Defeating Honeypots: Network Issues, Part 2

by Laurent Oudot *and* Thorsten Holz

last updated October 7, 2004

0. Continuing from Part One

It is a difficult problem to deploy honeypots, technology used to track hackers, that cannot be detected. The value of a honeypot is in its ability to remain undetected. In part one of this article we introduced some of the issues related to discovering and fingerprinting honeypots, and then we discussed a few examples such as tarpits and virtual machines. Now we'll continue the discussion with more practical examples for detecting honeypots, including Sebek-based honeypots, snort_inline, Fake AP, and Bait and Switch honeypots.

If you have not yet read part one of this series, please familiarize yourself with that article before continuing on.

1. Practical examples (continued)

1.1 Sebek-based Honeypots

Sebek [ref 0] is client/server based application, and it is the primary data capture tool used by honeynet researchers to capture the activities of an attacker found inside a honeypot. It is a kernel-based rootkit that hijacks the read() system call and it is therefore able to record all data accessed via read(). Sebek lives entirely in kernel-space and has access to all data read, so it is able to access most communication unencrypted. It can, for example, log SSH-sessions, recover files copied with SCP and record all passwords used by intruders. The recorded data is sent in a covert manner via UDP to the Sebek server, the other part of Sebek's client/server architecture. This transmission is done by modifying the kernel in order to hide these outgoing packets such that an intruder cannot see them. In addition, all network counters and data structures have to be adapted in order to make detecting these changes more difficult. Further information about Sebek and its architecture can be found on the Honeynet site. [ref 1]

It is possibile to detect the presence of Sebek by using the network layer, however. Because Sebek records everything which is accessed via read() and then sends this data over the network, some congestion effects will be visible if we read lots of data coming out of the system. When we read a single byte via a read(1) call, Sebek has to transfer nearly 100 bytes of data, including all the network headers, over the network to the host doing the logging. So if we are able to do a read(1) some tens of thousands of times per second, this will lead to a congested network and finally to dropped packets.

We are able to generate many read(1) calls with the help of the dd command:

```
user@honey:~ dd if=/dev/zero of=/dev/null bs=1
```

And we are able to identify a congested network with the help of the ping command, as outlined below.

We first ping a local IP-address (for example, the gateway) in order to get an overview of the current congestion of the network. Then dd is started in the background and we run the ping command again. If Sebek is installed on a host, this leads to a significant increase in the average round-trip time. In practical tests, the average round-trip time grew from 0.7 milliseconds to over 4800 milliseconds.

There are also further methods to detect and circumvent the presence of Sebek if we use layers than the network layer. These attacks and further techniques will be covered in another article.

1.2 Snort_inline

Snort_inline [ref 2] is an inline packet modification engine, and it provides for packet rewriting of what looks like something dangerous into something harmless. It is basically a modified version of the popular Intrusion Detection System (IDS) Snort [ref 3] and adds several new rule types (drop, sdrop and reject) to tell iptables whether a packet should be dropped, rejected, modified, or allowed to pass based on a Snort rule set.

An example for this technique is the replacement of the string /bin/sh in shellcode with the string /ben/sh. Another example is the replacement of certain characteristic patterns in network traffic in order to render these attacks useless, as shown in the following listing which modifies x86-shellcode:

```
alert ip $HONEYNET any -> $EXTERNAL_NET any
   (msg:"SHELLCODE x86 stealth NOOP"; sid:651;
    content:"|EB 02 EB 02 EB 02|";
    replace:"|24 00 99 DE 6C 3E|";)
```

This kind of Intrusion Prevention System (IPS) is used in GenII (2nd generation) Honeynets [ref 4] in order to identify and block known attacks. Snort_inline is also necessary because it hampers the possibility of an attacker bouncing from a compromised honeypot to attack further systems. So Snort_inline also adds some safety from a legal point of view.

But the dropping or modification of outgoing packets leads to an easy way for attackers to spot the presence of Snort_inline: If outgoing packets are dropped, that might look suspicious for an attacker, like a kind of black hole. If outgoing packets are modified, this will be detected by an sophisticated attacker. She will send packets with malicious content to another host which she controls and check whether the packet arrives at the destination in unaltered form (albeit with some small things changed, like a decremented Time To Live). If the content of the packet differs, she will probably conclude that this is a bit fishy. The stealthiness of Snort_inline can be increased with fewer rules, but this leads to more attacks that will not be noticed. A trade-off between fewer rules, but this leads to more attacks that will not be noticed. A trade-off between stealthiness and safety has to be found.

Moreover, GenII Linux Honeynets typically use the netfilter/iptables-functionality [ref 5] of the kernel to implement connection limiting. This prevents the abuse of a compromised honeypot for denial-of-service attacks or mass scanning and nevertheless allows an attacker to download toolkits or setup automated bots. As an example, the default configuration of the rc.firewall script provided by the Honeynet Project allows 15 outgoing TCP-connections and 50 outgoing ICMP packets per day.

```
[...]
### Set the connection outbound limits for different protocols.
SCALE="day"
TCPRATE="15"
UDPRATE="20"
ICMPRATE="50"
OTHERRATE="15"
[...]
```

This connection limiting also leads to a problem with the secrecy of a honeypot: an aggressor can simply try to open many outgoing connections to different hosts or send a lot of ICMP echo request packets to varying hosts. If she notices that after some time no more outgoing packets can reach the target, then this is a clear sign that something weird is going on and that she probably fell for a honeypot. Again, a trade-off between stealthiness and safety has to be found. If you allow more outgoing packets, the detection will be harder, but an attacker has more ways to abuse the honeypot.

1.3 Fake AP

Fake Access Point [ref 6] is a Perl-script that generates bogus 802.11b beacon frames with random ESSID and BSSID (MAC-address) assignments. It sends these frames on a random channel and is also able to simulate the wired equivalent privacy (WEP) protocol. A normal access point will "hide in sight amongst Fake AP's cacophony of beacon frames" [ref 6] and therefore this kind of tool can also be used as a wireless honeypot: just deploy one Linux machine running Fake AP near your wireless network and watch for any suspicious traffic. Legitimate users will know the SSID of the network and can connect without problems. Malicious users will try to connect to your network with different SSIDs and can thus be spotted easily.

In its current version, Fake AP does not generate fake traffic on one of the simulated access points and hence there is an easy way to detect the presence of Fake AP: This tool only sends beacon frames and also does not send any real traffic. So an attacker can just monitor the network traffic and easily notice the presence of Fake AP.

1.4 Bait and Switch Honeypots

Traditionally, information security follows the classical security paradigm of "Protect, Detect and React". In other words, try to protect the network as best as possible (such as by using firewalls), detect any failures in the defense (with intrusion detection systems), and then react to those failures (perhaps by alerting the admin via mail). The problem with this approach is that the attacker has the initiative, and she is always one step ahead. The Bait and Switch Honeypot [ref 7] is an attempt to turn honeypots into active participants in system defense. It helps to react faster on threats. To archieve this goal, the Bait and Switch Honeypot redirects all malicious network traffic to a honeypot after a hostile intrusion attempt has been observed. This honeypot is partially mirroring the production system and therefore the attacker is unknowingly attacking a trap instead of real data. Thus the legitimate users can still access all data and work on the real systems, but the attacker is lured away from all interesting systems. As an additional benefit, the actions of the aggressor can be observed and then his tools, tactics and motives can be studied. A Bait and Switch Honeypot is based on Snort [ref 3] , iproute2 [ref 8], netfilter/iptables [ref 5] and some custom code.

An attacker might detect the presence of a Bait and Switch Honeypot by looking at specific TCP/IP values like the Round-Trip Time (RTT), the Time To Live (TTL), the TCP timestamp, and others. After a switch event, the attacker will stop talking to the real computer, and will start to interact with the honeypots. During the switch from the real system to the honeypot, a sudden change in the IPID can be observed. Previous TCP/IP values will also probably change after the switching has taken place and this can be observed by a sophisticated attacker.

Once again, tcpdump and friends are valuable tools for attackers to gather information about what is going on. Furthermore, the honeypot will probably differ noticeably from the real system. The attacker will presumably try to find a way to identify the honeypot by looking at specific differences that might exist between the real system and the honeypot. Notice that some attackers will use multiple IP addresses as sources of their attacks, in order to defeat such kinds of IPS. For example, if the shellcode of the attacker is a reverse shell that connects back to an IP source which is different from the IP that sent the exploit, the IPS will not be able to change anything. The modus operandi will differ with every deployment of a Bait and Switch Honeypot, and so the operator of this kind of honeypot has to take great care in the setup process.

2. Summary

It is clearly a difficult problem to deploy honeypots in a very stealthy manner -- and the effectiveness of honeypot technology exists only if an attacker does not know that she is attacking a trap and not a real system. The operator of a honeypot therefore must be aware of many of the possibilities for attackers to identify honeypots. As outlined in the previous sections, and in part one of this article, there are many ways to detect the presence of a honeypot if an attacker simply looks at the network layer.

In this article series we explained how attackers often behave when they try to identify honeypots, and we gave some technical examples of some different methods. We hope that this helps security specialists who want to setup and use honeypots. It is important that the operator of a honeypot customizes and adapts it to his own needs. For example, the MAC address (in case of Labrea or User-mode Linux) or error messages should be customized. In order to be a step ahead of attackers, the coders of honeypot software must also continually update and change their programs to avoid detection -- the arms-race between whitehats and blackhats has begun.

Note that there are even commercial tools such as Honeypot Hunter [ref 9] that use anti-honeypot technology. Honeypot Hunter checks with lists of HTTPS and

SOCKS4/SOCKS5 proxies for honeypots, and it is used by spammers in order to detect the presence of tarpits or other kinds of honeypots/proxies. Honeypot hunter works by opening a local (fake) mail server on port 25 (SMTP) and connects back to itself through the proxy. A honeypot is detected if the proxy reports that the connection is up but the tool does not receive a connection to this simulated mail server. This approach identifies most invalid proxies and honeypots and the approach is quite simple. But it can be circumvented easily if you allow allow a small, but limited, number of outbound connections from the honeypot/proxy. The mere availability of such a programm shows that the cyber battle between detection and stealthiness of honeypots has not only begun, but that an arms-race will likely follow.

3. Conclusion

This paper gave an overview of current state-of-the-art of honeypot detection by looking at the network layer. Further papers on this topic will move to the system world and the application layer and explain how to identify a honeypot by looking at these higher layers.

4. References

[ref 0, Sebek, by Edward Balas et al.: http://www.honeynet.org/tools/sebek/]

[ref 1, Know Your Enemy: Sebek: http://www.honeynet.org/papers/sebek.pdf]

[ref 2, Snort_inline, by Rob McMillen, William Metcalf, Jed Haile and Victor Julien: http://snort-inline.sourceforge.net/]

[ref 3, Snort, by Martin Roersch: http://www.snort.org/]

[ref 4, GenII honeypots, by the Honeynet Project http://www.honeynet.org/papers/gen2/index.html]

[ref 5, netfilter/iptables: http://www.netfilter.org/]

[ref 6, Fake AP tool, by Black Alchemy: http://www.blackalchemy.to/project/fakeap/]

[ref 7, The Bait and Switch Honeypot, by Jack Whitsitt: http://violating.us/projects/baitnswitch/]

[ref 8, iproute2, by Alexey Kuznetsov: ftp://ftp.inr.ac.ru/ip-routing/, http://www.linuxgrill.com/iproute2-toc.html]

[ref 9, Honeypot Hunter: http://www.send-safe.com/honeypot-hunter.php]

4.1 Credits

Thanks to Kelly Martin, Lance Spitzner, Dragos Ruiu, Maximillian Dornseiff, Felix Gärtner and folks from the German and the French Honeynet Projects.

5. About the Authors

Thorsten Holz is a research student at the Laboratory for Dependable Distributed Systems at RWTH Aachen University. He plans to graduate next spring and continue

Systems at RWTH Aachen University. He plans to graduate next spring and continue his studies as a Ph.D. student. His research interests include the practical aspects of secure systems, but he is also interested in more theoretical considerations of dependable systems. He is one of the founders of the German Honeynet Project and have given talks at both Black Hat and Defcon.

Laurent Oudot is a computer security engineer currently employed by the Commissariat a l'Energie Atomique in France. On his spare time, he is a member of the team Rstack with other security addicts. He has presented at different security events including Cansecwest, Defcon, Black Hat USA and Asia, HOPE, and others. Regarding honeypots, Laurent is a member of the steering committee of the Honeynet Alliance and is an active member of the French Honeynet Project.