

Know Your Enemy: GenII Honeynets

Easier to deploy, harder to detect, safer to maintain.

[HoneyNet Project](#)

<http://www.honeynet.org>

Last Modified: 12 May, 2005

The GenII (2nd generation) HoneyNet is the next step in the evolution of honeynet technology. Based on combining old and new techniques, the GenII HoneyNet can increase the flexibility, manageability, and security of honeynet deployments. This paper introduces the technology used in a GenII HoneyNet architecture. However, before you proceed, it is assumed that you have already read and fully understand the concepts, risks, and issues of HoneyNets outlined in [Know Your Enemy: HoneyNets](#). It is critical that you understand the basic concepts and risks of honeynets before covering the technical details.

This paper is an end-to-end overview of the components of a GenII HoneyNet. You can use the methods discussed below to manually build your own honeynet. However, we suggest instead that you use the [HoneyWall CDRom](#), as it automates all of the functionality discussed below (and much more). The first section introduces the concept of a honeywall, the administrative mechanism that will be the focus of our network's management capability. In the Data Control section we discuss the process of limiting the actions attackers can take on our network. The third section, Data Capture, details methods to covertly capture attacker activity at both the host and network level. The Automated Alerting section covers methods to notify administrators of possible successful attacks. Finally, the Testing section presents a series of ways to test the configuration of the previous steps. The deployment we describe is based on a honeywall running a Linux 2.4.X kernel on standard x86 hardware. You are not required to use the tools or methods discussed here, you can use any technologies you feel the most comfortable with, as long as it provides the same functionality.

The Architecture

As we discussed in [KYE: HoneyNets](#) a honeynet is an architecture, a highly controlled network used to contain and analyze attackers in the wild. How you deploy that architecture is up to you. We also discussed the document [HoneyNet Definitions, Requirements, and Standards](#) which outlined the requirements of HoneyNet deployments. This paper details one specific method of meeting those requirements. The [HoneyWall CDRom](#) uses these tools and techniques, in addition to new functionality such as data analysis. The key element of any honeynet is the gateway, this is what separates the honeynet victims from the rest of the world. The gateway acts as a wall, in fact we call the gateway a honeywall. All traffic entering or leaving the honeynet must go through this honeywall. This becomes your command and control center for the honeynet, all the magic happens here. You can see an example of our honeynet architecture in [Figure A](#). In this example, our gateway is a layer two bridge. A

layer three routing gateway can be used, but a bridge is preferred, as it is harder to detect. In our diagram, our honeynet is deployed on an internal 192.168.1.0/24 network. In the past, most honeynets have traditionally been deployed on external or perimeter networks. With the use of a layer two gateway, honeynets can now also integrate with internal networks, as ours is. This allows us to track and learn not only about external threat, but potentially internal threats.

The honeywall (the bridge gateway) separates production systems from the honeynet network, populated by our victim targets. The external interface of our gateway (eth0) is connected to the production systems' network. The internal interface of our gateway (eth1) is connected to the honeynet systems' network. Since this is a bridge, both internal and external systems are on the same IP network. We also have a third interface (eth2). The purpose of this interface is for remote administration of the gateway, including moving any logs or captured data to a centralized point. The internal and external interfaces will be in bridging mode so they have no IP address assigned to them. However, the third interface (eth2) has an IP stack assigned to it, in this example the IP address 10.1.1.1. This network is a separate, secured network used for administration purposes. The advantages of this architecture is the gateway is difficult to detect, as there is no routing hops, no TTL decrement, nor any MAC addresses associated with the gateway. Also, we can simplify honeynet deployment by combining both Data Control and Data Capture on the single gateway. The next step is to build our gateway to support this architecture. For our gateway we are using a minimized, secured installation of Linux. It is critical that this is a trusted system, that no attacker can access our gateway. Next, we have to ensure our gateway supports IPTables firewall in bridging mode. Most Linux distributions support this by default. If your system does not, you can get bridging kernel path at <http://ebtables.sourceforge.net/>. IPTables is critical, as not only will it help secure our gateway, we will be using it for Data Control (discussed later). Firewalling capabilities are critical, as they are not only used to secure your honeywall gateway, but used for Data Control.

Data Control

As we discussed in the [KYE: Honeynet](#) paper, the purpose of Data Control is to prevent attackers using the honeynet to attack or harm other non-honeynet systems. Data Control mitigates risk, it does not eliminate it. With Data Control, one of the questions you have to answer is how much outbound activity do you control? The more you allow the attacker to do, the more you can learn. However, the more you allow the attacker to do, the more harm they can potentially cause. So, you have to contain their activity enough so they can't harm other folks, but you can't contain it too much or minimize what you learn. How much you allow an attacker to do ultimately depends on how much risk you are willing to assume. To make this even more challenging, we have to contain the attacker without them knowing we are containing them. To accomplish just this, we will be implementing two technologies, connection counting and NIPS. Connection counting is when we limit how many outbound connections a honeypot can initiate. NIPS (Network Intrusion Prevention System) can block (or disable) known attacks. Combined, these two technologies make a redundant and flexible data control mechanism. We will be implementing both technologies on our layer two gateway. We implement Data Control on the gateway because this is where all inbound and outbound traffic must flow through, it's a choke point for the attacker's activity.

Once you have your gateway configured as described in part one, the next step is implementing connection limiting, we contain how many outbound connections an attacker can initiate from a

honeypot. The purpose here is to count outbound connections, and when a certain limit has been met, block any more connections. This is primarily used to reduce the risk of mass scanning, attacking, or denial of service attacks, activity that requires many outbound connections. We use IPTables for this, which is configured and implemented by the rc.firewall script which comes with the HoneyNet CDROM. In IPTables, we set how many times an attacker can initiate a TCP, UDP, ICMP, or OTHER outbound connection. How many connections you allow depend on how much risk you are willing to assume. Limiting the number of outbound connections prevents attackers from using the honeynet to scan or attack large numbers of other systems, or to launch Denial of Service attacks. Its hard to do alot of damage when you are limited to the number of connections you can initiate outbound. However, keep in mind that this can also become a signature. An attacker may be able to detect your honeynet simply by initiating outbound connections, and seeing if they are blocked after a certain number. The default for connection limits in Honeywall CDROM is as follows. Note, the variable OTHER is any IP protocol that is NOT TCP, UDP, or ICMP (such as IPsec, IPv6 tunneling, Network Voice Proctol, etc).

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

This is how IPTables implements connection limiting. When an attacker breaks into a honeypot, they may initiate connections out of the network for a variety of reasons (download toolkits, setup automated bots, IRC chats, send emails, etc). Everytime time one of these connections is initiated outbound, the firewall counts them. When the limit is reached, IPTables blocks anymore connections from that honeypot. Then, IPTables resets itself, allowing as many connections outbound per time scale allowed. For example, lets say we set our TCP limit to 25 outbound connections per day. When the attacker breaks into our honeypot, they are allowed 25 outbound TCP connections. When they reach the limit of 25 TCP connections, they cannot start anymore. IPTables then resets itself allowing 25 more connections out over whatever the scale is, in this case 25 connections over the next 24 hours, limiting one connection per hour. If we had set the scale to hour, then once the limit was met, it would have allowed 25 connections over each successvie hour, or one connection every 2.4 minutes. To see a real world example of this behavior, refer to [these IPTable logs](#) of a Win2000 honeypot infected with the Code Red II worm, and its attempt to scan outbound. One nice feature of IPTables, when the TCP limit has been met, it does not effect any of the UDP, ICMP or OTHER traffic, until their limits have been met also.

Once you have implemented connection limiting, the next step is implementing NIPS functionality. Remember, the purpose of our NIPS is to identify and block known attacks. It does this by inspecting each packet as it travels through our gateway. If any packet matches any of the IDS rules, not only is an alert generated (like a traditional NIDS) but the packet can be dropped (blocking the attack) or modified (disabling the attack). The advantage is we dramatically reduce the risk of a known outbound attack being successful. The disadvantate is this works with only known attacks. In the case of rate limiting, we are allowing by default 15 outbound TCP connections a day. What happens if your honeynet gets infected with a worm, and in those first 15 outbound connections it attempts to infect other systems? While rate limiting has reduced the number of systems it can infect, you still have that

risk. The idea of an NIPS is it would block or disable any known attacks identified in those first 15 connections. For this, we use the [Snort_inline](#), functionality that has been integrated into the new version of [Snort](#).

For `snort_inline` to work as a NIPS (in gateway mode) it has to have something route the packets for it, `snort_inline` does not know how to act as a router (`ip_forward`). So, we have something route packets for `snort_inline`, and during the routing process the packet is given to `snort_inline` for analysis. Once `snort_inline` is done with the packet, it hands it back to the routing process. That process is IPTables. We configure IPTables to take packets it is forwarding, put them in user space for `snort_inline` to analyze, then IPTables continues to route the packet. This feature of IPTables is called user-space queuing, as it requires the `ip_queue` module to be loaded into the kernel. One thing to keep in mind when combining `snort_inline` and IPTables counting capabilities, IPTables will count the outbound connection regardless if `snort_inline` allows the packet through or blocks the packet. That is because all packets traverse through the internal interface first before being analyzed by `snort_inline`. The connection has been counted before `snort_inline` ever sees it.

If you enable `snort_inline` capabilities, then you you MUST have `snort_inline` running. If you do not have `snort_inline` running, or if for some reason `snort_inline` should die, then no packets will route through IPTables (this is a fail safe feature, not a bug :). As such, our next step is to configure `snort_inline`. This is very similar to `snort`, except it uses a different rulebase. Keep in mind, our goal is not to drop all outbound traffic, only attacks. So, we want to use a ruleset that only has actual attacks or exploits. You do not want to block outbound informational queries, such as ICMP ping, finger query, or a simple HTTP GET command. If we use ALL the Snort rules, then the attacker will once again not be able to do anything outbound. As such, we only use the Snort rules that define actual attacks. Each organization may have a different definition of what an attack is, so we recommend that you review and modify the `snort_inline` rules before using them. Also, our ruleset has to be reversed from a typical Snort rulebase, as these focus on inbound attacks. With `Snort_inline`, we want to focus on outbound attacks. The goal is to protect the outside world from the honeynet. Last, the rules are different, as we are not alerting on activity, but actually dropping or modifying attacks. A [drop ruleset](#) purpose is to analyze packets, and if it sees an outbound attack, to block or drop the packet containing the attack. You can see a specific example of this in the dropping of the [Code Red II](#) attack we mentioned earlier. A [replace ruleset](#) does not block attacks. Instead, it modifies the contents of the actual attack, disabling the exploit. This is potentially a more difficult control method for the attacker to detect. They will see their attacks reaching their intended targets, but not be able to figure out why the attacks are failing. The tool [snortconfig](#) is a flexible script that converts standard Snort rules to `snort-inline` capabilities, such as `drop`, `sdrop`, or `replace`. This tool is very important to maintaining a current `snort-inline` ruleset.

Data Capture

Once we have deployed Data Control, we can deploy Data Capture. The purpose of Data Capture is to log all of the attacker's activity. This is the whole purpose of the honeynet, to collect information. Without Data Capture, our honeynet has no value. The key to Data Capture is collecting information at as many layers as possible. No single layer tells us everything. For example, many people think all you need is the attacker's keystrokes, however this is not true. What happens when the attacker launches a tool, how will you know what the tool does if you do not capture the tool itself, or the network traffic? The Honeynet Project has identified three critical layers of Data Capture; firewall logs, network traffic,

and system activity. We will detail how to implement all three of these.

Firewall logs are very simple, we already did that part. By implementing IPTablesfirewall script, we are already logging all inbound and outbound connections to /var/log/messages. This information is critical, as its our first indication of what an attacker is doing. Its also our first warning when outbound attacks have been initiated. Based on our experience, firewall logs have proved critical in quickly identifying new or unknown behaviour. The Honeywall CDROM identified four different types of traffic; TCP, UDP, ICMP, and OTHER. Just as with Data Control, OTHER represents any non-IP proto 1, 6, or 17 type traffic. It also tends to be the most interesting, when someone is using non-standard IP traffic, they are most likely trying a new attack or method never published before (as in the backdoor channel seen in [Scan of the Month 22](#)).

The second element is capturing every packet and its full payload as it enters or leaves the honeynet. While the snort_inline process could conceivably do this, we don't want to put all of our eggs into one basket. Instead, we configure and run a second process to capture all of this activity. We do this by using a standard Snort process to capture all IP traffic, regardless of the IP type. In the startup script we bind the sniffer to the internal interface, eth1. This is critical. If by mistake you bind the sniffer to the external interface (eth0) not only will you log honeynet data, but all other traffic related to the external network. This will pollute the data you capture. By sniffing from the internal interface, you will capture only inbound and outbound honeynet traffic, which is exactly what you want.

The third element is the most challenging, capturing the attacker's activity on the honeypot itself. Years ago this was simple, as most remote interaction with systems was done over cleartext protocols, such as FTP, HTTP, or Telnet. You merely had to sniff the connections to capture keystrokes. However, attackers, just like you, have adopted encryption. Nowadays they are just as likely to use SSH or 3DES channels to communicate with compromised computers. We can no longer capture keystrokes off the wire, instead grab them from the system itself. One advantage to this system is that most encrypted is decrypted at the system end point, in our case the honeypot. If we could capture the data as its decrypted on the honeypot, we can bypass encrypted communications. [Sebek](#) is a tool designed to do just that. Sebek is a hidden kernel module (or patch) capable of logging attacker's activity. Once installed on a honeypot, the Sebek client runs in the kernel. The information gathered by the Sebek client is not stored on the honeypot where it can be discovered by the attacker. Instead, the Sebek client transmits its data via UDP to a sniffing machine (such as the honeywall gateway, or a remote logging system on another network). Attackers cannot see nor sniff these packets, as the Sebek client on the honeypots hide them. Even if attackers download or use their own sniffing tools, Sebek activity should be hidden from them. This is done by modifying the honeypot so it cannot see nor sniff any packets with a predesignated magic number and UDP port. Sebek then simply dumps the attacker's collected information on the wire, which are then captured by the gateway. Since all the honeypots are controlled with Sebek, none of them can be used to sniff keystrokes dumped on the wire. Note, if you have a honeypot that does not have Sebek installed, (or Sebek improperly configured) and an attacker takes control of that system, then they can now sniff Sebek packets coming from other systems as those packets are not hidden.

As Sebek runs in the kernel, it has to be compiled for the specific OS and kernel version of your honeypot. While each client version is different for different operating systems, they all come with the same configuration file. The purpose of the configuration file is to determine what information is

gathered, and how that information is dumped to the wire. By default, Sebek captures all activity on the system, however you have the option of capturing keystroke activity only. The combination of a magic number (Sebek specific) and the destination UDP port number determine what packets will be hidden. All honeypots in the same group must share the same combination to work effectively. Once configured, Sebek dumps all system activity to the network. These packets are then used to reconstruct the attacker's activities, and the impact it had on the system. To learn more about Sebek, refer to the paper [Know Your Enemy: Sebek](#).

Alerting

There is one last element you need to consider before finishing your honeynet, alerting. Having someone break into your honeynet is a great learning experience, unless you are unaware that someone has broken into it. Ensuring that you are notified to a compromise (and responding to it) are critical for a successful honeynet. Ideally you could have round-the-clock monitoring by a seasoned admin. However, for organizations that cannot support 24/7 staff, one alternative is automated alerting. One option for automated monitoring is [Swatch, the Simple Watcher](#). Swatch is an automated monitoring tool that is capable of alerting administrators of possible successful attacks on the honeynet. Swatch monitors log files for patterns described in a configuration file. When a pattern is found it can disseminate alerts via email, system bells, phone calls, and can be extended to run other commands/programs. A simple Swatch rule contains the pattern to watch for followed by a list of actions to take. By default Swatch will include in email alerts the line in the log file that matched the given rule. An example email for the above rule would look like the example below.

```
To:  admin@honeynet.org
From: yourdatacontrol@yourdomain.
org
Subject:  ----- ALERT!:  OUTBOUND CONN
-----
```

```
Apr  6 17:19:05 honeywall FIREWALL:OUTBOUND CONN UDP:
IN=br0
PHYSIN=eth1 OUT=br0 PHYSOUT=eth2 SRC=192.168.1.101
DST=63.107.222.112 LEN=123 TOS=0x00 PREC=0x00 TTL=255 ID=43147
PROTO=UDP SPT=5353 DPT=79 LEN=103
```

Even with the automated tools described in the Data Control section, an effective honeynet requires constant supervision. Properly configured, Swatch can be used to quickly notify administrators of events on their network. However, do not depend on outbound connections as your only source of alerting. For example, attacker may compromise the system, but never attempt an outbound connection. Be sure to monitor other sources of information, such as keystrokes collected by the Sebek clients. More advanced detection, reporting, and alerting mechanisms are under development for the Honeywall CDROM.

Testing

Once we have configured Data Control and Data Capture, the next step will be to test the gateway. To

test your deployment, below are some basics steps you can take. The Honeywall CDROM comes with a more thorough test plan, which can be found in the documentation section. To test the gateway, we will need a system on the external interface, we will call this the test system. Based on [Figure A](#), we will use the system 192.168.1.20 as our test system. We begin by first testing Data Control, does our honeynet successfully contain inbound and outbound activity?. First, initiate a connection from the test system to one of the honeypots within the honeynet. Based on your ruleset, this connection should have most likely been allowed. If so, you would have an entry similar to this in /var/log/iptables

```
Mar 23 20:55:09 honeywall kernel: INBOUND TCP: IN=br0 PHYSIN=eth0 OUT=br0
PHYSOUT=eth1 SRC=192.168.1.20 DST=192.168.1.101 LEN=60 TOS=0x00 PREC=0x00
TTL=63 ID=48699 DF PROTO=TCP SPT=36797 DPT=21 WINDOW=5840 RES=0x00 SYN
URGP=0
```

Once you have confirmed inbound connectivity is working, the next step is to test outbound. Begin by accessing one of the honeypots behind the gateway, (we recommend you access honeypots from the console, as the honeypot will log locally any remote connections, such as SSH) . From there, initiate multiple outbound connections to the test system. This will replicate one of the honeypots has been compromised, and an attacker is attempting to initiate outbound connections, and potentially an attack. The connections should be logged to /var/log/iptables on the Honeywall CDROM. In our case, we can attempt multiple outbound FTP connections to the test system on the production network. When our limit of 15 TCP connections is hit, a "Drop TCP" entry is logged. You would most likely have entries similar to this.

```
Mar 23 17:45:36 laptop kernel: OUTBOUND CONN TCP: IN=br0 PHYSIN=eth1
OUT=br0 PHYSOUT=eth0 SRC=192.168.1.101 DST=192.168.1.20 LEN=60 TOS=0x00
PREC=0x00 TTL=64 ID=36399 DF PROTO=TCP SPT=1026 DPT=80 WINDOW=5840 RES=0x00
SYN URGP=0
Mar 23 21:14:07 laptop kernel: Drop TCP after 15 attempts IN=br0
PHYSIN=eth1 OUT=br0 PHYSOUT=eth0 SRC=192.168.1.101 DST=192.168.1.20 LEN=60
TOS=0x00 PREC=0x00 TTL=64 ID=63391 DF PROTO=TCP SPT=1030 DPT=21 WINDOW=5840
RES=0x00 SYN URGP=0
```

Next, we will want to confirm that our NIPS technology, snort_inline is working. Fortunately, the snort-inline toolkit comes with test rules, designed specifically for test. Be sure to enable those rules before testing snort_inline. The rulebase you are running will also determine which test you run. If you are running a drop ruleset, you test by simply by first enabling the default test rule, restart snort_inline using the start script, then attempt an outbound Telnet connection. Snort_inline should detect, drop, and log the attempt. Your Telnet attempt should not work, it should simply time out. If you are running a replace ruleset, you test by enabling the default replace test rule, restart snort_inline, then attempt a simple HTTP GET command. Snort_inline should detect, modify, and log the attempt. To confirm the modification happens, be sure to sniff the EXTERNAL interface eth0. Do NOT sniff the internal interface eth1, as the GET command is not modified until after it passes through the internal interface, but before it leaves the external interface. Be sure that once you are done testing, you disable the rules (or the bad guys could simply run the default tests also).

```
03/23-21:21:05.915340 [**] [1:0:0] Dropping Telnet connection [**]  
[Priority: 0] {TCP} 192.168.1.101:39528 -> 192.168.1.20:23  
03/23-21:21:24.054533 [**] [1:0:0] Modifying HTTP GET command [**]  
[Priority: 0] {TCP} 192.168.1.101:38533 -> 192.168.1.20:80
```

Once we confirm Data Control we then want to ensure that Data Capture is working. Remember, if our honeynet is not logging all activity, then the honeynet has no value. We confirm Data Capture by looking at the logs, did the Honeynet capture the Data Control test we just ran? We begin with the firewall logs. This test is simple, all of our connections should have been logged to /var/log/iptables. Specifically, we should first see the inbound connections from the test system to the honeypot. Second, we should see the outbound connections from the honeypot to the test system. Last, we should see an alert message indicating that the outbound limit has been met, and all further connections will be dropped. We already tested this when we confirmed Data Control Next, we review the network logs. We want to be sure we captured every packet and the full payload of every connection, both inbound and outbound. Based on experience, we have found its best to rotate the logs on daily basis. The log we are primarily interested in is the binary log capture, called snort.log.*. In addition, you may find various directories made up IP addressess. These contain the any output of packets containing ASCII content, such as FTP commands or an .html page. You can confirm Snort logged all packets by analyzing the binary log file, as follows:

```
honeywall #snort -vdr snort.log.*
```

Finally we review the Sebek logs. These are the keystrokes captured by the Sebek kernel module, then dumped onto the network. These packets should have been captured by the network sniffer (Snort). Its highly recommend you use the GUI interface WAlleye that comes with the Honeywall CDROM for all Sebek analysis. If you were able to analyze the collected data, you Data Capture is working successfully! Also, check your email, you should have been alerted to the test you just ran. Swatch should have alerted you to your own tests that you just ran. Now, since you are a true security professional, we know you are going to reboot your honeynet gateway and test Data Control and Data Capture one more time, just to be sure. Also, we encourage organizations to do more extensive testing, as documented in the Honeywall CDROM docs section.

Conclusion

We have just completed an overview of how to build and deploy a GenII Honeynet based on a bridging gateway using Linux. This deployment represents some of the more advanced features of honeynet technology. However, keep in mind that information security is similar to an arms race, as we release new technologies to capture attacker activities, these very same threats can develop their own counter measures. If you are interested in deploying your own honeynet, we highly recommend the [Honeywall CDROM](#).

The Honeynet Project