BAIT-TRAP: a Catering Honeypot Framework

Xuxian Jiang, Dongyan Xu
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{*jiangx, dxu*}@cs.purdue.edu

Abstract

The honeypot has been proved effective in understanding intruders' tactics and tools which exploit system or software vulnerabilities. However, most current honeypots are manually *and* statically *composed and deployed, leading to the following disadvantages: (1) It only exhibits a small and fixed* spatial *vulnerability window in terms of number and variety of vulnerable services; (2) It ignores current network activities and can only provide information on threats to deployed services. New vulnerabilities in a service not deployed in the honeypot will remain undetected.*

To address the limitations, this paper proposes the notion of catering honeypots *and presents a catering honeypot architecture called BAIT-TRAP. The catering honeypot is a honeypot architecture that constantly monitors network traffic, identifies "bait" services that are currently attractive to intruders, and dynamically creates honeypots running such services in the hope of quickly trapping the subsequent exploitations. To the best of our knowledge, this is the first proposal and implementation of catering honeypots. Our real-world deployment of BAIT-TRAP has captured a number of "trendy" attack incidents, demonstrating the timeliness and trend awareness of catering honeypots.*

1 Introduction

Security threats and worm outbreaks are growing in both number and severity [13, 18, 3, 19, 16] in recent years. Meanwhile, researchers are actively developing new security systems and tools to detect, analyze, and prevent network attacks. As demonstrated in practice, the honeypot [7] is effective in understanding intruders' tactics and tools which compromise system or service softwares . Using unused IP addresses, honeypots have a fundamental advantage: every incoming/outgoing packet is suspicious.

However, most current honeypots are *manually* and *statically* composed and deployed, leading to the following limitations:

- A *static* honeypot usually exhibits relatively small and static spatial vulnerability window, because only a few pre-determined vulnerable services are installed and running. Only threats to deployed services will be exposed while vulnerabilities in many other services are left undetected.
- A static honeypot is unaware of current attack trends, lacking the capabilities of "attracting" some of the imminent attacks - for example, by sensing the network traffic and deploying new "bait" services on the fly.
- It is either impossible or ineffective to include a large number of services in one static "bloated" honeypot. System log noise and interference among the many services will immediately make it hard to extract and analyze useful attack information from the honeypot log.

It is desirable that the right honeypot with one or a few "attractive" vulnerable services be automatically created at the *right* time, namely when they are likely to be probed or compromised. To realize this vision, we present a novel concept called *catering honeypots*: Instead of blindly waiting for attacks, the catering honeypot will actively glean information from current network traffic and create honeypots that are likely to be probed or attacked within next few hours, minutes, or even seconds. Catering honeypot can be applied in many environments: An anti-virus software company may deploy it to capture the largest spectrum of Internet worms. An enterprise may install it to discover new vulnerabilities associated with the applications running in the enterprise domain.

It is challenging to realize an efficient catering honeypot architecture since it needs to sense and profile network traffic, identify or predict "bait" services, and automatically instantiate a "just-in-time" target honeypot.

In this paper, we propose the design and implementation of BAIT-TRAP, a catering honeypot architecture. By carefully monitoring network activities, BAIT-TRAP dynamically identifies "bait" services and automatically composes "attractive" honeypots in order to capture the expected attacks. Within seconds, a newly composed

honeypot will be automatically deployed and exposed to potential attackers. To the best of our knowledge, we are the first to propose and implement the notion of catering honeypots.

The rest of this paper is organized as follows: Section 2 describes background information about conventional honeypots and presents the vision of catering honeypot. The next section presents the design of BAIT-TRAP, a prototype of catering honeypots, while the implementation details are described in Section 4. Section 5 evaluates BAIT-TRAP's efficacy and practicality by describing several real-world attack incidents captured by actual deployment. Related work is presented in Section 6. Finally, we conclude this paper in Section 7.

2 Honeypots and Catering Honeypots

Lance Spitzner pioneered the work on honeypots and define a honeypot as the "security resource whose value lies in being probed, attacked, or compromised" [29]. Any actual computer system, emulated services, and even honeytokens [31] could assume the role of security resource. In this paper, we will focus on actual computer systems as honeypots.

There are different criteria to classify honeypots. One categorization is based on the fact whether honeypot services are emulated or not. A *physical honeypot* is an actual computer system providing real services while a *virtual honeypot* is emulated by another machine that responds to network traffic sent to the virtual honeypot. As an example, a physical RedHat 8.0 Linux box with Sendmail services could be a physical honeypot; a virtual machine providing the Sendmail services could be a virtual honeypot. Recent advances in virtual machine techniques [8, 22] have prospered the development of virtual honeypots. For example, VMware [8] not only provides an authentic environment emulating a physical machine, but also supplies additional features like system image snapshot and untamperable logging, which prove to be extremely helpful for post-mortem forensic analysis.

Another classification criteria is based on the level of interaction with intruders. The honeypots can be categorized into *high-interaction* honeypots, *medium-interaction* honeypots, and *low-interaction* honeypots. Associated with highest level of risk, high-interaction honeypots give attackers direct access to a real operating system where nothing is emulated or restricted. However, great risk comes with great value. High-interaction honeypots have been used to discover new attack tools and vulnerabilities in operating systems and applications, like the buffer overflow vulnerability in Solaris systems running the CDE Subprocess Control Service [10]. Medium-interaction honeypots provide less flexibilities and more restrictions to interact than high-interaction honeypots, but more functionalities than low-interaction honeypots. One example is the use of *jail* or *chroot* in a UNIX environment. Low-interaction honeypots provide the least functionalities, but are the easiest to install, configure, deploy, and maintain. It can emulate a variety of services, with which intruders are limited to interact. *honeyd* is one such low-interaction virtual honeypot framework [27].

Based on the level of dynamics or adaptability, honeypots can be also divided into *static* honeypots and *dynamic* honeypots. A static honeypot is a honeypot which always exhibits the same appearance or *personality* to outside while a dynamic honeypot can present different *personalities* based on various intentions of attempts. For example, an unmodified FreeBSD 5.0 system only running Apache web service could be a static honeypot. Catering honeypots, discussed in this paper, belong to the category of dynamic honeypots.

2.1 Catering Honeypots: Vision and Challenges

Catering honeypots share the same goal as conventional honeypots and have similar requirements for information collection and protection: all traffic related to a catering honeypot should be recorded, and the honeypot should be avoided for other unintentional purposes like in stepping stone attacks. Catering honeypots can span the spectrum of high-interaction, medium-interaction, and low-interaction honeypot categories, and can be either physical or virtual honeypots. However, the notion of catering honeypots can be differentiated from current honeypots in following ways:

- Firstly, a catering honeypot architecture *senses* and *infers* current network traffic. Histogram of access patterns exposes services or resources[1] of interests to intruders. Additionally, network conditions can be continuously profiled and applied to tune or refine the timely selection of a target service for honeypot purpose so that imminent attacks could be detected and understood.

- Secondly, a catering honeypot reacts to current needs by *automatically* composing and deploying a target honeypot with services of interest. Automation is necessary for avoiding the time-consuming process in manual service composition and deployment.

Figure 1 shows the vision of catering honeypots behind which there are two control logics: *activation logic* and *deactivation logic*. Activation logic inspects current network traffic for any abnormality and decides the right timing and suitable services to create catering honeypots. Deactivation logic closely monitors created honeypots and decides when to disconnect or recycle them. Activation logic is driven by

[1] A networked machine with idle CPU cycles could be an attractive resource to particular intruders.

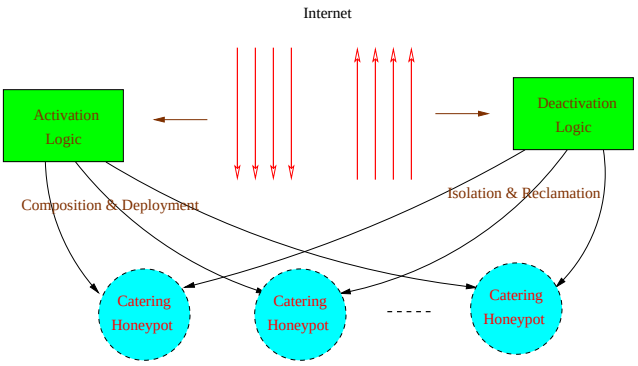


Figure 1. Vision of Catering Honeypots

temporal and *spatial* knowledge on traffic histogram as well as the traffic characteristics of different services. Based on various intentions, a set of "bait" services are selected for the creation of target honeypots. Being highly suspicious, every communicating packet with the catering honeypot needs to be recorded for later analysis. Furthermore, it is necessary to recycle or isolate the honeypot if (1) there is no more associated activities with the honeypot; or (2) the honeypot has been compromised and other unintended operations like DoS attacks have been performed.

In order to realize the vision of catering honeypot, the following questions need to be answered:

- *When and what to deploy?* Which vulnerabilities are of high interest and when will the corresponding honeypots be deployed? The requirement is specific to different scenarios and could vary based on different intentions of deploying these honeypots and different sets of interested services. Anti-virus companies like Symantec would be interested to capture any emerging worms no matter what kind of vulnerable services are exploited. So any probing with services containing recently discovered vulnerabilities could trigger a honeypot for them. A software vendor, like Rhino Software, Inc, would be mostly interested in any threats to offered softwares including Serv-U [2]. The chance is high to capture and understand potential exploitations by initiating a Serv-U honeypot once certain suspicious inputs like abnormal format string are observed.

- *How to compose?* When the decision is made to deploy a certain honeypot containing specific vulnerable services, how to compose the honeypot so that the service(s) will run properly? Based on the large number of vulnerable services, it is impossible to create a huge collection of honeypots which contains all possible vulnerable services over different OS platforms. An on-demand and automatic composition function is needed.

- *How to deploy?* Once the honeypot is composed, how to deploy it in such way that it would be immediately exposed to outside. It is desirable to have the capability of deploying honeypots within a short period of time so that it will not miss forthcoming attacks. However, the deployment technique could highly depend on the types of honeypots: a low-interaction virtual honeypot could be started within seconds, while a high-interaction physical honeypot could take as longer as several minutes to bootstrap.

- *When to deactivate?* Honeypots needs to be closely watched and extra care is necessary for keeping track of any activities involved with honeypots. It is better to isolate the honeypot once it is found participating a DDoS attack on some well-know sites or actively propagating worms.

In the following, this paper presents the design, implementation, and evaluation of a catering honeypot framework called BAIT-TRAP.

3 BAIT-TRAP Approach

Figure 2 shows the architectural view of BAIT-TRAP. BAIT-TRAP hosts and manages a local and dedicated darknet space[2] in which high-interaction honeypots are deployed. In order to meet different intentions of deployment, BAIT-TRAP maintains a complete list of candidate services. Inside BAIT-TRAP, a *profiler* module captures any activity in that darknet space and presents the results to the *activation/deactivation logic* modules. The reason for choosing darknet is to discern intrusions more easily and avoid potential disturbance to other production networks. However, the traffic for profiling could be collected from production networks. Based on the services of interests and current profiled results, the activation logic module would select the most interested service(s) and ask *honeypot manager* to compose and deploy a honeypot with chosen service(s). A complete deployment of all interested services will not only unnecessarily incur high management overhead and low resource utilization (including IP), but also introduce additional security risks and distraction on forensic analysis. Deactivation module, instead, will closely monitor deployed honeypots and ask honeypot manager to disconnect them if necessary.

Denote the set of services of potential interest as $S =< s_1, s_2, ..., s_n >$. n is usually very large (hundreds or thousands). We represent the incoming probings as p_j ($j \geq 1$) and corresponding time clock as t_{p_j}. The profiler module reports the type and number of attempts $A(s_i)$ for each candidate service during the last Δ period of time

[2] A darknet is a portion of routeable, allocated IP space in which no active services or servers reside.

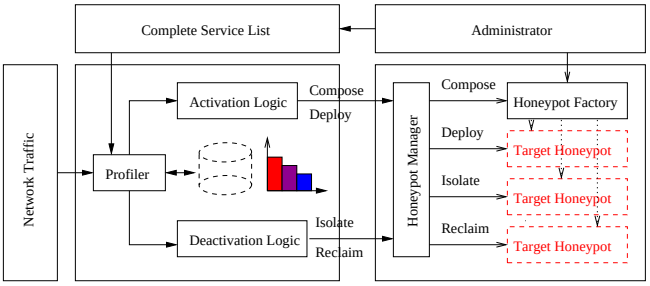


Figure 2. Architectural View of BAIT-TRAP

to the activation logic. The period Δ defines the size of a *sliding* window to observe network activities. The activation logic will select a target service during this time window according to a configured selection strategy. For example, the following simple algorithm selects the service with largest number of attempts during last Δ time.

SERVICEWITHMAXATTEMPTS()

1 $now \leftarrow$ current time clock
2 $P \leftarrow \{p_j : j \geq 1 \bigwedge t_{p_j} \geq now - \Delta\}$
3 $A_{s_i} \leftarrow 0$, for any $i \in \{1..n\}$
4 **while** $P \neq$ **NULL**
5 **do** Select a p_j from P
6 $P \leftarrow P - \{p_j\}$
7 **if** There exists a service s_i w.r.t p_j
8 **then** $A_{s_i} + +$
9
10 Select s_k such that $A(s_k) = \max_i(A(s_i))$
11 **return** s_k

Generally, the mapping from probing traffic to responsible services is straightforward. For example, it is highly likely that a TCP SYN packet on port 80 of a honeypot will attempt to exploit vulnerabilities in the HTTP daemon. Based on these probings or attempts, a "bait" service will be chosen and honeypot manager can be asked to compose or deploy a target honeypot with that service. Honeypot manager contains a set of honeypot images, which have been generated so far. If the intended honeypot is not in the set, honeypot manager will send a request to honeypot factory (described in Section 4.1) asking for a specific honeypot image with required service. Since the honeypots in BAIT-TRAP are high-interaction ones, the composed images contain not only related services, but also the underlying operating systems like Linux, Windows, or FreeBSD.

Once the honeypot image is available, it will be deployed automatically. Physical honeypots require on-demand priming of raw machines with particular images, while virtual honeypots could be instantiated with virtualized resources. A physical honeypot provides a completely native environment to interact with intruders. However, it occupies a dedicated physical node and takes longer, like minutes, to generate and deploy. A virtual honeypot consumes less resources and could take only tens of seconds from the time a request is received to the time a target honeypot is deployed. In current prototype and deployment of BAIT-TRAP, a standard RedHat 7.2 Linux on top of VMWare[8] can be instantiated within 30 seconds, while a customized Linux image with only 32 MBytes[20] could take almost 1 minute to deploy. More details will be described in Section 4.2.

It is worth noting that honeypots are created with vulnerable services and these honeypots can be taken full control by remote intruders. The deactivation logic needs to meticulously check any activities initiated from honeypots and take precautions in mitigating and even isolating potential threats. It has been found effective in practice to (1) limit concurrent outgoing TCP connections; (2) restrict relevant traffic volume; or (3) drop packets containing well-known attack signatures. If these anomalies are detected, the honeypots need to be taken offline for forensic analysis. Also, it is possible that some honeypots running one vulnerable service can be recycled for another vulnerable service.

4 Implementation

In this section, we describe the implementation of BAIT-TRAP. The profiler module is implemented as a packet sniffer with added feature to classify the probing according to the type of interested service types. The activation logic module implements a simple algorithm selecting services with the highest percentage of attempts. The deactivation logic module monitors any abnormality from the honeypots, and disconnects them if observed. In the following, we will focus on the ways how these honeypots are dynamically composed and deployed and how associated security risks are mitigated.

4.1 Honeypot Composition

Honeypot factory receives the requests from honeypot manager for a target honeypot with a particular service. A specific operating system is also needed to host the service. There are two levels of composition in honeypot factory for generating a target honeypot: *kernel level* and *service level*. Different honeypots might require different types of operating systems. With the same type of operating system, a specific version may be required. Even with the same type and the same version of operating system, different honeypots may still have different types of services and different versions for the same type. Kernel-level composition is more coarse-grained when compared to service level and service-level composition requires the ability to accommodate system services on need basis. In current prototype, BAIT-TRAP only supports open-source Linux as the operating system when dynamic service-level composition is needed. Other systems like a IIS/Windows

honeypot are generated offline and directly dispatched by honeypot manager.

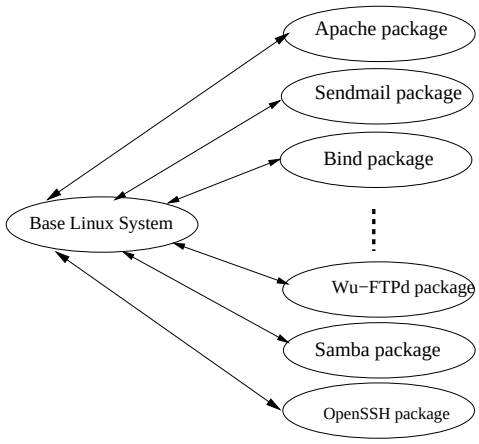


Figure 3. Honeypot Composition

Considering dynamic service-level composition with one particular version of Linux kernel, honeypot factory creates a *package* for each service type and service version. During each package definition, we identify the dependency on services, libraries, or even particular configuration files. When the package is needed, other required services, libraries, and particular configuration files will be automatically included in the target honeypot. If two services are in conflict or incompatible with each other, then the two services can not exist in the same target honeypot. BAIT-TRAP uses *pacman* [32] for managing and updating packages.

Figure 3 illustrates six optional service packages. The *Apache* package contains all files such as binaries, libraries and configuration files related to a particular version of Apache, while the *Sendmail* package includes all related files associated with the e-mail agent. The *Bind* package is related to name resolving services. Some particular service like Wu-FTPd, Samba, or OpenSSH can also be an independent package. For each package included in the target honeypot, a service-specific startup script is linked into */etc/rc.d/rc3.d*. Thus the system will automatically start the required services once bootstrapped.

It should be noted that since these honeypots are likely to be controlled by intruders. It is necessary to plant certain sensors which is able to understand during forensic analysis the activities performed by intruders and capture the tools used by them. Particularly, it is desirable to record any keystrokes of intruders without their awarenesses. For this purpose, a kernel-level data capture tool - Sebek [4] is installed as default in every composed honeypot image. In Section 5, we will further show the intruders' activities in several real-world incidents.

Current prototype can automate the process of dynamic service composition within 10 seconds. With current price and availability of storage, it is possible to create several popular target honeypot images offline. However, the ability of dynamic service composition is still desirable.

4.2 Honeypot Deployment

Once a right target honeypot is generated, it needs efficient ways to deploy it. However, different types of honeypots need different mechanisms for deployment. A physical honeypot requires the dedicated usage of one physical machine while several virtual honeypots could share one physical node at the same time.

Physical Honeypots Figure 4 shows the diagram in actually deploying a high-interaction physical honeypot. The whole process is based on standard BOOTP/TFTP protocols:

- When a raw node is rebooted, one particular bootstrapping code takes control of the raw node and send BOOTP requests for local domain information (gateway and assigned IP addresses) and system image information (image name and image server IP address).
- Once required information is obtained, the instructed image is downloaded and executed. This image is another special trampoline code based on Xinu[20], which will attempt to fetch a target honeypot image.
- Successfully downloaded honeypot image contains its own bootstrapping code which will take over the physical node. Finally, hosted (vulnerable) service will be automatically started and exposed for public access.

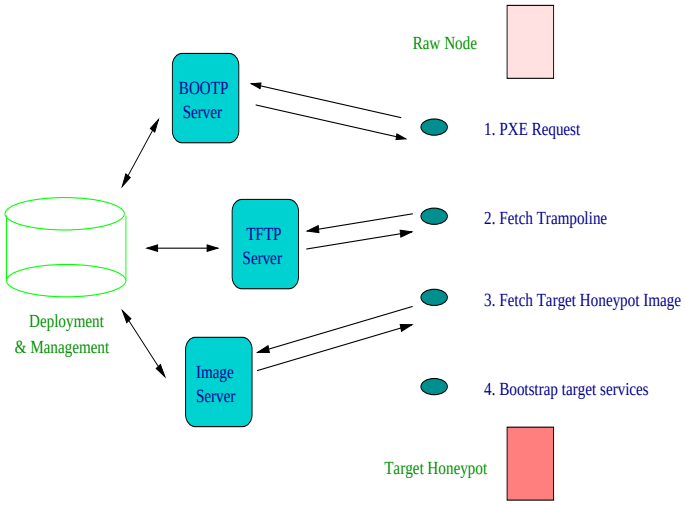


Figure 4. Honeypot Deployment

There still remains one question: how to automatically reclaim back those physical nodes after they are fully compromised? BAIT-TRAP leverages a special power unit[21],

which is connected to every physical node. That power unit can be remotely controlled and a particular software[21] has been implemented to power-cycle the node even normal console or network access is denied.

Virtual Honeypots Virtual honeypots can adopt similar approaches with physical honeypots. However, virtualization technique provides another even more convenient way: VMWare has supplied a set of powerful Perl API interfaces [9] so that a Perl script can be created to automate the process of VM initiation, snapshotting, and teardown; An UML VM could be easily started with a single shell command and provides a management console called *mconsole*[22] to monitor internal states and even halt or reboot the VM.

Both physical and virtual honeypots can be dynamically deployed in a very short period of time. Table 4.2 lists the times necessary to deploy different target honeypots. As expected, physical honeypots usually take longer than virtual honeypots since significant time is needed to transfer system image from a remote image server.

Image	*Linux configuration*	*Image size*	*Time*
S_I	rh-7.0_base_1.0/UML	29.3MB	3.0 sec.
S_{II}	rh-7.2-server/UML	253MB	22.0 sec.
S_{III}	rh-7.2-server/VMWare	500MB	30.0 sec.
S_{III}	rh-7.2-server/VMWare	500MB	30.0 sec.
S_{IV}	rh-7.0_base_1.0/Raw	32MB	50.2 sec.

Table 1. Service Bootstrapping Time

4.3 Security Mitigation

Deployed honeypots has in-born security risks and the main responsibility of the deactivation logic module is to closely monitor the traffic of them. It needs to take certain measures to isolate them once abnormalities like SYN flooding are detected. To mitigate associate risks, BAIT-TRAP leverages two open-source projects: snort-inline[5] and bridge-utils[1]. Snort-inline is able to limit the number of concurrent outgoing connection and silently drop malicious packets which contain well-known attack signatures. Bridge-utils is a Linux-based Ethernet bridging tool. When combined together, these tools are capable to act as a transparent firewall inspecting every packets heading/leaving honeypots and taking actions accordingly.

5 Evaluation

In this section, we show actual deployment of BAIT-TRAP and demonstrate the power of catering honeypots by presenting several captured intrusions.

5.1 Scanning Trend

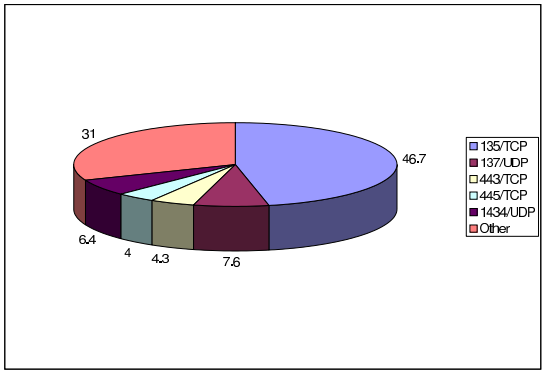
Figure 5(a) shows the average probing distribution detected in an experimental BAIT-TRAP testbed from Nov. 1st to Dec. 15th 2003. We identify several accountable service ports and show the probing trend in figure 5(b). The interested service ports include 135/TCP, 137/UDP, 443/TCP, 445/TCP, and 1434/UDP. The top five ports occupy 46.7%, 7.6%, 4.3%, 4%, and 6.4% of total abnormal traffic respectively, and contribute to almost 70 percentile of suspicious traffic.

- The 135/TCP and 445/TCP correspond to a security vulnerability in a Windows Distributed Component Object Model (DCOM) Remote Procedure Call (RPC) interface [17]. The infamous msblast worm[18] and similar variants like Welchia worm [19] exploit such vulnerability.
- The 137/UDP is related to the Samba server, which contains an exploitable buffer overflow[14] in versions 2.0.x through 2.2.7a. That vulnerability allows an external attacker to remotely and anonymously gain Super User (root) privileges.
- The 443/TCP is with popular Apache web server for HTTPS and there exists a vulnerability [12] for versions 2.0 through 2.0.36 within the code responsible for the handling of chunk-encoded HTTP requests. The vulnerability could allow remote attackers to execute arbitrary code with Apache account.
- The 1434/UDP is exploited by the infamous Sapphire worm [3] based on a stack buffer overflow vulnerability such that arbitrary code could be run on the SQL Server computer.

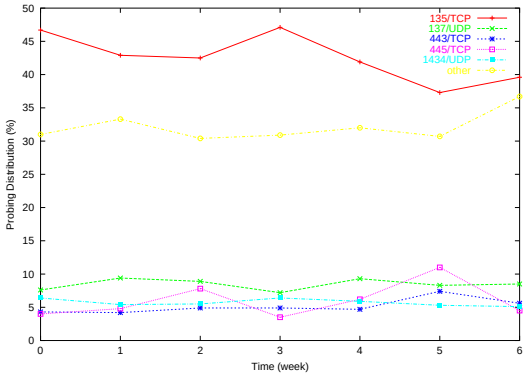
In this prototype of BAIT-TRAP, we are interested in deploying catering honeypots with popular services. The aforementioned ports are used and three catering honeypots are created: one with default Windows XP home edition, another with a vulnerable Samba/Linux server, and the last one with a vulnerable Apache/Linux service. The Windows honeypot was deployed inside VMWare and the image is generated offline. the Samba/Linux honeypot was dynamically created and deployed as an UML-based high-interaction virtual honeypot while the Apache/Linux honeypot was dynamically created and deployed as a high-interaction physical honeypot.

5.2 A WinXP/VMWare Catering Honeypot

Based on the profiled results, 135/TCP was the mostly attempted port number and the activation logic module instructed the deployment of corresponding vulnerable service - RPC DCOM service in Windows NT 4.0, Windows



(a) Probing Distribution on Destination Ports



(b) Probing Trend on Destination Ports

Figure 5. Probing Distribution & Trend on Destination Ports

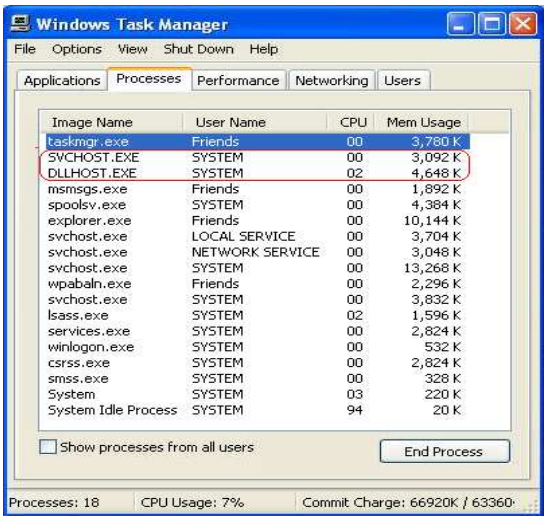


Figure 6. Screenshot: Welchia Worm in Action

2000, or Windows XP. A Windows XP honeypot named *cater_xp* without patching RPC DCOM vulnerability was deployed at 22:00pm, Dec. 5th, 2003, and got immediately infected by Welchia[19] worm in as short as 10 minutes. Before exposing the honeypot, a sniffer, i.e., *tcpdump*, was recording all of traffic coming in/out of the honeypot. Due to the space constraints, the infection process with anonymized source and destination IP addresses is available at project website [11]

At first, a TCP connection is initiated to *cater_xp* on port 135. Once the connection is established, RPC DCOM exploiting code is sent over. The exploiting code contains a connect-back shell code, which would connect back to the initiator on TCP port 707 if the vulnerability exists and could be successfully exploited using the exploiting code. The initiator will listen on the TCP port 707 and wait for the connection. Infection phrase succeeds when the exploiting code gets executed. Once the second connection is established from *cater_xp* to the initiator, the initiator will instruct the victim to download a copy of Welchia worm using tftp. Once the copy is cached in the victim side, the propagation phrase is done. The copy is executed immediately so that it becomes a new worm node. It only takes 2 seconds for infection to succeed. Based on the forensic analysis, the size of Welchia worm is only 10,240 bytes. Large volume of out-going ICMP echo requests initiated by the *cater_xp* after infection triggers the deactivation logic, which promptly isolates the *cater_xp* and prevents further infections.

Figure 6 is a snapshot of the Windows Task Manager in *cater_xp* showing the running instance of the Welchia worm. The process with SVCHOST.EXE is essentially a legitimate tftpd program renamed by the Welchia worm and the process with name DLLHOST.EXE is the Welchia worm itself. Welchia worm will be propagated from one machine to another based on peer-to-peer fashion and no centralized coordinator is involved. It is interesting to note (1) Welchia worm will try to end msblast process and remove the msblast [18] file if exists; (2) Welchia worm will attempt to connect to Microsoft's Windows Update website and download and apply the appropriate RPC DCOM vulnerability patch (*WindowsXP-KB823980-x86-ENU.exe* in this case), so that the victim will not be compromised again using the same vulnerability; (3) It may disable and remove itself if it is started in the year of 2004.

5.3 A Samba/Linux Catering Honeypot

The second incident is related to the second largest attempt: 137/UDP. According to current service set, it

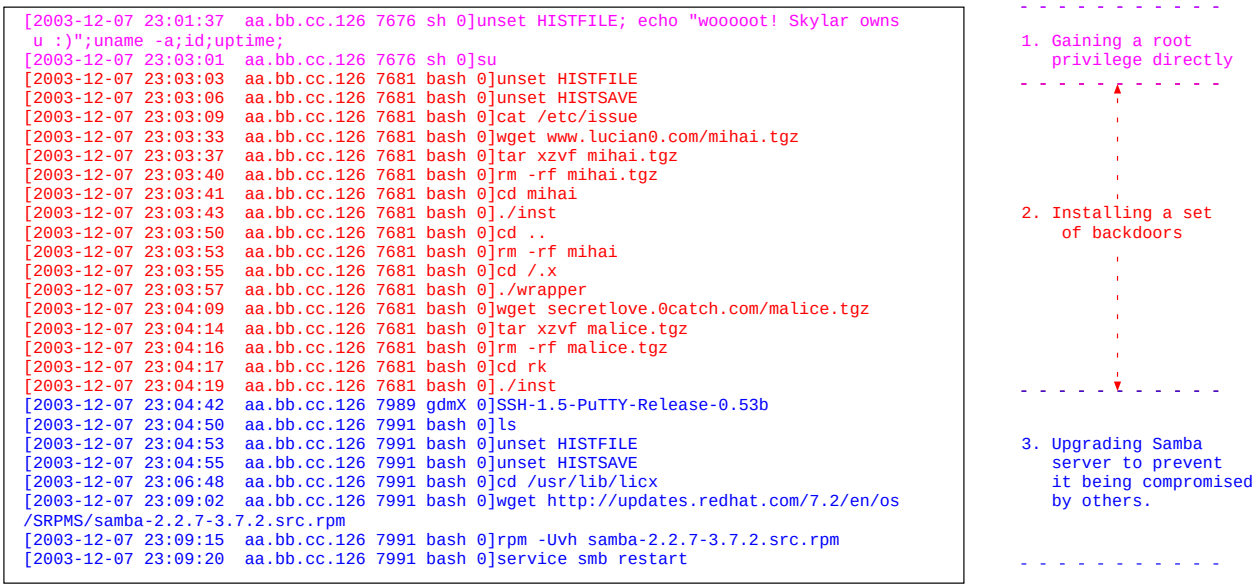


Figure 7. User Activity Monitoring After Successful Samba/Linux Break-in

is related to a vulnerability of Samba/Linux service. To understand the potential threats to this service, a Samba/Linux honeypot named *cater_samba* with unpatched version 2.2.1a-4 was deployed at 21:00pm, Dec. 7th, 2003, and got compromised in just 2 hours. With the honeypot factory module, *cater_samba* is pre-installed with Sebek [4], a kernel-level data capture tool, such that the keystrokes performed by the intruders after successful compromise will be recorded.

At first, a scan packet (netbios name packet) was sent against UDP port 137 and *cater_samba* responds with the mac address 00-00-00-00-00-00. Such response positively confirms the running status of a vulnerable samba server. After receiving the response, a TCP connection against port 139 was made, and several malicious packets were sent in the hope of causing buffer overflow. Due to the nature of vulnerability, the success of attempts will depend on the correctness of return address contained in malicious codes. The malicious code contains a port-binding shell code, which will listen on port 45295. Based on captured log information, we can identify several attempts (to be exact, 6 attempts) with different return addresses[3]. After successful exploitation, the intruder gained root privilege directly because of the Samba vulnerability and installed a set of pre-packaged tools (*mihai.tgz* and *malice.tgz*). The tools include a rootkit named SucKit [28] and a trojaned sshd daemon. Interestingly, based on the recorded keystrokes in figure 7, the intruder used the trojaned sshd daemon and upgrade the vulnerable Samba service with a new version so that no other intruders can later take advantage of the same vulnerability in gaining access to *cater_samba*.

[3]The return addresses used are 0xbffffed4, 0xbffffda8, 0xbffffc7c, 0xbffffb50, 0xbffffa24, 0xbffff8f8 accordingly.

5.4 An Apache/Linux Catering Honeypot

The third incident is related to the third largest probing: 443/TCP. According to current service set, it is related to a vulnerability in the Apache/Linux service. In order to understand the potential threats to this service, an Apache/Linux honeypot named *cater_apache* with unpatched version 1.3 was deployed at 20:00pm, Dec. 6th, 2003, and got compromised after shortly (2 hours and a half, to be more accurate). Similar to *cater_samba*, *cater_apache* is pre-installed with the kernel- level data capture tool, i.e. Sebek [4], by honeypot factory.

Firstly, a TCP connection heading for port 443 on *cater_apache* is firstly established. Once connected, the attacker sends multiple malicious packets containing a specially-crafted chunk-encoded HTTP request. The request will cause buffer overflow in Apache web server and execute the malicious code contained in the request. In this case, the code spawns a UNIX shell using apache account. The following keystrokes are captured and shown in figure 8.

Once a regular account is obtained, the intruder exploits some local vulnerability to escalate into the Super User (root) privilege. According to the recorded keystrokes, the intruder downloads a software named *expl*, which turns out a ptrace-exploiting [15] tool. Once executed, the intruder gains the omnipotent root privilege and begins to install a pre-packaged script, i.e., *naky.tgz*, which contains a trojaned ssh daemon, some infamous kernel-level rootkits like adore and knark, and a log cleaner etc. The trojaned ssh daemon

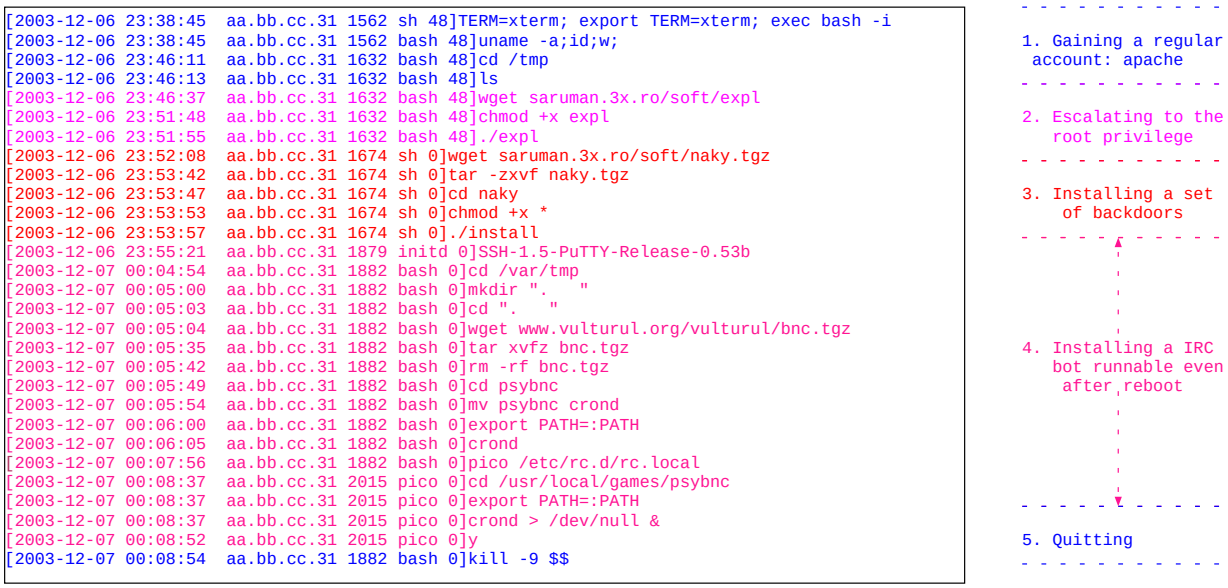


Figure 8. User Activity Monitoring After Successful Apache/Linux Break-in

will give the intruder a direct shell with root privilege. After the installation, the intruder takes the advantage of the trojaned ssh daemon to encrypt the communication and further install an IRC bot and reconfigure some system-wide files to make it auto-started even after system rebooting. The IRC bot enables the remote control of victims through a IRC channel so that the identity of the intruder will not be disclosed.

6 Related Work

The honeypot has recently emerged as an effective tool to detect intrusions and provide insights into new attacks [26, 10]. Catering honeypots share the same goal and have similar running requirements as conventional honeypots including efficient *ingress* and *egress* packet control [5] and high-value logging [4]. However, conventional honeypots, though effective, are manually deployed to wait passively for potential intrusions. As mentioned before, there is a need for catering honeypots. Most related work are bait-and-switch honeypots [6] and dynamic honeypots [30].

In bait-and-switch scheme, a honeypot partially mirrors a production server and cooperates with an intermediate node like a router or a firewall, which acts as the bait and switch. The intermediate node needs to accurately differentiate service-specific hostile intrusion attempts and transparently redirect all hostile traffic to the honeypot instead of the production server. Furthermore, such scheme is restrictive since it requires a close shadow execution of service to keep a synchronized soft states potentially only recognized by the service itself.

Dynamic honeypot is another interesting vision proposed by Lance Spitzner in [30]. A dynamic honeypot is a honeypot which could learn about the production network automatically and adapt itself into it. The dynamic honeypot should be automatically deployed in the production network once the learning phrase is done. One honeyd-based solution is also suggested. However, the vision focuses on low-interaction virtual honeypots and considers it difficult to have physical honeypot support. The notion of catering honeypot is similar to dynamic honeypot in [30]. However, instead of learning the production network, it learns from current network conditions and infers any potential imminent threats. Additionally, BAIT-TRAP is able to support high-interaction physical honeypots.

There are other notable improvements to single honeypot, like backtracker [25], ReVirt[23], and VM-based introspection [24] etc. Backtracker[25] is able to identify automatically potential sequences of steps that occurred in an intrusion based on system call recording. ReVirt is able to replay the attack scenario on instruction-by-instruction basis. VM-based introspection[24] is able to inspect inner machine states from the VM monitor, though it could be less effective when encountering encrypted traffic like ssh. These advancements could be directly adopted by BAIT-TRAP to help the management and forensic analysis of catering honeypots.

7 Conclusion

Instead of using current passive and static honeypot model, this paper presents the notion of catering honeypots and its implementation, i.e., BAIT-TRAP. A catering honeypot actively profile current network conditions and identify

one or a set of *attractive* target services for intruders. The target service will immediately be composed into a target honeypot and started as a "bait" to quickly understand interested attacks. BAIT-TRAP is able to automate this process within seconds. To the best of our knowledge, we are the first to propose the notion and implementation of catering honeypots. Real-world deployment and quickly captured incidents demonstrate the efficacy and practicality of catering honeypots.

References

[1] Bridge - Linux Ethernet Bridging. *http://bridge.sourceforge.net/*.

[2] Rhino Software, Inc. *http://www.serv-u.com/*.

[3] Sapphire Worm. *http://www.caida.org/analysis/security/sapphire/*.

[4] Sebek. *http://www.honeynet.org/tools/sebek/*.

[5] Snort-inline. *http://sourceforge.net/projects/snort-inline/*.

[6] The Bait and Switch Honeypot. *http://www.violating.us/projects/baitnswitch/*.

[7] The Honeynet Project. *http://www.honeynet.org*.

[8] VMWare. *http://www.vmware.com/*.

[9] VMWare Perl API. *http://www.vmware.com/support/gsx/doc/perl_api_gsx_linux.html*.

[10] CERT Advisory CA-2001-31 Buffer Overfbw in CDE Subprocess Control Service. *http://www.cert.org/advisories/CA-2001-31.html*, Jan. 2002.

[11] BAIT-TRAP. *http://www.cs.purdue.edu/homes/jiangx/BaitTrap*, Dec. 2003.

[12] CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability. *http://www.cert.org/advisories/CA-2002-17.html*, Mar. 2003.

[13] CERT/CC Overview Incident and Vulnerability Trends, CERT Coordination Center. *http://www.cert.org/present/cert-overview-trends/*, May 2003.

[14] CERT/CC Vulnerability Note VU-298233. *http://www.kb.cert.org/vuls/id/298233*, Mar. 2003.

[15] Linux Kernel Ptrace Privilege Escalation Vulnerability. *http://www.secunia.com/advisories/8337/*, Mar. 2003.

[16] MA-055.082003: W32.Nachi Worm. *http://www.mycert.org.my/advisory/MA-055.082003.html*, Aug. 2003.

[17] Microsoft Security Bulletin MS03-026. *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS03-026.asp*, 2003.

[18] MSBlast Worm. *http://isc.sans.org/diary.php?date=2003-08-11*, Aug. 2003.

[19] Welchia Worm. *http://securityresponse.symantec.com/avcenter/venc/data/w32.welchia.worm.html*, Aug. 2003.

[20] D. E. Comer and X. Jiang. Embedded Linux Design in Xinu Lab Environment. *Department of Computer Sciences Technical Report CSD TR 03-012, Purdue University, West Lafayette, IN*, May 2003.

[21] D. E. Comer and J. C. Lin. A Laboratory Environment For Experiencing With Xinu. *Department of Computer Sciences Technical Report CSD TR 96-047, Purdue University, West Lafayette, IN*, Jan. 1996.

[22] J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.

[23] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[24] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection . *Internet Society's 2003 Symposium on Network and Distributed System Security (NDSS)*, Feb. 2003.

[25] S. T. King and P. M. Chen. Backtracking Intrusions. *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[26] J. Levine, R. LaBella, H. Owen, D. Contis, and B. Culver. The Use of Honeynets to Detect Exploited Systems Across Large Enterprise Networks. *Proceedings of the 2003 IEEE Workshop on Information Assurance United States Military Academy, West Point, NY*, June 2003.

[27] N. Provos. A Virtual Honeypot Framework. *CITI Technical Report 03-1, University of Michigan, Ann Arbor*, Oct. 2003.

[28] sd. Linux on-the-fly kernel patching without LKM. *Phrack, 11(58):article 7 of 15*, Dec. 2001.

[29] L. Spitzner. Honeypots: Tracking Hackers. *Addison-Wesley, 2003 ISBN: 0-321-10895-7*.

[30] L. Spitzner. Dynamic Honeypots . *http://www.securityfocus.com/infocus/1731*, Sept. 2003.

[31] L. Spitzner. Honeytokens: The Other Honeypot . *http://www.securityfocus.com/infocus/1713*, July 2003.

[32] J. Vinet. Pacman: A simple package manager for Linux . *http://archlinux.org/pacman/*.