manageable (on average 8K per process).

Both policy-based and learning approaches to defining normal behavior have been thoroughly explored in non-deception contexts, most notably for their applications in intrusion detection systems (e.g., surveys [1, 5, 33, 4, 31], and various research systems and prototypes).

The example network in Section 2.1 was ideal for our study as legitimate user behavior was well profiled, there was a limited set of services available to the general public, and remote management was according to strict policy guidelines. As network access policy was primarily enforced by the border firewall, as was the redirection to deceptive response scripts, any adjustments to that policy were immediately reflected by the honeypot coverage.

2.4.2 Response to anomalies

For hooks in the honeypot's TCP/IP stack, while the target process has not yet acquired enough state, the kernel wrapper can generate, possibly on a private network interface, a request, directed to its gateway router, to redirect further traffic from the source that triggered the alert to a different machine where it could be contained and handled. The router honoring such requests can be implemented based on `iptables`: a daemon listening for redirection requests on the private network concealed from the attacker, will insert a packet mangling (rewriting) rule into the corresponding redirection chain. This scheme can also be applied to alerts raised by filters positioned elsewhere in the kernel, if the amount of state accumulated by the process in the context of which the alert was triggered allows the redirection to succeed without breaking the attacker's session. As we have remarked earlier, performing the necessary context lookups in the Linux kernel does not present a problem.

For processes whose existing context cannot be easily replicated on a physically separate system, steps can be taken to terminate it, or, better, to isolate it on the same system and activate extra mechanisms for logging or tracing. Response options can include lowering the process capabilities, redirecting the process to a separate copy-on-write filesystem, and limiting its view of the environment, or even confining it inside a virtual environment such as a UML or a `vserver` context.

Deception as a response to known attack vectors. Popular attack techniques, such as overflowing the program stack with executable shellcode, using format string and heap allocation vulnerabilities and `ptrace`-based manipulations of process memory image often inspire responses in the form of custom modifications of the defended system that include *run-time condition checks* that any attack of the targeted class would trigger (e.g., [11, 10, 41], `Kfence`, *Solar Designer*'s patches, various ad-hoc anti-tracing patches). A common feature of these responses is that they define an abnormal run-time condition, incorporate checks for it into a production system, and immediately

in his seminal Phrack 50 article [20]

[18] Jeff Dike has implemented this fully in his honeypot extensions to User Mode Linux – http://user-mode-linux.sourceforge.net/tty_logging.html

[19] http://www.immunitysec.com/MOSDEF/

[20] http://www.grsecurity.com

raise an error, preventing its possibly successful exploitation.

However, if, upon early detection of hostile intentions, the attacker can be reasonably contained, leading him to believe that his attack has succeeded and observing his further actions may provide valuable intelligence about his identity and intentions. In particular, the aforementioned run-time sanity checks, instead of terminating the session that they have revealed as a hostile one, could provide a deceptive response, while designating the attacked process as "tainted". This designation could happen either directly in the kernel space, if the corresponding sanity check happens there, or through a special system call, if the sanity check is performed in user space (such as with `StackGuard` and `FormatGuard`). In principle, a process could be marked at tainted at any place where interposed code performs a security condition check that is likely to suggest hostile data (e.g., in the loader, dynamic linker, a protocol parser etc).

Such tainted processes will need to be treated specially by the kernel, in particular, their view of the filesystem should be virtualized, so that their writes (and, possibly, their reads of sensitive files) are redirected, their capabilities reduced and the corresponding capability-related system calls instrumented to provide deceptive responses, and their view of the system environment limited (e.g., to exclude the observing and administrative processes). Additionally, the scheduler and the network stack could be modified to limit the consumption of resources by tainted processes.

While clearly non-trivial, such instrumentation of the system for deceptive responses appears possible, based on the experience of projects such as `vserver`[21] and `SILK` a.k.a. `plkmod`[22]. The former introduces limiting contexts for processes, restricting their view of the system environment and the ability to communicate with other processes, and virtualizes the file system, while the latter replaces the socket API and modifies the scheduler algorithm. This existing instrumentation of the kernel, based on an extended version of POSIX capabilities, could be coupled with a framework for providing deceptive responses where the restrictions enforced by a tainted context would result in denying the operation. In particular, the deception framework, unlike these projects, will provide the inquisitive attacker with fake listing of the actually unavailable devices and `/proc` entries (possibly using snapshots of the real filesystem).

Performance considerations Feasibility of ubiquitous checks for abnormal conditions (based either on policy rules or statistical models) depends on the overall amount of overhead they would impose on the system. Existing studies of performance loss due to system call wrapping for auditing [2] and statistical model-based detection of anomalous system call arguments [26], and due to `StackGuard`'s run-time canary checks[23], well as performance measurements of the previously cited hardened systems suggest that the average performance loss can be acceptable.

[21] http://www.linux-vserver.org

[22] http://www.cs.princeton.edu/˜acb/plkmod/

[23] http://immunix.org/StackGuard/performance.html

3 Conclusions

In this paper we considered various kinds of practical deception techniques based on redirection. The blackhat community has convincingly demonstrated the power of kernel level redirection for malicious deception of legitimate users. Understanding the threat of discovery, rootkit developers implement survival-motivated rule-based deception "policies" through process hiding, file hiding, module list filtering, etc.

On opposite end of the spectrum, we have witnessed good results from limited, rule-based policy redirection used for defensive purposes, in particular in honeypots. However, many opportunities are being missed by initiating redirection and offering deceptive responses only on certain discrete sets of anomalous conditions. Instead, we suggest implementing a default policy of deception at multiple decision nodes throughout the host and network context. In other words, we propose to take the approach *that which is not specifically allowed is subject to deceptive response*. Recognizing the cost of this approach in performance and administration overhead, we posit that it is within the essential philosophy of honeypot technology, and may result in significant gains in timely intelligence gathered and the corresponding increase in defensive capability.

For the proposed defense paradigm to gain ground on production systems, further research is needed in a number of areas. These include performance costs of system instrumentation for redirection and deception, possible negative effects of benign mistakes on legitimate users' experience and methods of coping with them, as well as formal verification of misuse detection and confinement models for attack sessions and their non-interference with the specifications of intended system functionality.

4 Acknowledgments

We acknowledge the support of the United States Department of Homeland Security - Office of Domestic Preparedness, Institute for security Technology Studies at Dartmouth College, and the members of the Honeynet Project for their support. Additional contributors include Jeff Dike, Robert Gray, Chris Carella and Trey Gannon of ISTS.

Supported under Award number 2000-DT-CX-K001 from the U.S. Department of Homeland Security, Science and Technology Directorate. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security or the Science and Technology Directorate.

References

[1] J. Allen, A. Christie, W. Fithenand, J. McHugh, J. Pickel, and E. Stoner. State of the practice of intrusion detection technologies. Technical Report CMU/SEI-99-TR-028, Software Engineering Institute, Carnegie Mellon University, Jan. 2000.

[2] A. Apvrille. Evaluation of a few security audit tools for linux. Technical report, Open Systems Lab, Ericsson Research, Canada, Apr. 2003.

[3] O. Arkin. Network Scanning Techniques, November 1999. PubliCom Communications Solutions.

[4] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Chalmers Univ., Mar. 2000.

[5] R. Bace and P. Mell. Intrusion detection systems. Technical Report SP 800-31, National Institute of Standards and Technology, Aug. 2001.

[6] S. Bradner. RFC 2026: The internet standards process - revision 3, Oct. 1996.

[7] buffer. Hijacking linux page fault handler exception table. *Phrack*, 61–7, Aug. 2003. Avaliable from http://www.phrack.com/.

[8] S. N. Chari and P.-C. Cheng. Bluebox: A policy-driven, host-based intrusion detection system. In *Proceedings of the Network and Distributed System Security Symposium*, 2002.

[9] W. R. Cheswick. An evening with Berferd, in which a cracker is lured, endured, and studied. In *Proceedings of the Winter* USENIX *Conference*. The USENIX Association, Enero 1992.

[10] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Aug. 2001.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[12] M. de Vivo, E. Carrasco, G. Isern, and G. O. de Vivo. A review of port scanning techniques. *SIGCOMM Comput. Commun. Rev.*, 29(2):41–48, 1999.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol – http/1.1, June 1999.

[14] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security & Privacy*, 1996.

[15] Fyodor. Remote OS detection via TCP/IP Stack Fingerprinting, October 1998. http://www.insecure.org/nmap/nmap-fingerprinting-article.html.

[16] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[17] S. Gerwehr and R. H. Anderson. Employing deception in infosec. In *Proceedings of the* 3rd *IEEE Information Survivability Workshop (ISW 2000)*, Boston, MA, October 2000.

[18] A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. pages 51–62, 1999.

[19] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[20] halflife. Linux tty hijacking. *Phrack*, 50–5, Apr. 1997. Avaliable from http://www.phrack.com/.

[21] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed Systems Security Symposium*, Feb. 2000.

[22] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Symposium on Operating Systems Principles*, pages 80–93, 1993.

[23] kad. Handling the interrupt descriptor table. *Phrack*, 59–4, July 2002. Avaliable from http://www.phrack.com/.

[24] kossak and lifeline. Building into the linux network layer. *Phrack*, 55–12, Sept. 1999. Avaliable from http://www.phrack.com/.

[25] C. Kreibich and J. Crowcroft. Automated nids signature creation using honeypots. ACM SIGCOMM 2003 - Data Communications Festival, August 2003. poster.

[26] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *Proceedings of the 2003 European Symposium on Research in Computer Security*, Gjovik, Norway, October 2003.

[27] B. A. Kuperman and E. Spafford. Generation of Application Level Data via Library Interposition. Technical Report CERIAS TR 1999-11, COAST Laboratory, West Lafayette, Indiana 47907-1398, Oct. 1999.

[28] J. Levine, R. LaBella, H. Owen, D. Contis, and B. Culver. The use of honeynets to detect exploited systems across large enterprise networks. In *Proceedings of the 2003 IEEE Workshop on Information Assurance*. IEEE, June 2003.

[29] mayhem. Linux x86 kernel function hooking emulation. *Phrack*, 58–8, Dec. 2001. Avaliable from http://www.phrack.com/.

[30] mayhem. Advances in kernel hacking ii. *Phrack*, 59–5, July 2002. Avaliable from http://www.phrack.com/.

[31] L. M'e and C. Michel. Intrusion detection: A bibliography. Technical Report SSIR-2001-01, Sup'elec, Rennes, France, September 2001.

[32] D. B. Moran. Trapping and tracking hackers: Collective security for survival in the internet age. In *Proceedings of the* 3rd *IEEE Information Survivability Workshop (ISW 2000)*, Boston, MA, October 2000.

[33] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *IEEE Network*, pages 26–41, May–June 1994.

[34] palmers. Advances in kernel hacking. *Phrack*, 58–6, Dec. 2001. Avaliable from http://www.phrack.com/.

[35] J. Postel. RFC 791: Internet Protocol, Sept. 1981.

[36] J. Postel. RFC 793: Transmission control protocol, Sept. 1981.

[37] pragmatic/THC. (nearly) complete linux loadable kernel modules, Mar. 1999.

[38] N. Provos. A virtual honeypot framework. Accepted paper, USENIX Security 2004, October 2003.

[39] rd. Writing linux kernel keylogger. *Phrack*, 59–14, July 2002. Avaliable from http://www.phrack.com/.

[40] J. Reynolds and S. Ginoza. STD 1: Internet Official Protocol Standards, Nov. 2003.

[41] T. J. Robbins. libformat. Available at http://www.wiretapped.net/ fyre/software/libformat.html.

[42] R. Russell. Linux 2.4 Packet Filtering HOWTO v. 1.2. http://netfilter.samba.org/documentation/HOWTO/packet-filtering-HOWTO.txt, 2002.

[43] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. pages 29–40.

[44] R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 338–347, 1998.

[45] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, Aug. 2000.

[46] L. Spitzner. The Honeynet Project: Trapping the hackers. *IEEE Security & Privacy*, 1(2):15–23, Mar./Apr. 2003.

[47] C. Stoll. *The Cuckoo's Egg - Tracking a Spy through the Maze of Computer Espionage*. Doubleday, New York, NY, USA, 1989.

[48] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002.