Internet attacks monitoring with dynamic connection redirection mechanisms

Éric Alata, Ion Alberdi, Vincent Nicomette, Philippe Owezarski and Mohamed Kaâniche
LAAS-CNRS, University of Toulouse, France
Email: {ealata,ialberdi,nicomett,owe,kaaniche}@laas.fr

Abstract

High-interaction honeypots are interesting as they help understand how attacks unfold on a compromised machine. However, observations are generally limited to the operations performed by the attackers on the honeypot itself. Outgoing malicious activities carried out from the honeypot towards remote machines on the Internet are generally disallowed for legal liability reasons. It is particularly instructive, however, to observe activities initiated from the honeypot in order to monitor attacker behavior across different, possibly compromised remote machines. This paper proposes to this end a dynamic redirection mechanism of connections initiated from the honeypot. This mechanism gives the attacker the illusion of being actually connected to a remote machine whereas he is redirected to another local honeypot. The originality of the proposed redirection mechanism lies in its dynamic aspect: the redirections are made automatically on the fly. This mechanism has been implemented and tested on a Linux kernel. This paper presents the design and the implementation of this mechanism.

1 Introduction

Nowadays, honeypots have become a very common tool to monitor and analyze malicious activities on the Internet. The various implementations that have been proposed so far differ by the level of interaction and the possibilities offered to the attackers. The higher the interaction level is (i.e., the more faithfully a honeypot can simulate a 'real' environment), the more possibilities are offered by the honeypot to observe an entire attack.

For example, a honeypot such as honeyd [1] offers limited possibilities to the attackers, as generally only a subset of services are partially emulated. The emulated services cannot be used to compromise the honeypot or to attack other machines on the Internet. Nevertheless, even when the interaction level is low, the information recorded by the honeypot provides useful insights about the services that are frequently targeted by the attackers. In particular, the development of more sophisticated honeypots aimed at the detailed analysis of attacker behavior could be focused on these services.

There are also other honeypot implementations that offer higher levels of interactions. These run real operating system services and application software, which makes them riskier than low interaction honeypots. It is noteworthy that these two types of honeypots (generally referred to as *low-* and *high-interaction* honeypots) are complementary. In addition, another category called hybrid honeypots combining the advantages of low and high interaction honeypots has been proposed in [2, 3].

With the current honeypot techniques mentioned above, it is impermissible for legal liability reasons to bounce from the honeypot machine to compromise or run attacks against third party machines. This limitation precludes the possibility of observing complete attack scenarios unfold on the Internet. Moreover, it might also have an impact on attacker behavior: Attackers might stop their attack and decide to never use again the honeypot for future malicious activities. To address this limitation, some implementations limit the number of outgoing connections from the honeypot through the use of "rate limiting" mechanisms. Although this solution allows more information about the attack process to be captured, it remains insufficient and does not address the liability concerns.

In the context of our research work dealing with the observation and analysis of attack processes, we have developed a VMWare-based high-interaction honeypot [4, 5] and deployed it on the Internet. The data collected from the experiments allowed us to obtain interesting information about successful intrusions and attack attempts targeting SSH services. For these experiments, outgoing connections were limited to avoid potential attacks against third party machines. Only outgoing connections towards port 80 were allowed, under strict control. However, during these experiments, several connection attempts from the honeypot to the Internet have been observed.

Clearly, it is important to investigate new solutions which allow such connections without harming third-party machines. The mechanism presented in this paper is aimed at fulfilling this objective. The proposed technique consists of redirecting outgoing connections to a local machine, while making the attackers believe that they are able to bounce outside the honeypot.

The idea of connection redirection in the context of honeypots has been investigated in various studies, e.g. [6, 7, 3]. For example, the Collapse architecture presented in [6] integrates mechanism to forward traffic malicious targeting the end system of a local network to the Collapsor Center. However, their mechanism is not designed to address the problem of redirection of outgoing connections addressed in this paper. The study presented in [7] follows a similar objective to ours. However, the proposed solution is less generic. In fact, [7] proposes a way to redirect identified malicious traffic to a honeypot by hijacking TCP sessions. Their implementation requires the modification of the TCP/IP stack of the honeypot. Our implementation is aimed to be totally compliant with the standard TCP/IP interface and furthermore to be more generic by including not only TCP but also UDP, ICMP, etc. We can also mention the hybrid honeypot architecture proposed by [3]. The redirection mechanism presented in their study is used to support the analysis of malware and their propagation. However, little detail are provided about how the mechanism is implemented.

This paper is structured into six sections. Section 2 presents the main design principles and architecture of the proposed redirection mechanism implemented in the Linux kernel. Section 3 describes the implementation of the mechanism. Section 4 presents some experimental results illustrating the performance of the mechanism. In particular, we analyze the overhead incurred by this mechanism considering TCP sessions establishment times as an example. Section 5 outlines two experimental studies aimed at validating the proposed mechanism. The last section presents some conclusions and future work.

2 Principles

We have designed and implemented a selective mechanism which allows outgoing Internet connections from the honeypot to be automatically and dynamically redirected. The goal is to make the attacker believe he can connect from the honeypot to hosts on the Internet, whereas in reality, the connections are simply redirected towards another honeypot. Hence, the originality of our method is the dynamic nature of this redirection mechanism, as discussed in 1.

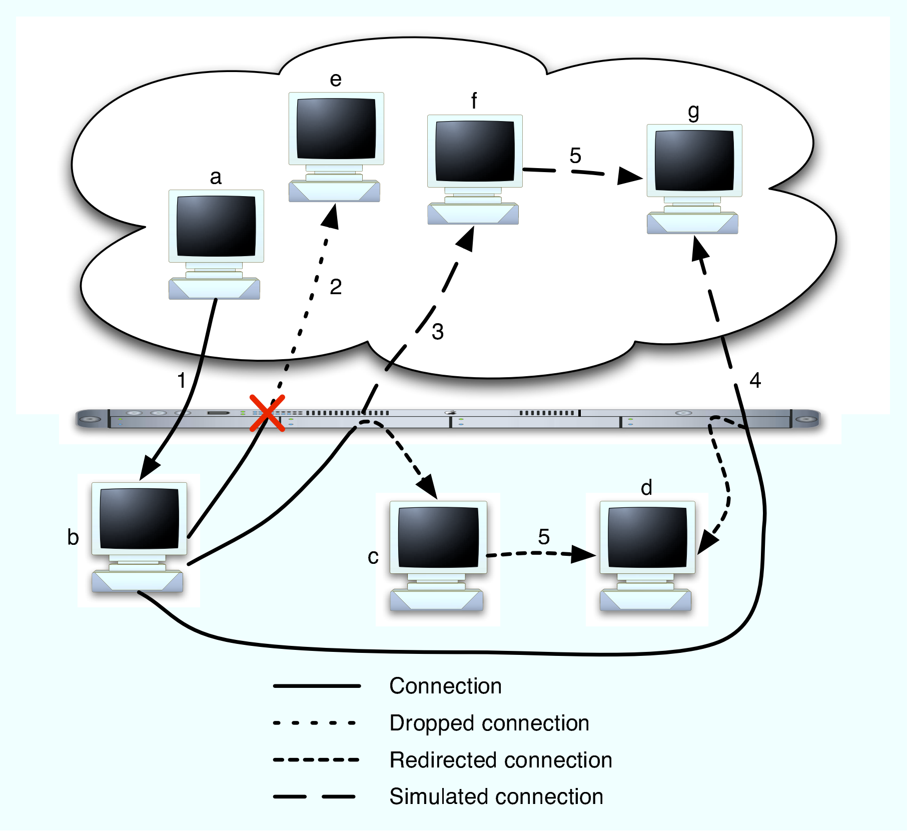

Figure 1: Example of redirection

Let us take the example presented in Figure 1. In this example, b, c and d are honeypots and a, e, f and g are machines on the Internet. The honeypots b, c and d are virtual machines running on a real host whose kernel has been modified to implement our redirection mechanism. An attacker from Internet host a breaks into honeypot b (connection 1). From this honeypot, the attacker then tries to break into Internet host e thanks to connection 2. This connection is blocked by our mechanism.

The attacker then tries another connection 3 towards Internet host f. This connection is accepted and automatically redirected towards honeypot c. The attacker is under the illusion that his connection to f has succeeded, whereas it has merely been redirected to another honeypot. The attacker tries to establish another connection 4 towards Internet host g. Similar to connection 3, this connection is accepted and automatically redirected towards honeypot d. The attacker finally initiates another connection (5) to Internet host g from host f (in reality, from host c). This connection is also accepted and is redirected towards honeypot d.

This mechanism is interesting because it allows attacker activity on different hosts to be observed. In general, a honeypot allows the activity of the attacker to be observed at only one side of the connection. The other connection end is the machine that interacts with the honeypot. For all redirected connections, we can observe an attacker on *both* connection ends. In the previous example, the attacker establishes a connection between host b and host c and it is possible to observe both hosts b and c.

On the other hand, it is possible for a clever attacker to see through the hoax. For example, in Figure 1, suppose the attacker already controls the machines a, e and f. He can then check, after establishing connection 3, if the machine he is connected to really is machine f. This limitation does exist; however we believe that many attackers will not systematically do such checks, in particular if the attack is carried out by non-sophisticated automatic scripts. Just as `honeyd` provides us some useful albeit limited information, more attack information will be gleaned from systems that implement our redirection mechanism than those that do not.

Of course, the redirection mechanism must be as reliable as possible so that we can collect data which faithfully captures attacker behavior. Thus, the implementation of this mechanism must have the following properties:

- it must be adaptable according to the needs of the administrator.
- it must be as elusive as possible in order to allay suspicions of the attacker
- it must not increase in a visible way the latency of the communications.

The following section presents an implementation of our redirection mechanism aimed at meeting these requirements.

3 Implementation

The dynamic redirection mechanism has been implemented in the Gnu/Linux operating system. Nevertheless, it has been designed in such a way that it can easily be adapted to other systems.

As illustrated in Figure 2, the mechanism includes three components:

- the `redirection module` (inside the kernel) extracts packets.
- the `dialog_handler` decides whether the extracted packets must be redirected or not.
- the `dialog_tracker` is a link between the `redirection module` and the `dialog_handler`.

Our redirection mechanism implementation is thus situated both in the kernel and user spaces. It would have been possible to implement it exclusively in user or kernel space. However, the first solution introduces an unacceptable latency which can be used for detection purposed by an attacker (during scan attacks, for example). The second solution is less flexible than our approach and does not allow the use of tools like databases for example (see later on), even if it is probably better from a performance point of view.

In the following subsections, we present each of the three components.

3.1 The redirection module

The redirection module must extract packets in such a way that they can be redirected or blocked. To do so, our module interacts with the `netfilter` component of the kernel [8]. This component is a firewall which includes five chains. Each chain is used to intercept and possibly modify packets at different stages on their way through the IP stack (see Figure 3):

- `INPUT`: chain processing packets addressed to the firewall

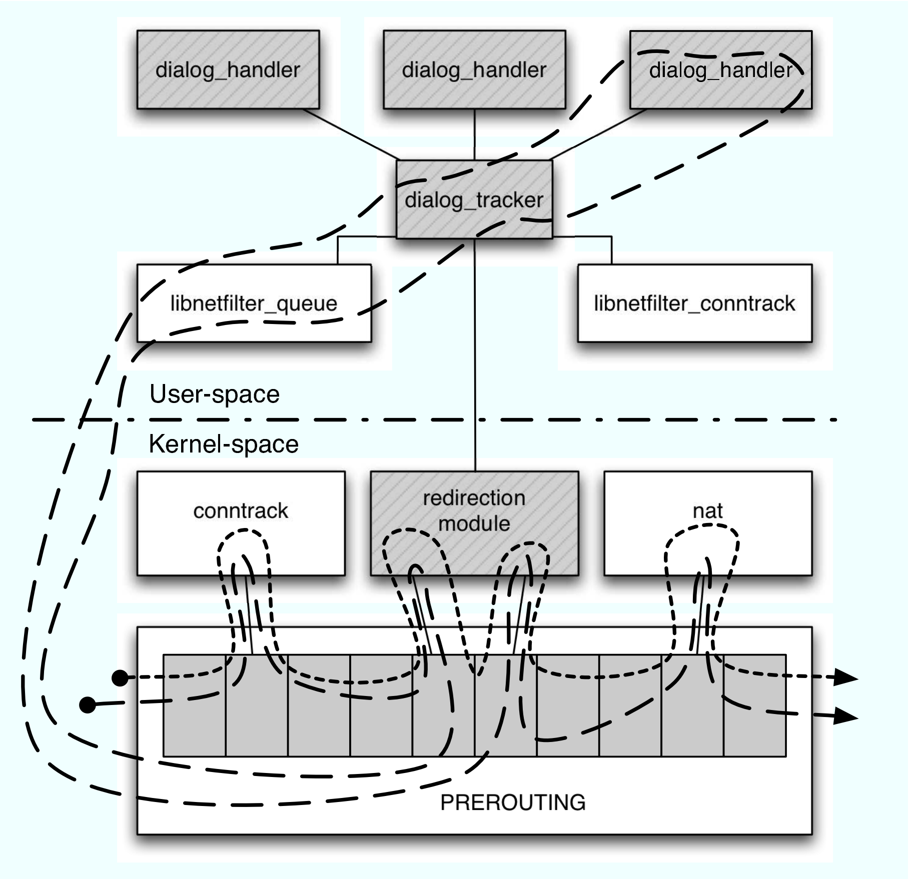

Figure 2: Redirection architecture

- `OUTPUT`: chain processing packets emitted by the firewall
- `FORWARD`: chain processing packets going through the firewall
- `PREROUTING`: chain processing, before they are routed, packets going through the firewall
- `POSTROUTING`: chain processing, after they are routed, packets going through the firewall

A set of particular functions named `hooks` is associated with each chain. Each hook has a specific role and processes the packets that pass through the chain. For example, the hook `conntrack` updates the state-machine of the connection corresponding to the processed packet. The hooks associated with a chain are ordered by priority. Hence, in a chain, the hook with the highest priority processes packets before all others and the hook with the lowest priority processes packets after all others. Adding kooks by inserting a module in the kernel is a simple way to enrich `netfilter`.

In our implementation, we benefit from the already existing hooks of `netfilter` (see Figure 2). More precisely, we use the `DNAT` hook's ability to modify packet destination addresses. The priority of this hook is `NF_IP_PRI_NAT_DST` = -100 in the `PREROUTING` chain. We also benefit from the `conntrack` hook (which associates a state-machine per pending connection) because it automatically identifies the first packet of a new connection.

In order to implement our redirection mechanism, we have developed two hooks and inserted them between the hook `conntrack` and the hook `DNAT` (which changes the destination address of a packet) in the `PREROUTING` chain. Our first hook is in charge of extracting packets and sending them to the `dialog_tracker` in user space, in order to decide whether they have to be redirected or not, whereas our second hook is in charge of tagging the corresponding connections as "redirected" if the decision to redirect them has been taken. We do not systematically redirect all the connections initiated from the honeypot to the Internet. Most of them are blocked and only a few of them are redirected.

In fact, thanks to the hook `conntrack` of netfilter, the redirection of a whole connection only requires the redirection of the first packet of this connection (the other packets are automatically processed like the first one).

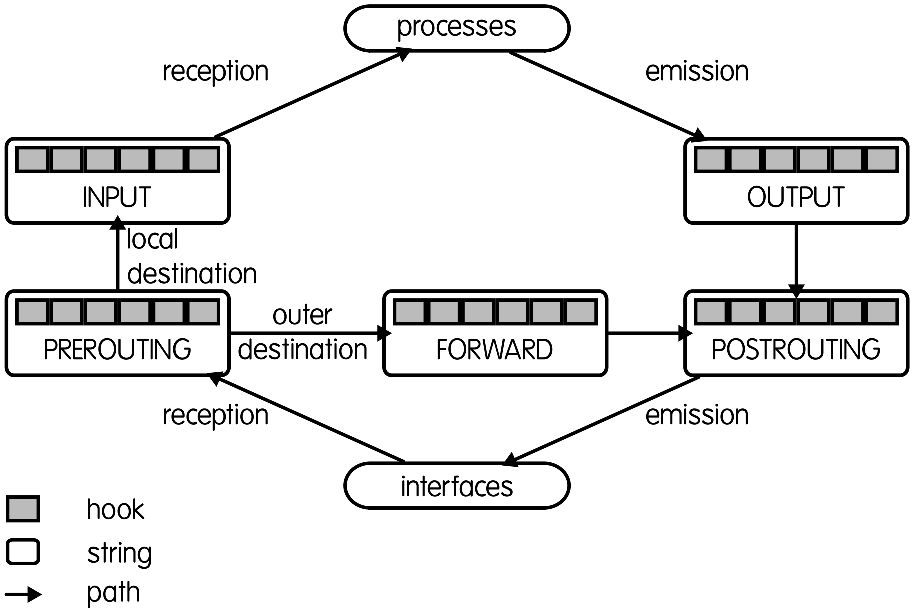

Figure 3: netfilter hooks and packet processing

Thus, for each connection, our first hook extracts only the first packet and sends it to the `dialog_tracker`[1]. Then, the `dialog_tracker` forwards the packet to the `dialog_handler`, which decides if this packet has to be redirected or blocked. This decision may be evaluated according to different rules such as for example "a connection upon 10 or 100 is redirected, the others are blocked"[2].

When the decision is made, the `dialog_tracker` informs the kernel module through a `netlink` socket that the corresponding connection has to be tagged as "redirected" and the packet is re-injected into the next hook of the chain, which is our second hook. This second hook is simply in charge of tagging the corresponding connection as "redirected". The packet is then re-injected in the list of hooks of the corresponding chain. One of them is the hook `DNAT` which indeed redirects packets of connections tagged as "redirected", by modifying the destination address of the packet (the destination address is changed to the address of one of our honeypots).

In order for this modification to be done correctly, the `DNAT` hook must be configured. This configuration is made through a rule via the `iptables` command. For example, the following rule configures this hook to redirect all packets of a connection tagged as "redirected" with the tag `0x03FEA8C0` to the machine 192.168.254.3:

```
iptables -t nat -A PREROUTING \
    -m connmark --mark 0x03FEA8C0 -j \
    DNAT --to-destination 192.168.254.3
```

In order to invoke the `dialog_tracker` from the kernel space, we used the `libnetfilter_queue` library. Indeed, this library already implements:

- the "theft" of packets to send to the user space,
- the management of the list of the stolen packets,
- the replication of the content of the `nf_reinject` function enabling to reinject a packet previously stolen, after a decision is taken for it.

Thanks to the `libnetfilter_queue` library, we can minimize our efforts. When a decision is made for a packet, the latter automatically goes on its way, from the hook whose priority is just higher than that of our first hook.

[1] The first packet can be easily identified because the hook `conntrack` which is just before our first hook tags the first packet of each connection as `NEW`.

[2] Definition of this decision process is part of future work.

3.2 The dialog_tracker

The dialog_tracker component establishes a link between the redirection module and the dialog_handler. This way, the implementation of the dialog_handler is totally independent of the architecture and the operating system. The interactions between the dialog_tracker and the kernel are implemented by a netlink socket and functions of the libnetfilter_queue and libnetfilter_conntrack libraries. The libnetfilter_conntrack library enables a user space program to be notified when connections are created or destroyed. The interactions between the dialog_tracker and the dialog_handlers are implemented by a AF_INET socket.

The dialog_tracker receives a verdict request through the libnetfilter_queue library. The request is formatted to be readable by the dialog_handler. The request content determines a dialog_handler and the request is forwarded to it. The dialog_handler responds with a verdict (packet reject or redirected), which is returned through the libnetfilter_queue library.

The dialog_tracker receives an event from the libnetfilter_conntrack library each time a connection is created or destroyed. This event is formatted to be readable by the dialog_handler. Again, event content determines which particular dialog_handler is selected to receive the event.

At any moment, one of the dialog_handlers may decide to add or remove information in the memory cache of the kernel redirection module. The information is received by the dialog_tracker through a AF_INET socket. It is sent to the redirection module through a netlink socket.

The dialog_handlers and the dialog_tracker may be executed on different machines. In that case, the packets allowing these components to communicate cross the PREROUTING chain. The redirection module intercepts them. Since these packets are needed to support the communication inside our redirection mechanism, the redirection module must not reject them or send them to the dialog_tracker. To this end, the dialog_tracker has been extended to upgrade the redirection module so that it can accept these packets.

3.3 The dialog_handler

The dialog_handler component decides how to proceed with the intercepted packets. Several algorithms can be used for this purpose. All have two inputs: The packet, as well as the historical information recorded in a database.

Let us justify the existence of such a database. The redirection decision could have been taken by the redirection module in the kernel space. By implementing it in the user space, we can use historical information stored in a database for our redirection mechanism. For example, if an attacker is redirected towards a particular IP address, it is important that this attacker be redirected towards the same IP address if he comes back to our system a few days later. This particular association (IP addresses of the attacker/IP address of the redirection) must be stored in a database. If the redirection decision was implemented in the redirection module, we could not benefit from the use of databases (it is not possible from the kernel space) and this would prevent us from recording useful information.

The consistency of the redirection mechanism strongly depends on the algorithms used to make decisions about the packets. In some cases, this decision is not particularly easy to make. Let us present two examples: When two connections, initiated from two different IP addresses, try to connect to the same IP address, what decision must be made by the dialog_handler? Two different IP addresses on the Internet may be used by the same attacker.

As a consequence, we made the following choice: all connections towards the same IP address of Internet must be redirected towards the same honeypot and connections towards different IP addresses of Internet must be redirected towards different honeypots. As we cannot redirect all Internet IP addresses, we choose to redirect only a subset of IP addresses. Different strategies may be used to select said subset. This topic is beyond the scope of this paper and is part of future work.

4 Performance evaluation

One of the main issues raised by the redirection mechanism is the necessary transparency from the end user's[3] perspective. It is obvious that the modifications implemented on netfilter, together with the communications between the user and kernel spaces, may slow down network connections. Therefore, it is important to evaluate the latency introduced by our mechanism and to ensure that the overhead is sufficiently low as to prevent detection of the redirection mechanism by the attacker.

[3] By user we mean blackhat or software developed by blackhats.

$N_{address}$	T_{scan} (seconds)
2^8	4
2^9	4.33
2^{10}	7
2^{11}	12.3
2^{12}	23.5
2^{13}	60.5

Table 1: Evaluation of the two strategies for the second method.

The management of new session establishment attempts is the most time-consuming part of our redirection mechanism. Once the verdict for a given session has been given (i.e redirect the flow or not), most of the work is subsequently done by a single process in kernel space. In contrast, new session management involves cooperation between processes that run in both the user and kernel spaces.

Thus, in order to estimate the latency due to the redirection mechanism, it is important to analyze the performance of our mechanism during its most time consuming phase.

The mechanism is particularly stressed during network scans. First, we had to implement scans in a sufficiently efficient way so that the latency generated by the mechanism would be perceptible: Were the scan too slow, we would fail to see discernable latency.

We discuss different alternatives for implementing network scans and estimating experimentally the latency incurred by the redirection mechanism.

Commonly used scanning tools like nmap were not fast enough to stress our application which is why we developed our own algorithm. We considered the following case: Network scan of $N_{address}$ different addresses, a given TCP port, and simulation of n_{host} available among these addresses. We decided to send a single SYN segment to each address, and considered a host unavailable if we did not receive any answer after $T_{timeout}$ seconds.

Unless $n_{host} = N_{address}$, at least one of the timeouts will expire, which implies a lower bound for the duration of such a scan $b_{Inf} = T_{timeout}$.

We can distinguish two ways of sending SYN segments:

1. Launch n_{thread} connection attempts in parallel and wait for the verdicts of these connections before launching the next ones.

2. Launch the $N_{address}$ connection attempts one after the other without waiting for verdicts. Then, analyze packets once they arrive, or consider a given host unavailable once $T_{timeout}$ second elapsed since the sending of the corresponding SYN segment.

With the first method, if we have $N_{address}$ parallel threads and assume that at least one timeout will expire, b_{Inf} can eventually be reached. However, this solution may not be scalable. For example, the Microsoft Windows Thread API[4] imposes a limit of $n_{thread} \leq 2028$. We therefore need to equally distribute the $N_{address}$ addresses among the n_{thread} threads. If we do the following Euclidean division, we obtain:
$N_{address} = q \times n_{thread} + r$, $0 \leq r < n_{thread}$.

Therefore, if we assign $q + 1$ addresses to the first r threads[5] and q to the remaining ones, we can conclude that a scan where the timeout always expires will last more than $q \times T_{timeout}$, and that n_{thread} threads will be necessary to obtain this value.

If we implement the scan according to the second method, the main difficulty resides in the management of timers expiration. We need to store $N_{address}$ data structures in the worst case. In addition, the complexity of the associated algorithm is high. As a result, the necessary time to manage $N_{address}$ timers can be higher than the one necessary to manage 2 times $\frac{N_{address}}{2}$ timers, by scanning the first half, and then the second half of the $N_{address}$ addresses.

We experimentally evaluated these two strategies considering different values for $N_{address}$ from 2^8 to 2^{13}. The results for the second method are presented in the table 1:

We notice that until $N_{address} \leq 2^{12}$, $T_{scan}(N_{address}) < 2 \times T_{scan}(\frac{N_{address}}{2})$. This result means that in those scan ranges, it is more efficient to launch one complete scan instead of launching two consecutive ones, each for each half of the considered address range. However, this is no longer true starting from $N_{address} = 2^{13}$. Therefore, we chose to launch at most 2^{12} simultaneous scans in the following experiment.

[4] http://msdn2.microsoft.com/en-us/library/ms682453.aspx

[5] Possible because $r < n_{thread}$.

$N_{address}$	T_{scan} without redirection (seconds)	T_{scan} with redirection (seconds)
2^8	3.0	3.5
2^9	4.5	5.0
2^{10}	7.0	6.5
2^{11}	12.0	13.0
2^{12}	23.5	26.0
2^{13}	48.5	54.5
2^{14}	93.0	114.0
2^{15}	117.5	255.5
2^{16}	219.0	673.5

Table 2: Evaluation of the latency.

We implemented the two scanning methods discussed above in a C program that was executed on the `Windows XP SP1` operating system with the register value `MacTcpRestransmissions`[6] assigned to 0, the default value $3s$ to $T_{timeout}$, $n_{host} = 3$ and $N_{address} \in \{2^8, ..., 2^{16}\}$.

We used:

1. The `Thread` and `Winsocket` API of `Microsoft Windows` using synchronous SOCK_STREAM sockets for the first method.
2. The `Microsoft Windows` API and asynchronous SOCK_STREAM `Winsocket`s for the second method.

This system was executed on the QEMU virtual machine with 128MB RAM capacity. The host system was a 3.00 Ghz Pentium 4 with 1 GB of RAM, running a 2.6.19 Linux kernel with the `kqemu` kernel module. It used `bridged tap devices` for network access to the virtual system. We assigned the maximum priority to the QEMU process with `nice`.

We obtained the following results with the second method, reported in the table 2.

Compared to the first method discussed in the beginning of this section (i.e., parallel execution of $n_{threads}$), we demonstrate the efficiency of the second method by means of case $N_{address} = 2^{16}$. In keeping with the previous notation, we have $T_{minMethod1} \geq q * T_{timeout}$ with $q = \lfloor \frac{N_{address}}{n_{thread}} \rfloor$.
So to obtain the same scanning time T_{scan} as the second method, we should roughly have: $n_{threads} \simeq \frac{N_{address}}{T_{scan}} \times T_{timeout}$. With $T_{scan} = 219.0$ s the numerical computation gives $n_{thread} = 897$, which isn't implementable in our experimental platform[7]. The first observation we can make is that b_{Inf} is reached for $N_{address} = 2^8$, and that $|T_{scan} - b_{Inf}|$ then increases[8]. We should, however, evaluate the time $T_{sending}$ needed to send $N_{address}$ SYN segments, as well. If the last scanned host is not available, $b_{Inf} = T_{sending}(N_{address}) + T_{timeout}$, and therefore the new $|T_{scan} - b_{Inf}|$ should decrease.

However, our program was not designed towards this optimal value; just enough speed to express the latency of our redirection mechanism. With this scanning method, we see that until $N_{address} = 2^{12}$, the proposed mechanism does not incur a significant overhead from the performance point of view. Indeed, the execution times with and without connection redirection are very close. The difference, generally related to the experimental conditions, is not significant.

5 Experiments

This section describes two experimental studies that have been set up to validate and illustrate the usefulness of the redirection mechanism described in this paper.

5.1 Malware execution observation

As described in Figure 4, the first experiment concerns *botnets* infiltration by executing and observing in a sandbox *malware* downloaded from the *Nepenthes* honeypot [9].

[6] This ensures that a single SYN is sent.

[7] With more than $n_{max\_thread} = 64$ threads, one scanning thread finishes its job whereas others haven't started their execution yet. This results in a consecutive execution of some threads whereas we want them to run concurrently. Therefore launching more than n_{max_thread} scanning threads is useless in our platform.

[8] For the sake of comparison, the reader should know that with nmap the scanning time measured for $N_{address} = 2^{10}$ is 53.755 s, to be compared to 7 s with our algorithm.

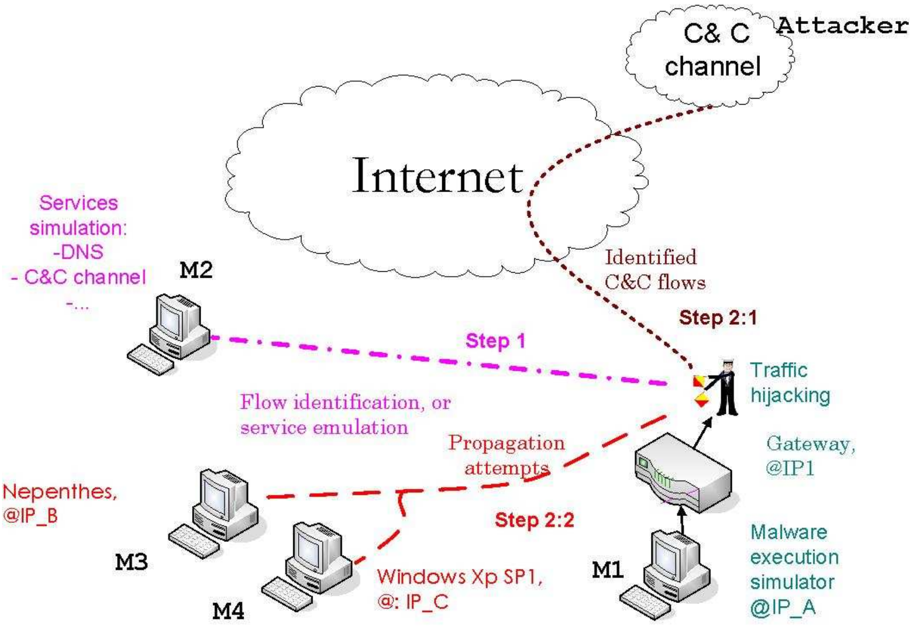

Figure 4: Sandbox

A *botnet* is a network of compromised computers that can receive orders from a controller to launch, among other things, distributed attacks. Most of the time, compromised machines and the controller communicate by using a protocol similar to *IRC*. Our observation methodology consists in first identifying the `Command and Control` flows associated to a *malware* instance by redirecting all the packets from the *malware* to a machine we control [10] (step 1). In a second step, we launch this malware by letting such `Command and Control` flows communicate with the Internet (step 2:1), and observe the attack process unfolding by redirecting the flows received from the attacker and by executing the malware (step 2:2).

We chose to execute the *malware* on an instance of QEMU with *Microsoft Windows XP* OS installed. We also added two honeypots (M3 and M4) on two other QEMU processes to simulate the vulnerabilities on two different targets:

1. M3 corresponds to a low-interaction honeypot (*Nepenthes*) to see if the level of interaction is high enough to correctly simulate the vulnerability and hence enable a successful attack.

2. M4 is a a high interaction honeypot (*Windows XP SP1*) to hedge for the case when the low-interaction honeypot fails to correctly simulate the vulnerability.

We connected those virtual machines with `bridged tap devices`, with the following addresses: **@IP_A** (the *malware*) M1, **@IP_B** M3, **@IP_C** M4.

The *malware* has Internet access for `Command and Control` and for `DNS` flows only. Those flows are `source NATted` to a public $IP1$ address. The flows targeting TCP $\{135, 139, 445\}$ ports are handled by our redirection mechanism, and others are blocked.

After connecting to an *IRC* server located at a non-standard TCP IRC port(5190), the *malware* instance received the following order:

```
:hub.24324.com 332 seivNbtC #last:=BGX5tCMl9HMuPlQRIfr7ZDvrWvjsrx3QcTGwmkNACosllT7o+6BL
/FkEl1LzB/AkO7BSNYd1ycZi/zOu/AWHE5fJNTO2YoaGogFZbHO3O9Z/OVp4bDrWR3gJyIug2Eee3JVQHBn/fWG
6ANlrYrOmZbtKuh
```

Apparently, this *IRC* channel uses some obfuscation techniques to evade signature based approaches aimed at detecting *IRC* commands used by different bots [11].

The *malware* then launched a network scan for open TCP ports 135, on an address range generated from $IP1$. The *malware* has no way to obtain this information by its own, therefore this information has been communicated in the encoded message. Our mechanism successfully redirected two connection attempts to both of the honeypots. After noticing that two addresses had TCP port 135 open, two exploits targeting the honeypots were successfully sent to the two honeypots.

Finally the *malware* reported those successes on the Command and Control channel, by numbering the number of successes[9]:

```
PRIVMSG #last :-04dcom2.04c- 1. Raw transfer to @IP_B complete.
PRIVMSG #last :-04dcom2.04c- 2. Raw transfer to @IP_C complete.
```

Thanks to the redirection mechanism, we could partly understand an encoded Command and Control channel and observe in more detail the resulting behavior of the malware.

The reader should notice that what happened during our experiment could rouse the suspicion of the *botnet* administrator: After scanning an address range, its *malware* instance claimed that it managed to compromise two addresses that were not part of the range[10]. Here again, our mechanism could be improved.

5.2 Observation of attacker activities

The second experimentation consists of deploying a high-interaction honeypot implementing the redirection mechanism on the Internet. The objective of this experiment is to collect data on real attacks. Such data will enrich our knowledge on the attackers behavior and malicious activities on the Internet. Moreover, the thorough analysis of the results obtained from this experimentation and the comparison with the results from [4] should serve as a useful gauge of our mechanism and identify its potential weaknesses.

As presented in Figure 5, the experimental platform consists of four virtual machines running on top of one real machine. QEMU [12] is used in order to run the virtual machines. The kernel of the real machine has been modified to include our redirection mechanism, as described in Section 3.

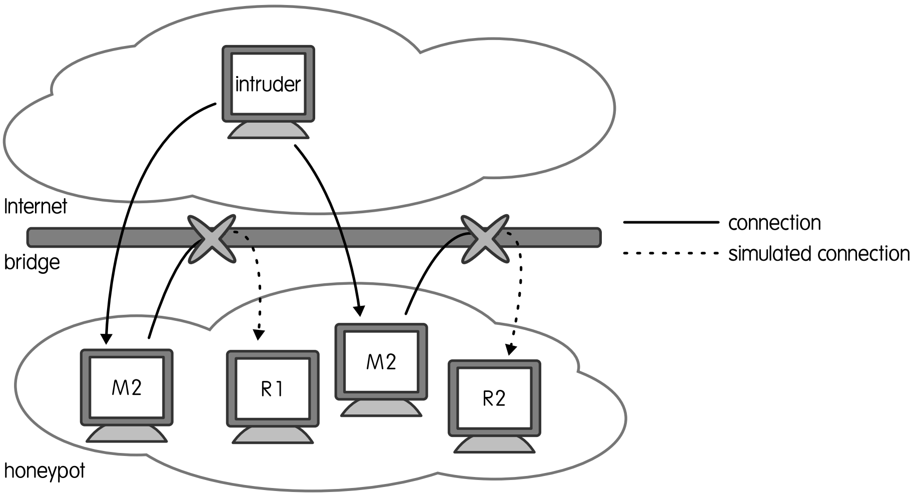

Figure 5: Platform for observation of attackers activities

More precisely, two out of four virtual machines, M1 and M2, are directly accessible from the Internet thanks to a SSH connection. The two other machines R1 and R2 are not accessible from the Internet. We modified the kernel of each virtual machine (Linux) as described in [4] in order to observe and collect all the commands executed by any attacker who attempted to connect to the machine.

Accounts with weak passwords have been created on machines M1 and M2. Attackers can then break into these machines provided that they find the right login and password information. We have also configured our dynamic redirection mechanism to automatically redirect some of the SSH scans from machines M1 and M2 to machines R1 and R2. As a consequence, if an attacker for example breaks into machine M1 and initiates some SSH scans from that machine against a particular set of IP addresses, the attacker will believe some of these scans to have been executed against real targets, when in reality, they will be redirected towards R1 or R2.

The platform has been operational for one month and a half. The results we obtained are, for the moment, partial. We nevertheless checked that the dynamic redirection mechanism is fully operational. So far, we did observe attackers breaking into machine M1 or M2 and initiating SSH scans against sets of remote IP addresses. While in [4] we did not enable this kind of scans, the implementation of our redirection mechanism has enabled us to include such scans and to make the attackers believe that their scans were successful. Thus,

[9] See ...last:-<exploited_flaw>- <nb_of_success>.

[10] @IP_B,@IP_C were not in the range generated from $IP1$.

the implementation of the redirection mechanism is fully operational. This is illustrated for example in the following captures:

```
james@M1:~/rep_hacker$ ./unix  66..
[+][+][+][+][+] UnixCoD Atack Scanner [+][+][+][+][+]
[+]   SSH Brute force scanner : user & password   [+]
[+]        Undernet Channel : #UnixCoD            [+]
[+][+][+][+][+][+][+] ver 0x10  [+][+][+][+][+][+][+]
[+] Scanam: 66.221.4.* (total: 2) (1.6% done)
66.221.8.* (total: 2) (3.1% done)
66.221.12.* (total: 2) (4.3% done)
66.221.16.* (total: 2) (5.9% done)
66.221.19.* (total: 2) (7.5% done)
66.221.23.* (total: 2) (9.0% done)
66.221.27.* (total: 2) (10.2% done)
66.221.30.* (total: 2) (11.8% done)
66.221.34.* (total: 2) (13.3% done)
66.221.38.* (total: 2) (14.5% done)
66.221.41.* (total: 2) (16.1% done)
66.221.45.* (total: 2) (17.6% done)
66.221.49.* (total: 2) (18.8% done)
66.221.52.* (total: 2) (20.4% done)
66.221.56.* (total: 2) (22.0% done)
66.221.60.* (total: 2) (23.1% done)
```

The display of `total: 2`, meaning that two scans were successful, shows that the redirection mechanism is operational. As a matter of fact, the redirection mechanism was configured in order to automatically redirect two IP addresses towards the two machines `R1` and `R2`. This kind of SSH scans and the associated dynamic redirections were observed several times on the platform since the beginning of the experimentation.

On the other hand, and that is why our results are still partial, we did not observe for the moment any attacker trying to connect from `M1` or `M2` to the IP addresses that were successfully scanned. In the above example, the two IP addresses of the `66.x.y.z` network were not used by the attacker after the successful scan.

We may try to explain this behavior in several ways. The attacker may judge the number of successfully scanned IP addresses insufficiently high to justify an attack on this network. It is also possible that the attacker we observed is only responsible for scanning IP addresses but not for directly attacking them once successfully scanned. In that case, we can imagine that he simply stores the list of the compromised IP addresses and that they will be attacked later, from our `M1` and `M2` machines or from other sites. It is difficult to answer these questions at this stage due to the short period of time since the platform has been deployed. The continuation of our experiments and the collection of more data will enable us to have better insights about these issues.

6 Conclusion

The efficiency of attack monitoring and observation mechanisms implemented in the honeypots is directly related to the level of interaction and the possibilities offered by the honeypots to the attackers. Traditional honeypot implementations generally restrict or forbid outgoing connections to the Internet for legal liability reasons to prevent the honeypots from being used as a stepping stone to attack third party machines. The dynamic connection redirection mechanism presented in this paper is aimed at providing enhanced possibilities for observing attack scenarios and their progress on the Internet across different machines.

The proposed mechanism has been implemented on a GNU/Linux environment. Nevertheless, it can be easily ported to other environments. Two experimental studies have been set up to validate the feasibility of the proposed approach and analyze its benefits and weaknesses. The partial results obtained so far show that the proposed mechanism offers enhanced possibilities for observing attacks. However, the current design of the proposed mechanism needs to be improved, e.g., to make it more transparent and difficult to detect by skilled attackers. Further analysis of the data collected from the high interaction honeypot (including the redirection mechanism currently deployed on the Internet) will prove useful in assessing and extending the proposed mechanism.

Acknowledgement

This research has been partially supported by the French ACI project CADHO, and by the European Commission (Projects ReSIST IST 026764 and CRUTIAL IST 027513)

References

[1] Niels Provos. Honeyd - a virtualhoneypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany*, February 2003.

[2] Corrado Leita, Marc Dacier, and Frédéric Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *RAID 2006, 9th International Symposium on Recent Advances in Intrusion Detection, September 20-22, 2006, Hamburg, Germany - Also published as Lecture Notes in Computer Science Volume 4219/2006*, Sep 2006.

[3] M Bailey, E Cooke, D Watson, F Jahanian, and N Provos. A hybrid honeypot architecture for scalable network monitoring. Technical Report CSE-TR-499-04, University of Michigan, 2004.

[4] E Alata, V Nicomette, M Kaâniche, Marc Dacier, and M Herrb. Lessons learned from the deployment of a high-interaction honeypot. In *EDCC'06, 6th European Dependable Computing Conference, October 18-20, 2006, Coimbra, Portugal*, Oct 2006.

[5] Jason Nieh and Ozgur Can Leonard. Examining VMware. *j-DDJ*, 25(8):70, 72–74, 76, aug 2000.

[6] Dongyan Xu Xuxian Jiang. Collapsar: a vm-based architecture for network attack detention center. In *13th USENIX Security Symposium, San Diego, CA*, August 2004.

[7] David Duncombe, George Mohay, and Andrew Clark. Synapse: auto-correlation and dynamic attack redirection in an immunologically-inspired ids. In *ACSW Frontiers '06: Proceedings of the 2006 Australasian workshops on Grid computing and e-research*, pages 135–144, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[8] Duncan Napier. IPTables/NetFilter – Linux's next-generation stateful packet filter. *j-SYS-ADMIN*, 10(12):8, 10, 12, 14, 16, December 2001.

[9] Felix Freiling, Thorsten Holz, and Georg Wicherski. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. Technical Report AIB-2005-07, RWTH Aachen, 2005.

[10] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Internet Measurement Conference 2006 (IMC'06), Proceedings of*, October 2006.

[11] John Kristoff. Botnets. In *32nd Meeting of the North American Network Operators Group*, October 2004.

[12] Fabrice Bellard. Qemu, a fast and portable dynamic translator. pages 41–46.