A Novel Stealthy Data Capture Tool for Honeynet System

NGUYEN ANH QUYNH, YOSHIYASU TAKEFUJI
Graduate School of Media and Governance
Keio University
5322 Endoh, Fujisawa, 252-8520
JAPAN

Abstract: Data capture tool is one of the core components of a honeynet system. The most vital requirement of this component is: it must function as stealthily as possible, so the intruder is not aware of its presence. Currently Sebek is the most sophisticated tool for this purpose. Unfortunately Sebek is rather easy to detect, even with unprivileged right access. This paper presents a novel approach to improve Sebek on this aspect. We proposes a design and implementation of a tool named Xebek, which based on Xen technology, to fix the most outstanding problems of Sebek. Our experimental results prove that Xebek is much more covert, while the reliability and efficient are improved significantly.

Key-Words: Xen, stealthy communication, data capture tool, intrusion detection, security attack, honeynet

1 Introduction

Honeynet ([1], [2]) is a high-interaction type of honeypot [3] with the purpose: to gather information about threats. These collected information is used to better understand threats, how they are evolving and changing, in order to counter those threats in the best way possible. If applying honeynet properly, we can discover the novel attack patterns and unknown security holes. Honeynet also helps to study the attacker's motives.

The honeynet consists of 3 key components:

- Data control: this component is used to contain the intruder's activities and ensure that he does not cause any harm to other production systems outside the honeynet.
- Data capture: honeynet must capture all activities within the honeynet, together the information entered and left the system.
- Data collection: the gathered information got from the capture component must be securely and secretly forwarded to a central data server. This allows data captured from various honeynet sensors to be centrally collected for analysis and archiving.

Regarding data capture tool, Sebek [4] is the widely used in current honeynet technology. Sebek architecture consists of 2 key components: a kernel module run on honeypot system, and a central server to collect data. The first component, Sebek kernel module, can capture intruder's activities and transfer the collected data to a Sebek server (sebekd) run on a central machine, and the analyzing process will be taken there with some utilities provided with Sebek package.

One of the vital requirements of the data capture component is: it must be as covert as possible, so the intruder never knows that he is under watch eye. To satisfy that demand, Sebek applies many tricks borrowed from the black-hat community. Unfortunately those tricks are not enough to cover Sebek: many methods can be used to detect Sebek's presence, and some of them are not even required privileged access right.

Besides, sebekd server must expose to the network to capture data forwarded from honeynet, and this can causes another headache: this server must be protected at all cost, or the attacker will bring it down, then he can do anything he likes to the honeynet he broken it without worrying that he is being observed.

This paper proposes a novel approaches to eliminate the above problems of Sebek. We will present a new architecture together a prototype implementation of a new data capture tool named Xebek on Xen Virtual Machine environment. While Xebek still has the capture ability of Sebek, it is much more stealthy, far more reliable and the efficiency also significantly increases.

This paper consists of 6 parts: the next part discusses in detail the drawbacks of Sebek, while the 3rd part outlines the architecture and design of Xebek and points out the advantage of this approach, then discusses in detail about Xebek implementation. Some necessary steps to deploy Xebek are explained.
6. Sebek sends the captured data to the central server

4. Sebek replaces some systemcalls with its own

2. Though Sebek tries to patch the network stack to

5. After detecting Sebek, the intruder can remove it

7. The central server must expose to the network to

1. Unprivileged local user can run a simple command

Unfortunately while many efforts are put on Sebek to
make it more covert, Sebek is still rather easy to
uncover and defeat. Here are some of the most
noticeable methods:

1. Unprivileged local user can run a simple command
that generates big input or output, such as the dd-
attack method proposed by M.Dornseif et al [6], then
check to see if the network performance is decreased
with ping command. The reason is data captured from
all the input and output are forwarded to the central
Sebekd server via network stack, and this activity
may lead to congest network stack if the data is
enormous

2. Though Sebek tries to patch the network stack to
hide the correct network statistics when using
network layer to send data, we can still compare the
internal statistics got from inside the kernel with what
the system shows to userspace [6]. The difference
will disclose the presence of Sebek.

3. Sebek is a kernel module inserted into the system,
so it is listed in the kernel module list (with lsmod
command on Linux). Though we can try to hide it
with another kernel module (like the clean method
proposed by adore-ng [7]), Sebek module can still be
found with a brute-force scanning technique [8]

4. Sebek replaces some systemcalls with its own
functions. This lead to another way to discover
Sebek: we just need to check to see if the address of
these system calls is are abnormal places in the
memory. If that is the case, chances are Sebek is
present in the kernel.

5. After detecting Sebek, the intruder can remove it
easily by recovering the original system call (see
unsebek.c [9]). The fact that Sebek is a kernel module
makes it easier to do that.

6. Sebek sends the captured data to the central server
via network. If the intruder has a sniffer (such as
tcpdump [10]) installed at the right place, he will see
those data and easily figure out that his penetrated
system is a honeynet.

7. The central server must expose to the network to
receive data sent from the honeynet. That will tempt
the intruder to attack this server to bring down the
fundamental component of our honeynet. This is not
such a method, in which sebekd will be taken over if
it uses a libpcap library with buffer overflow bug.

As we see, there are too much problems with the
current Sebek, and they all make honeynet less
attractive solution for security practices.

3 Xebek Solution

This part presents Xebek solution, which can replace
Sebek as an effective data capture tool, while it can
eliminate many problems of Sebek. Because Xebek is
made to work in Xen environment, we will first take
a brief look at Xen technology, and then discuss more
about Xebek architecture and implementation.

3.1 Xen Virtual Machine

Xen [12, 13] is a virtual machine monitor initially
developed by the University of Cambridge Computer
Laboratory and now promoted by various industrial
monsters like Intel, AMD, IBM, HP, RedHat, Novel
and by the whole open source community. Being
released under the open source GNU GPL license,
Xen can be used to partition a machine to support the
concurrent execution of multiple operating systems
(OS). Commodity OS (now officially Linux, FreeBSD, NetBSD are supported) can run on Xen
with small changes to the kernel. Xen is outstanding
because the performance overhead introduced by
virtualization is negligible: the slowdown is around
only 3% [14]. Various practices take the advantages
offered by Xen, such as server consolidation, co-
located hosting facilities, distributed services and
application mobility.

Xen community is working hard to gradually push
Xen into Linux kernel, so it will be available for
every Linux users. The process is expected to start
from kernel 2.6.15.

3.2 Xebek Design

Goals and Approaches: Xebek is designed with the
aim to overcome the problems posed by Sebek in
honeynet environment.

1. The first goal of Xebek is to capture data as Sebek
does on honeynet system. In our Xen diagram the
honeynet system runs on a DomU, and all the
activities happened inside this domain must be
captured: this includes keystrokes, input and output
from file system and socket. To do that, Xebek
employs the same techniques as Sebek does by
modifying kernel system calls. But while Sebek
works as a module, we propose Xebek as kernel
patch, so we do not need to worry about hiding kernel
module as Sebek does, and it is also more difficult for intruder to remove Xebek from kernel.

2. Another mission for Xebek is to eliminate the problem of leaving many traces while sending data to through the network stack with Sebek. To solve this trouble, Xebek is designed so all the data is forwarded to the central server via shared memory, and it never uses network stack like Sebek does. Consequently the intruder cannot detect the data by looking at network traffic like he can with Sebek. One more advantage of this approach is: data is sent via shared memory, so the reliability and efficiency is significantly increased.

3. One more target is to protect the central server against the possible attack from outside. Regarding this issue, we put the central server (called xebekd) on Domain-0 (Dom0), and this server will get all the data sent out from user domain (DomU) via shared memory. Because we no longer use network to deliver data, xebekd is not necessarily exposed to the network like sebekd does.

4. To make Xebek an attractive option to practical and research community, it is a good idea to make the output logging data and add-in tools compatible with Sebek as much as possible. This will help people familiar with Sebek to switch to Xebek.

5. The final goal is Xebek must be flexible, so the administrator can disable or enable it as he desires at run-time.

All of those goals and approaches lead us to the architecture for Xebek as followings.

**Architecture:** Xebek consists of 4 main components: data capture tool in DomU (xebekU), data receiver in Dom0 (xebek0), data collection daemon (xebekd) and analyzing utilities. (See figure 1)

**xebekU:** xebekU is a kernel code in kernel of DomU. This code patches the system calls (such as open, close, read, write, socket,...) to gather the data coming in and out of system. The collected data are then forwarded to xebek0 via a shared memory between DomU and Dom0. At run-time, the administrator can choose to enable or disable xebekU with an instruction sent from Dom0's user-space.

**xebek0:** xebek0 is a kernel code in kernel-space of Dom0. xebek0 waits for and gathers data sent from xebekU. xebek0 and xebekU share a memory area and this memory is used to exchange the collection data. Besides, xebek0 registers a software device (at /dev/xebek) and sends data records to the collection daemon xebekd in Dom0's user-space via this device.

**xebekd:** this daemon process runs in user-space of Dom0, and records the data sent from xebek0 put in a device mentioned above. The recording data is separated for each domain in a logging directory.

**Add-on utilities:** Xebek has some utilities to extract interested data from the logging files of xebekd. We intend to provide what Sebek provides with Sebek package, it is easier for people to adopt Xebek. For the time being, a tool to extract keystrokes from logging data and another tool to upload data to a SQL server are available.

### 3.2 Xebek Implementation

At the moment Xebek is implemented only in Linux. The reason is other OSes (like FreeBSD and NetBSD) are not ready for Xen 3.0, the most advanced Xen version we are working on, yet. So in this part we will present Xebek for Linux environment. The same techniques can be applied for others, however.

**xebekU:** xebekU is the kernel code run in DomU. One of the important jobs of xebekU is to gather the data from I/O systemcalls such as open, close, read, write, socket.... To do that these systemcalls in DomU's kernel is modified, so the patched systemcalls deliver their data to xebekU. With each of these systemcalls, we define corresponding type, and the type is recorded with the logging data, so we can distinguish these data when analyzing them later. Some of the types are: OPEN, CLOSE, READ, WRITE (for sys_open, sys_close, sys_read, sys_write,...). Those records will be saved in a structure of xebek packet type (see figure 2), in which we save also the owner's uid, process ID and inode number of the corresponding file. The actual data will follow the packet. This format is compatible with Sebek logging format, and that is one of our
important targets.

```
struct xebek_packet {
  u32 magic: /* magic value of packet*/
  u16 version: /* xebek's version */
  uid_t uid: /* tty's owner */
  pid_t pid: /* process id */
  duint16_t length: /* payload size */
  long inode: /* file's inode */
  struct timeval time: /* time of event */
} __attribute__((packed));
```

Figure 2: xebek_packet structure

As DomU and Dom0 run on the same physical machine, xebekU and xebek0 can share memory with each other. When xebekU initializes, it allocates some memory for sharing (the amount of shared memory is configurable at runtime - by default is 4 page, which is equivalent to 16KB on x86 systems), and grants those memory to Dom0 by using Xen grant reference API ([15]). This shared memory will be used to store the logging data fetched from the above system calls.

To communicate with xebek0, xebekU assigns an event-channel port to send notifications to xebek0. After that, xebekU informs xebek0 the value of grant reference got in the above step, together with the event-channel port. At this moment, the event-channel is not established yet, so xebekU writes these information to xenstore via xenbus interface.

At run-time, xebekU puts the gathered logging data into the shared memory, then notifies xebek0 via the event-channel about the newly-arrived data. xebek0 would be awakened from the possible sleep and reads the data out, then updates the internal share memory structure respectively. (More about the structure of shared memory will be discussed later)

**xebek0:** In Dom0, when initializing xebek0 registers a xenbus watch to listen for change to xenstore. When it detects the new notifications written to xenstore by xebekU, xebek0 will map the shared memory reference granted by xebekU. Subsequently, it allocates an event-channel port corresponding to the event-channel port of xebekU. Finally, xebek0 binds its event-channel to an irq handler, so it can handle the notification about the new logging data dispatched from xebekU. From then on, xebekU and xebek0 can contact through the established event-channel.

Another job xebek0 must do when initializing is to register a misc device (this device locates at /dev/xebek). Whenever xebek0 gets the notification from xebekU, it wakes up and gets the data from the share memory, then puts these data into its internal buffer of the device. The size of this buffer is also configurable at boot time, which is 8 pages by default (equivalent to 32KB on x86).

To distinguish the logging data from different domains, xebek0 puts the logging data into a C structure named device_packet. (See figure 3) This structure will save domain id, so later xebekd can figure out which DomU sent this message. Together with domid, the length of message is also stored. Other fields are taken from xebek_packet structure. The actual logging data is appended at the end of the structure, and everything is put into the buffer.

```
struct device_packet {
  domid_t domid: /* who sent this log? */
  u32 magic: /* magic value of packet*/
  u16 version: /* xebek's version */
  uid_t uid: /* tty's owner */
  pid_t pid: /* process id */
  duint16_t length: /* payload size */
  long inode: /* file's inode */
  struct timeval time: /* time of event */
  char buf[0]: /* actual payload */
} __attribute__((packed));
```

Figure 3: device_packet structure

When receiving the request for data from userspace (xebekd in particular) via the device /dev/xebek, logging data will be extracted out from this internal buffer and sent to userspace.

**Shared memory structure and xebek0's internal buffer:** Since Xebek is designed to collect logging data from some systemcalls, chances are the incoming data is so big that Xebek cannot handle the data fast enough. Though it is favorable to give Xebek a big buffer for its share memory and internal buffer, too much data arrives at the same time are what we must take into account.

Another difficulty is: the shared buffer must be read and written at the same time by xebekU and xebek0. These conflicted activities can causes the race issues.

Those troubles direct us to the decision: the shared buffer should be designed as a ring buffer. Ring buffer is special data structure which has 2 heads: one for reading and one for writing, and these heads can wrap-around when they reach the end of the buffer. Writing data to buffer will take away some spaces, but reading from the buffer will release some spaces, and the free space then can be used for another writing request later. Figure 4 below declares the ring buffer structure.
The internal buffer of xebek0's device also uses the same data structure, so at the same time it can be written to with data from shared memory, and read by user space's request from xebekd.

**xebekd**: xebekd is the user-space daemon in Xebek architecture: its job is to gather logging data from the device put at /dev/xbek. While all other codes are written in C, this daemon is written in Python language. With Python, we can easily connect to and get internal information of xend, such as domain name.

At run-time, xebekd repeatedly queries /dev/xbek for new arriving data. As new data shows up, xebekd extracts information according to device_packet, figures out how much logging data appended after this record thanks to the length field. Subsequently xebekd attempts to read exactly that amount of data. Then it converts the domain ID in device_packet structure to domain name (thanks to the exported information of xend), and open a corresponding log file to add the logging data to the tail of that file.

To be more flexible, xebekd has one option to enable or disable xebekU of a certain domain, so the administrator can start or stop TTY logging data from any domain any time he desires.

**Add-on utilities**: Similarly to Sebek, Xebek provides some tools to extract data out from the logging files output from xebekd. For the time being we provides 2 kits: xebek_key is the tool to extract keystroke from the logging data, and xebek_upload to upload the data to a SQL server for analyzing later.

These tools are all written in Python language and easy to customize.

## 4 Deploying Xebek

Deploy Xebek in production environment is simple. In the following parts we will see the necessary steps, and discuss the methods to harden Xebek, so intruder has tough time to detect its presence.

### 4.1 Deploying Xebek

Following the below steps to deploy Xebek:

- Patch DomU's kernel with xebekU patch provided with Xebek package.
- Patch Dom0's kernel with xebek0 patch provided with Xebek package.
- Recompile Dom0 and DomU's kernels after patching in the above steps.
- Install xebekd in Dom0 and let it run at boot up.

In order to help users to troubleshoot the possible problems, kernel patches can be installed with DEBUG option. xebekd can also run in DEBUG mode, and the debug output will be placed in /var/log/xbek directory.

### 4.2 Bastile Xebek

Xebek provides patch for DomU in xebekU and this code are applied on DomU's kernel as built-in, so it is not shown in the kernel module list as Sebek might be, and consequently we cut down one chance for the intruder to detect Xebek's existence. This approach also makes it harder to remove Xebek from the memory if the intruder wants to do that.

There might be one more place the intruder can investigate to discover Xebek's presence: kernel binary and kernel symbol files. Fortunately, in Xen architecture DomU is run by loading the kernel from Dom0, so we will not need to have kernel binary file, together with kernel symbol files in DomU's file system. That nice feature would cut down one more chance to probe Xebek.

Last but not least, all the access to kernel memory should be prohibited, so the DomU's kernel should be compiled with /dev/{kmem,mem,port} removed [16], and the ability of loading kernel module at run-time should be eliminated, too. This can lead to some objections: the honeynet becomes too restrictive, and the attacker might suspect. But we argue that this kind of harden environment is increasingly popular, and it should be expected by the attacker on any production systems.

## 5 Related works

Honeynet is one of the hottest topics on security research. Many papers focus on applying honeynet to improve defense system or trap malwares. But there are few attempts to point out the weak points of honeynet or the method to improve Sebek, which are related to topic of this paper.

In [9] and [11], J.Corey points out some problems with honeynet, especially with Sebek, and several methods were proposed to defeat it.
M. Dornseif et all also presents few other methods to detect and exploit Sebek in [6].

As we are aware, Xebek is the first project to bring up the idea of employing Xen for honeynet research.

6 Conclusion

This paper proposes the design and implementation of Xebek solution to eliminate some problems of Sebek, a data capture tool in use in honeynet technology. We demonstrated that Xebek can be used instead of Sebek in Xen environment, and if being installed in a strict manner, Xebek is stealthier, harder to detect, even with privileged user. Moreover, Xebek is more flexible, far more effective and reliable.

At the moment Xebek works for Linux-based DomU. We plan to provide support for other Oses such as FreeBSD, NetBSD once these ports are working stably on Xen.

Availability: Xebek and related utilities are released under open-source license GNU GPL at authors' website [17]

References:
[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebar, Ian Pratt, Andrew Warfield, Xen and the art of virtualization, ACM Symposium on Operating Systems Principles. October 2003
[14] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne and Jeanna Neefe Matthews, Xen and the art of repeated research, Freenix 2004