Technical Report
TR-2009-008

Amun: A Python Honeypot
Technical Report

Jan Göbel
Laboratory for Dependable Distributed Systems
University of Mannheim, Germany
jan.goebel@informatik.uni-mannheim.de

Abstract

In this report we describe a low-interaction honeypot, which is capable of capturing autonomous spreading malware from the internet, named *Amun*. For this purpose, the software emulates a wide range of different vulnerabilities. As soon as an attacker exploits one of the emulated vulnerabilities the payload transmitted by the attacker is analyzed and any download URL found is extracted. Next, the honeypot tries to download the malicious software and store it on the local harddisc, for further analyses. As a result, we are able to collect at best unknown binaries of malware that automatically spreads across the network. The collected samples can for example be used to help anti-virus vendors improve their signatures.

1 Introduction and Motivation

Autonoumously spreading malware is one of the main threats in todays' Internet. Worms and bots constantly scan network ranges worldwide for vulnerable machines to exploit and compromise. Compromised machines are then used to form large botnets for example to perform distributed denial of service attacks or send out masses of email spam.

With the help of honeypots we are able to capture such malware in a fast and straightforward fashion. Especially low-interaction honeypots, i.e. honeypots which allow little to no interaction with the attacker are very useful in this area of security. Server based honeypots ,like Amun, provide a range of emulated services to lure attackers and analyze the exploit code to get hold of the actual malware binary.

Low-interaction honeypots provide a low risk method for capturing information on initial probes, as there is no full interaction with an attacker. Thus, these honeypots are easier to maintain and enable the collection of information in an automated manner. This property renders low-interaction honeypots excellent sensors for intrusion detection systems (IDS).

2 Related Work

Several honeypot solutions have been developed in the past. In this section we introduce, three different implementations, that follow a similar approach as Amun does.

One of the most well known low-interaction honeypots is *Honeyd* [9]. It is a small Linux daemon, i.e. a program which runs in the background, creates a virtual host on a network and offers arbitrary vulnerable services. This virtual host can be configured to appear as a certain operating system, such as Microsoft Windows. Honeyd features a plug-in system for easy extension and some helpful tools like *Honeycomb* [7, 6], which can automatically generate intrusion detection signatures from the captured data. Generated signatures are currently support by Bro [8] and Snort [5]. The focus of Honeyd is mainly on the collection of attack information rather than capturing malware binaries.

The next two low-interaction honeypots follow the same scheme as Amun does. The first is called *Nepenthes* [1]. The second is called *Omnivora* [10]. Just like Amun, both honeypots aim at capturing malware in

an automated manner. The emulated services allow as much interaction as is needed for malicious software to download itself to the system. As soon as a connection is established to an emulated service, the appropriate vulnerability module is loaded to handle the incoming exploitation attempt. The payload send by an attacker is then analyzed to extract the location of the binary file, such as a trojan or worm. In the final step the honeypots download and store these files, so they can be further analyzed. Both solutions perform very well, but require good programming skills with either C++ or Delphi in order to extend the honeypots to personal needs.

In contrast to the latter two honeypots, Amun provides a wider range of vulnerability modules and is, due to its simpler structure, easier to deploy and maintain. The usage of a scripting language provides a straightforward way to extend Amun with new features without having to recompile the software everytime.

3 Amun Honeypot

In the following sections we describe the implementation and setup of the Amun honeypot software. First, we give a broad overview of the system followed by a detailed describtion of the different parts that are involved whenever an exploitation of the honeypot occurs.

3.1 Implementation

Amun is written in Python[1], a small and simple scripting language. The honeypot is made up of different components, which will be described in this section in more detail. Following is a short list of the most important components of Amun:

- Amun Kerneli (Section 3.1.1)
- Request Handler (Section 3.1.3)
- Vulnerability Modules (Section 3.1.4)
- Shellcode Analyzer (Section 3.1.5)
- Download Modules (Section 3.1.6)
- Logging Modules (Section 3.1.8)

[1] http://www.python.org

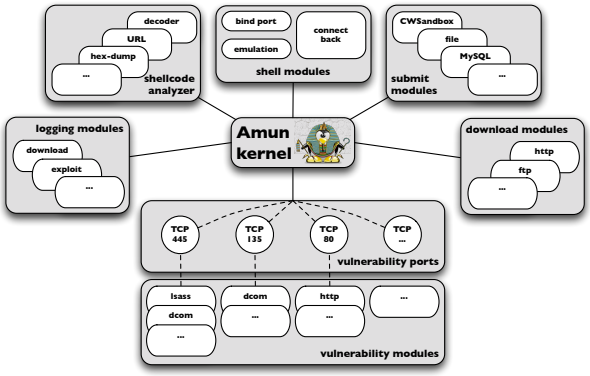

Figure 1: *schematic setup of Amun*

- Submission Modules (Section 3.1.7)

Figure 1 shows the schematic setup of Amun and the interaction of each part of the software with the kernel. Each of the above mentioned components is described in detail in the following sections.

3.1.1 Amun Kernel

The Amun Kernel is the core component of the honeypot. This part contains the startup and configuration routines, as well as, the main routines of the software. Amun is a single threaded application that uses the `select` operator to iterate over its sockets. Besides the socket operations Amun handles downloads, configuration reloads, shell spawning, and event logging in the main loop as well.

During the startup phase, the Amun Kernel initialises the regular expressions that are used for shellcode matching (Section 3.1.5), reads the main configuration file (Section 3.1.2), creates the internal logging modules, and loads all external modules. External modules are the vulnerability modules, that are responsible for emulating single vulnerabilities, the logging modules, that log attack information to other services like databases, and the submission modules, that for example write downloaded binaries to harddisc.

For each loaded vulnerability module the Amun Kernel retrieves the list of accociated ports and stores the vulnerability module in an array with the port as key (Figure 2).

```
Array
(
    [139] => Array
        (
            [0] => vuln-netdde
            [1] => vuln-ms06040
        )
    [445] => Array
        (
            [0] => vuln-ms08067
            [1] => vuln-ms06040
            [3] => vuln-ms06070
        )
)
```

Figure 2: *schematic view of the port to vulnerability array*

In the next step for each port a vulnerability module has registered to, i.e. the keys of the array (Figure 2), a TCP server is started. Amun also supports the use of UDP based services, but this feature is currently not in use and therefore it is not accessible through the configuration files.

After all initial modules are loaded and the appropriate TCP servers are started, Amun Kernel enters the main loop. During this loop, it iterates over all connected sockets, triggers download events, transfers information to certain modules, and re-reads the main configuration file for changes. The re-reading of the main configuration file allows to change certain settings during runtime, i.e. Amun does not have to be stopped and restarted.

3.1.2 Amun Configuration

Amun utilizes a single configuration file for adjusting all parameters necessary to run the honeyot. In this section we will briefly describe each of the options, their possible values and how it affects the honeypot. The main configuration file of Amun is the so-called *amun.conf* file, located in the configuration directory.

One of the core parameters is `ip`. It defines the IP address Amun will listen on during runtime. It takes a single IP address as parameter or the wildcard IP address `0.0.0.0` to listen on all addresses and interfaces assigned to the host system. It is also possible to provide an interface name (e.g. `eth0`), IP address ranges (*192.168.0.1 - 192.168.0.5*), CIDR notation for networks (*192.168.0.0/24*), or single comma separated IP adresses. Note that these last options do not scale well with large IP address ranges, as the operating system is limited to the number of opened socket descriptors. If more than about one hundred IP addresses are to be assigned, it is required to use the wildcard address.

Besides the IP address of the honeypot a `user` and `group` can be defined, which limit the privileges of Amun. After startup Amun will switch to the user and group defined here. However, in some cases exploits require the honeypot to open a ports below 1024, which can only be done with root privileges. In case Amun is running as non-root these request cannot be served.

Next, are some timeout parameters, which adjust the way Amun timeouts connections, open ports, and download requests. As some attacks might not work correctly it is possible, that attackers for example do not connect to the requested port, therefore, Amun needs to close this port after a certain amount of time has been passed. The options are named: `connection_timeout`, `bindport_timeout`, and `ftp_timeout`. The defined value represents the number of seconds to wait until Amun closes a connection.

Amun also offers the possibility to reject certain attacking hosts from reconnecting in the case of certain events. These events are: malware download was refused, download did not finish due to a timeout, a binary was already successfully downloaded, and the host already successfully exploited the honeypot. For each of the mentioned events the configuration file allows to set the block value and additionally a timeout value (seconds), defining how long a host should be blocked (Figure 3).

```
[...]
### block refused IPs, timeouts, successfull downloads,
### or successfull exploits for x seconds
### (can be changed while running)
refused_blocktime: 1200
timeout_blocktime: 1200
sucdown_blocktime: 1200
sucexpl_blocktime: 1200

### block ips which refuse a connection, throw a
### timeout, or from which we already have a
### successfull download or exploit
### (can be changed while running)
block_refused: 0
block_timeout: 0
block_sucdown: 0
block_sucexpl: 0
[...]
```

Figure 3: *configure certain block events*

These options are especially interesting if Amun is used as an intrusion sensor, for example. Most infected hosts attack a honeypot more than once, especially if the honeypot has more than one IP address assigned. To reduce the amount of log messages that are produced it is possible to block such a host for a certain amount of time. The reason why we also allow the blocking of hosts, from which we successfully downloaded a binary is, that in most cases a single host distributes only one binary within a certain time. It would be a waste of resources downloading the same file over and over again from the same host. Therefore we can reject any further connects from those hosts for a given time period.

For the TFTP download module Amun offers three extra options which can be modified: `tftp_retransmissions`, `tftp_max_retransmissions`, and `store_unfinished_tftp`. As TFTP uses the UDP protocol it can happen that packets get lost. For this reason Amun allows to set a number of retransmissions before giving up. The first option determines how many seconds to wait before a TFTP request is retransmitted, whereas the second options defines how many retransmissions Amun will make at all. The last option determines if Amun should also store unfinished TFTP downloads, i.e. a file is only partly downloaded.

A similar option as the `store_unfinished_tftp` is the `check_http_filesize` option. A lot of malware downloads use HTTP as transfer protocol and one feature of a HTTP server is to store the file size in the HTTP header of the reply. If `check_http_filesize` is enabled, Amun will compare the size of the downloaded binary with the value in received in the HTTP header. In case there is a mismatch, the downloaded file is discarded.

Another important feature is the `replace_local_ip` parameter. Whenever the Shellcode Analyzer extracts a download URL from the payload of an exploit, any found IP address is checked against a list of local IP addresses (e.g. 192.168.0.0/24). If *replace_local_ip* is enabled, Amun will replace all those IP addresses with the one of the attacker who send the exploit. Local IP addresses in shellcode occur whenever a host behind a Network Adress Translation (NAT) server is infected, because most malware acquires the IP address from the host configuration.

However, replacing the IP addresses also allows easy detection of the honeypot. If for example an attacker sends exploits with download URLs containing local IP addresses and the exploited host suddenly tries to download a file from the attackers host, the attacker knows that the IP address from the exploit must have been replaced and thus the attacked host must be a honeypot. Therefore, *replace_local_ip* is turned off by default.

Next, Amun allows the configuration of modules that should be started. The *submit_modules* list contains the modules that are responsible of handling any downloaded binary. The default module that is loaded is the *submit-md5* module, that simply stores any downloaded unqiue file to harddisc. Uniqueness is determined by the MD5 hash of the file. Additional modules of this type allow the transmission of binaries to external services like CWSandbox [11]. The *log_modules* are modules that perform certain logging functionality. In most cases these modules send information to external intrusion detection systems. The *vuln_modules* list contains all the vulnerability modules that should be load at the startup of Amun. Figure 4 displays the part of the configuration file that states what vulernability modules are to be loaded and what port is associated with each of the modules.

```
[...]
### define the vulnerability modules to load
### (can be changed while running)
vuln_modules:
        vuln-ms08067,
        vuln-netdde,
        vuln-ms06040,
        vuln-ms06070,
        [...]
        vuln-helix,
        vuln-hpopenview

### define ports for vulnerability modules
### (can be changed while running)
vuln-ms08067: 445
vuln-netdde: 139
vuln-ms06040: 139,445
vuln-ms06070: 445
[...]
```

Figure 4: *excerpt from the Amun main configuration file*

Finally, the configuration file contains some parameters that seldom need to be adjusted, namely: *honeypot_pingable*, *check_new_vulns*, *output_curr_sockets*, *log_local_downloads*, and *verbose_logging*. The first option allows to setup an iptables rule which blocks all incoming ping requests. The purpose of this option is to let the honeypot behave a little more like an out of the box

Microsoft Windows installation, as Windows also blocks ICMP echo requests by default. The second option determines the number of seconds to pass until Amun re-reads the configuration file for any changes. The third option is for debug purposes only. If it is set, Amun writes a list of all connected hosts to a file in the Amun root directory whenever a re-read of the configuration file occurs. The fourth option enables logging for download URLs containing local IP addresses and the last option provides more extensive logging for all parts of the honeypot. These options are usually needed for debugging, thus, by default they are turned off.

3.1.3 Request Handler

The Request Handler is responsible for all incoming and outgoing network traffic of the honeypot. For every connection request, that reaches the Amun Kernel a Request Handler is created, that handles the connection until it is closed. The Request Handler maintains the list of loaded vulnerability modules and delegates the incoming traffic to those modules that are registered for the current port.

Consider a connection coming in on port 445, if it is a new connection the Request Handler loads all vulnerability modules for port 445 by checking the vulnerability array (Figure 2) at the key 445. In the next step the incoming traffic is distributed to each of the modules returned by the previous step. Each of the vulnerability modules checks if the incoming traffic matches the service that is emulated and returns if it accepts or rejects the connection. As a result, the list of emulated vulnerabilities for a connection is thinned out with each incoming request of the attacker. In the worst case none of the registered modules matches the attack pattern and the connection is closed. Otherwise, there is exactly one module left, which successfully emulates all needed steps performed by the attacker and receives the final payload containing the download information of the malware. Note that incoming network packets can be distributed to all registered vulnerabilitiy modules, but a reply can only be send by one. In the best case there should only be one module left to reply after the first packet is received, however, if there are more left, the reply of the first module in the list is chosen.

Connections that for some reason do not match any of the vulnerability modules, or do not fit an emulated service at any stage create a log entry in the Amun Request Handler log. This log contains information about the attacking host and the request that was send. This information help to update existing vulnerability modules or create new ones.

The Request Handler also receives the results of the vulnerability module that successfully emulated a service and obtained the exploit payload from the attacker. This payload is passed on to the Shellcode Analyzer to detect any known shellcode. The results of the Shellcode Analyzer are again returned to the Request Handler, thus the Request Handler is the crucial point for any attack.

3.1.4 Vulnerability Modules

The vulnerability modules make up the emulated services which lure autonomous spreading malware. Each module represents a different service, for example a FTP server. The services are emulated only to the degree that is needed to trigger a certain exploit. That means, the emulated services cannot be regularly used, i.e. they do not offer the full functionality of the original service.

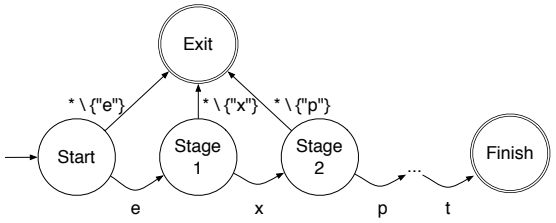

Figure 5: *finite state machine*

Vulnerabilities are realized as finite state machines. They usually consist of several stages that lead through the emulated service. Figure 5 shows an example of a finite state machine matching the word “exploit”. That means, each incoming network packet of an attacker is matched against the next state of the finite state machine. If it matches, the state of the vulnerbility module switches to the next stage, otherwise the vulnerabilitiy module rejects the incoming request. That way Amun assures that only requests that lead to the exploit of the emulated service are accepted. All data that leads to an undefined state is logged by the Request Handler. With this information

it is possible to determine changes in exploit methods and add new stages or even built new vulnerability modules.

To facilitate the process of writing new vulnerability modules, Amun supports XML to describe a module. This XML file is subsequentially transformed to Python code by Amun and can then be used as a vulnerability module. This means, that for simple vulnerability modules there is no need to write Python code.

Figure 6 illustrates an example of a XML document representing the parameters necessary to create the Plug and Play (PNP) vulnerability. It shows the number of stages (`<Stage>`) needed to trigger the exploit and for each stage the expected number of bytes (`<ReadBytes>`) together with the according byte sequence (`<Request>`). After the sixth stage the modul enters the shellcode collecting stage, i.e. at this point the exploit should already have taken place, and the attacker sends the shellcode. All data that is collected during the shellcode collection stage is subsequentially passed to the Request Handler and then to the Shellcode Analyzer.

To convert an XML file to its needed Python code there exists a small script named `vuln_creator.py`. Usage is as follows: `python vuln_creator.py -f filename.xml`. This eventually creates two new files named: `filename_modul.py` and `filename_shellcodes.py`. The first file contains the actual emulated service with the different stages and replies. The second file is optional and can contain certain requests needed to enter a new stage, like the request defined in the first stage of the example XML file displayed in Figure 6.

The final Python code of a vulnerability module consists of several different functions. The first function is for initialization of the module, here the name of the vulnerability, the starting stage, and a welcome message is defined. The welcome message is for example a banner displaying the service name and version upon the connection of an attacker. An example is shown in Figure 7. The main function of a vulnerability module is called `incoming`. This function receives the network packet, the number of bytes of this packet, the attacker IP address, a logging module, a previously created random reply, and the IP address of the honeypot. Figure 8 shows parts of the incoming function belonging to the vulnerability module that was created using the XML file described earlier.

In the first part of the incoming function a new reply

```
<Vulnerability>
 <Init>
  <Name>PNP</Name>
  <Stages>6</Stages>
  <WelcomeMess></WelcomeMess>
  <Ports>
   <Port>445</Port>
  </Ports>
  <DefaultReply>random</DefaultReply>
 </Init>
 <Stages>
  <Stage stage="1">
   <ReadBytes>137</ReadBytes>
   <Reply position="9">\x00</Reply>
   <Request>\x00\x00\x00\x85\xFF\x53\x4D\x42
            \x72\x00\x00\x00\x00\x18\x53\xC8
            \x00\x00\x00\x00\x00\x00\x00\x00
            \x00\x00\x00\x00\x00\x00\xFF\xFE
            \x00\x00\x00\x00\x00\x62\x00\x02
            \x50\x43\x20\x4E\x45\x54\x57\x4F
            \x52\x4B\x20\x50\x52\x4F\x47\x52
            \x41\x4D\x20\x31\x2E\x30\x00\x02
            \x4C\x41\x4E\x4D\x41\x4E\x31\x2E
            \x30\x00\x02\x57\x69\x6E\x64\x6F
            \x77\x73\x20\x66\x6F\x72\x20\x57
            \x6F\x72\x6B\x67\x72\x6F\x75\x70
            \x73\x20\x33\x2E\x31\x61\x00\x02
            \x4C\x4D\x31\x2E\x32\x58\x30\x30
            \x32\x00\x02\x4C\x41\x4E\x4D\x41
            \x4E\x32\x2E\x31\x00\x02\x4E\x54
            \x20\x4C\x4D\x20\x30\x2E\x31\x32
            \x00</Request>
  </Stage>
  <Stage stage="2">
   <ReadBytes>168</ReadBytes>
   <Reply position="9">\x00</Reply>
   <Request> [...] </Request>
  </Stage>
  <Stage stage="3">
   <ReadBytes>222</ReadBytes>
   <Reply position="9">\x00</Reply>
   <Request> [...] </Request>
  </Stage>
        [...]
  <Stage stage="5">
   <ReadBytes>106</ReadBytes>
   <Reply position="9">\x00</Reply>
   <Request> [...] </Request>
  </Stage>
  <Stage stage="6">
   <ReadBytes>160</ReadBytes>
   <Reply position="9">\x00</Reply>
   <Request> [...] </Request>
  </Stage>
 </Stages>
</Vulnerability>
```

Figure 6: *simple vulnerability module in XML*

```
[...]
def __init__(self):
  try:
    self.vuln_name = "PNP Vulnerability"
    self.stage = "PNP_STAGE1"
    self.welcome_message = ""
    self.shellcode = []
  except KeyboardInterrupt:
    raise
[...]
```

Figure 7: *vulnerability module initialization function*

```
[...]
def incoming(self, message, bytes, ip, vuLogger, \
                              random_reply, ownIP):
  try:
    self.reply = []
    for i in range(0,62):
      try:
        self.reply.append("\x00")
      except KeyboardInterrupt:
        raise

    resultSet = {}
    resultSet['vulnname'] = self.vuln_name
    resultSet['result'] = False
    resultSet['accept'] = False
    resultSet['shutdown'] = False
    resultSet['reply'] = "None"
    resultSet['stage'] = self.stage
    resultSet['shellcode'] = "None"
    resultSet["isFile"] = False

    if self.stage=="PNP_STAGE1" and (bytes==137 or \
                                          bytes==176):
      if pnp_shellcodes.pnp_request_stage1==message:
        resultSet['result'] = True
        resultSet['accept'] = True
        self.reply[9] = "\x00"
        resultSet['reply'] = "".join(self.reply)
        self.stage = "PNP_STAGE2"
        return resultSet
[...]
```

Figure 8: *vulnerability module incoming function*

is generated. Afterwards the result set, that is returned to the Request Handler after each stage, is defined. The result set contains the following keys:

- *vulnname* - name of the vulnerability module
- *accept* - defines if the incoming request matches the stage
- *result* - defines if the emulation is finished
- *reply* - contains the reply message
- *stage* - contains the current stage
- *shutdown* - indicates premature closing of the current connection
- *shellcode* - contains the shellcode that was transmitted by an attacker
- *isFile* - indicates if the shellcode field contains a file, i.e. instead of shellcode the attacker submitted a binary directly

The last part of the displayed incoming function shows the first stage check, the other stages are similar. In a first step a stage checks if the number of incoming bytes matches the length of an expected request, then it is checked if the request is identical to the one expected, this step is not always necessary. If all matched, the accept and result values are set to true, a reply is prepared and the next stage is set as new starting point for the next incoming request.

To quickly analyze certain ports for incoming attacks, Amun has a so-called analyzer vulnerability modul. This modul simply registers for certain ports defined via the configuration file, collects all incoming data, and sends it to the Shellcode Analyzer. The purpose of this module is to quickly analyze traffic hitting a certain port and see if there are any exploits in the wild.

Currently, Amun emulates 43 different vulnerable services on 53 different ports, an extract of the more well known vulnerabilities is displayed in Table 1. Most of the vulnerability modules have been constructed by analysing proof of concept exploits as provided by Milw0rm[2]. Others resulted from analysis of incoming requests recorded by the Request Handler.

[2] http://www.milw0rm.com/

CVE-ID	Description
CVE-2005-1272	Buffer Overflow CA ARCserver Backup Agent
CVE-2005-0491	Knox Arkiea Server Backup Stack Overflow
CVE-2003-0818	Buffer Overflow Microsoft ASN.1 - MS04-007
-	Axigen Mailserver Vulnerabilities
CVE-2005-0582	Buffer Overflow Comp-Associates License Client
-	DameWare Mini Remote Control Buffer Overflow
CVE-2003-0352	Buffer Overrun Windows RPC - MS03-026
CVE-2007-1748	Windows DNS RPC Interface - MS07-029
CVE-2007-1675	Buffer Overflow Lotus Domino Mailserver
-	Vulnerabilities in different FTP Server implementations
-	GoodTech Telnet Server Buffer Overflow
CVE-2006-6026	Heap Overflow Helix Server
CVE-2008-2438	HP OpenView Buffer Overflow
CVE-2006-4379	Stack Overflow Ipswitch Imail SMTP Daemon
CVE-2003-0533	Buffer Overflow LSASS - MS04-011
CVE-2005-0684	Buffer Overflow MaxDB MySQL Webtool
CVE-2005-4411	Buffer Overflow Mercury Mail
CVE-2005-2119	MSDTC Vulnerability - MS05-051
CVE-2005-0059	Buffer Overflow MS Message Queuing MS05-017
CVE-2006-3439	Microsoft Windows Server Service Buffer Overflow - MS06-040
CVE-2004-0206	Buffer Overflow Network Dynamic Data Exchange - MS04-031
CVE-2005-1983	Stack Overflow MS Windows PNP - MS05-039
CVE-2008-4250	Microsoft Windows RPC Vulnerability - MS08-067
-	Buffer Overflow Password Parameter SLMail POP3 Service
CVE-2006-2630	Symantec Remote Management Stack Buffer Overflow
CVE-2007-1868	Buffer Overflow IBM Tivoli Provisioning Manager
CVE-2007-4218	Buffer Overflows in ServerProtect service
CVE-2001-0876	Buffer Overflow MS Universal Plug and Play
CVE-2004-1172	Stack Overflow Veritas Backup Exec Agent
CVE-2004-0567	Buffer Overflow Windows Internet Naming Service
CVE-2006-4691	Workstation Service Vulnerability - MS06-070

Table 1: *excerpt of Amun vulnerability modules*

3.1.5 Shellcode Analyzer

In case a vulnerability module successfully emulated a service to the point where the attacker sends exploit code, all incoming data is recorded and finally transferred to the Shellcode Analyzer. The Shellcode Analyzer is the backbone of Amun, as it is responsible for shellcode recognition and decoding. Shellcode is recognized using several regular expression that match known parts of shellcode. In most cases this is the decoder part, a small loop that decodes the obfuscated shellcode back to its original. Figure 9 shows an example of a regular expression matching the decoder part of a certain shellcode. The four single bytes extracted make up the key that is used to decode the payload.

```
re.compile('\\xd9\\x74\\x24\\xf4\\x5b\\x81\\x73
\\x13(.)(.)(.)(.)\\x83\\xeb\\xfc\\xe2\\xf4', re.S)
```

Figure 9: *regular expression to match decoder part*

One can distinguish between clear text and encoded (obfuscated) shellcode. Obfuscation of shellcode is often achieved with the XOR operator using a single byte (simple XOR) or four bytes (multibyte XOR) or by using an alphanumeric encoding.

Clear text shellcode does not provide any methods of hiding its content, thus it simply contains for example an URL like *http://192.168.0.1/x.exe*. Therefore, one of the first steps of the Shellcode Analyzer is to check for unencoded URLs within the payload an attacker injected in our emulated vulnerabilities.

Simple XOR encoding means the shellcode is encoded using a single byte. The actual shellcode needs to be decoded prior to execution on the victim host, thus this kind of shellcode contains a so called decoder part at the beginning. The decoder part is a loop performing a XOR operation with the appropriate byte against the rest of the payload. The Shellcode Analyzer has several regular expression matching those decoder parts and extracting the needed XOR byte. In the next step the shellcode is decoded and the intructions are extracted. Instructions can again be a simple download URL, but also commands to open a certain port or connect back to the attacker and spawning a shell. The multibyte XOR variant is very much the same, but utilizes more than one byte to en-

code the shellcode. Figure 10 shows the decoder part for a multibyte XOR encoded shellcode. This assembler part preceeds the rest of the shellcode and is thus executed at first. The XOR key is `0x9432bf80`.

```
[...]
000001F9  EB19              jmp short 0x214
000001FB  5E                pop esi
000001FC  31C9              xor ecx,ecx
000001FE  81E989FFFFFF      sub ecx,0xffffff89
00000204  813680BF3294      xor dword [esi],0x9432bf80
0000020A  81EEFCFFFFFF      sub esi,0xfffffffc
00000210  E2F2              loop 0x204
[...]
```

Figure 10: *multibyte XOR decoder*

Alphanumeric shellcode encoding is a bit more different as its purpose is to use only alphanumeric characters for representation. The reason for using such prepared shellcode is that many new applications and intrusion detection mechanisms filter uncommon characters, thus using only characters like `0-9` and `A-Z` greatly reduces detection and improves the success rates.

In case the analyzed payload is not recognized by any of the regular expressions, a file containing the data is written to harddisc. This can be manually analyzed to integrate new regular expressions for shellcode detection.

```
cmd /c
  net stop SharedAccess &
  echo open 192.168.1.3 60810 >> tj &
  echo user d3m0n3 d4rk3v1l >> tj &
  echo get sr.exe >> tj &
  echo bye >> tj &
  ftp -n -v -s:tj &
  del tj &
  sr.exe &
  net start SharedAccess
```

Figure 11: *command found in shellcode*

Figure 11 shows an example of a FTP command that was found in encoded shellcode. The code instructs a Windows system to first disable the firewall and then write some instructions to a file named `tj`. This file is then executed as parameter to the FTP command. The file contains the address of the remote FTP server, username and password, as well as, the name of the file to download. After the binary is downloaded the `tj` file is deleted, the freshly downloaded file is executed, and the firewall is activated again.

The `-n` parameter given to the FTP command supresses the auto-login upon initial connection. The `-v` parameter supresses the display of remote server responses and `-s:filename` allows the specification of a text file containing FTP commands. The commands will be automatically executed upon the start of FTP.

The Shellcode Analyzer tries to extract all the information needed for the FTP download from such a command and triggers a download event at the Amun Kernel.

3.1.6 Download Modules

As described in the previous section the Shellcode Analyzer extracts the commands from the shellcode. These commands end up to be some kind of download method to get the actual malware, e.g. the worm binary.

As the goal of Amun is to capture autonomously spreading malware, we want to get hold of any advertised binary file, thus we need Amun to be able to handle different kinds of download methods. For each download method we can provide a module that is loaded upon the start of the honeypot. Amun currently provides four basic download modules, namely: HTTP, FTP, TFTP, and direct download. Following are examples for each of the different download methods. We use an URL like representation, as it is easier to read and display.

- *http://192.168.0.1/x.exe*
- *ftp://a:a@192.168.0.1:5554/32171_up.exe*
- *tftp://192.168.0.1:69/teekids.exe*
- *cbackf://192.168.0.1/ftpupd.exe*

The first three mehtods are well known and need not be described any further. The direct download method (`cbackf`) does not involve a transfer protocol. Amun simply connects to the provided IP address at a specified port and receives in return the binary directly. In a few cases some kind of authentication is needed, that is included in the shellcode. After connecting, the honeypot needs to send a short authentication string prior to receiving any data. This kind of download method has been named "connect back filetransfer" (`cbackf`).

Some shellcode does not contain download commands but require the honeypot to open a certain port or connect

to a certain IP address and spawn a Windows command shell. Such commands a handled by the bindshell module. This module emulates a Windows XP or 2000 shell to the connected attacker and understands a few commands. That means a human attacker will notice directly that this is not a real shell, however automated attack tools simply drop their instructions to the shell and exit. These instructions are collected and again analyzed by the Shellcode Analyzer to extract the actual download command. Figure 12 shows an interesting example of commands send to an emulated shell of Amun.

```
Cmd /c
  md i &
  cd i &
  del *.* /f /q &
  echo open new.setheo.com > j &
  echo new >> j &
  echo 123 >> j &
  echo mget *.exe >> j &
  echo bye >> j &
  ftp -i -s:j &
  del j &&
  echo for %%i in (*.exe) do
    start %%i > D.bat &
    D.bat &
    del D.bat
```

Figure 12: *command received at emulated shell*

The commands instruct the victim system to create a new directory called `i`, change to it and delete all files in there, using the parameters for quiet mode (`/q`), i.e. no questions asked, and the parameter to enforce deletion of read-only files as well (`/f`). In the next step a new file is created containing FTP commands to download certain files similar as in the example shown earlier. This time the attacker uses the `mget` command to retrieve multiple files. In the `FOR`-loop each downloaded binary is executed in its own separate window (`start`).

3.1.7 Submission Modules

Once a file has been downloaded using any of the above mentioned download modules it needs to be processed further. That means it can be stored to harddisc for example, or send to a remote service.

In the default configuration Amun only loads the submit-md5 module. This modules stores each downloaded binary to a certain folder on the harddrive. As a filename it uses the MD5 hash of the content of the file. The submit-md5 module consist of only a single function called `incoming` that is displayed in Figure 13.

```
[...]
def incoming(self, file_data, file_data_length, \
         downMethod, attIP, victimIP, smLogger, \
         md5hash, attackedPort, vulnName, downURL, \
         fexists):
  try:
    self.log_obj = amun_logging.amun_logging( \
         "submit_md5", smLogger)

    ### store to harddisc
    filename = "malware/md5sum/%s.bin" % (md5hash)
    if not fexists:
      fp = open(filename, 'a+b')
      fp.write(file_data)
      fp.close()
      self.log_obj.log("download (%s): %s (size: %i) \
        - %s" % (downURL, md5hash, file_data_length, \
        vulnName.replace(' Vulnerability','')), 12,  \
        "div", Log=True, display=True)
    else:
      self.log_obj.log("file exists", 12, "crit",  \
        Log=False, display=False)
  except KeyboardInterrupt:
    raise
[...]
```

Figure 13: *incoming function of the submit-md5 module*

The function gets several parameters from the Amun Kernel, including the file content, length, and MD5 hash. If the file does not already exist it is stored.

Other submission functions that are included with Amun are: *submit-cwsandbox*, *submit-anubis*, *submit-joebox*, and *submit-mysql*. These modules submit downloaded binaries to different sandbox services, that execute and analyze the behaviour of malware. The resulting reports are then accessible either by web or email.

Writing new submission modules is not very hard. The basic layout of a new submission module is illustrated in figure 14. The `__slots__` variable holds all variables which are global within an object created from the class. In this example this is the submission module name (`submit_name`) and the reference to the logging module (`log_obj`). If further global variables are defined, they need to be added to this list. In the `__init__` function the name of the submission module can be defined and all other preparations, that need to be made during the startup of Amun. The `incoming` function is called every time a binary is downloaded.

The following parameters are passed to the `incoming`

```
import psyco ; psyco.full()
from psyco.classes import *

import amun_logging

class submit(object):
  __slots__ = ("submit_name", "log_obj")

  def __init__(self):
    try:
      self.submit_name = "Submit MY_MODULE_NAME"
    except KeyboardInterrupt:
      raise

  def incoming(self, file_data, file_data_length, \
               downMethod, attIP, victimIP, smLoggr, \
               md5hash, attackedPort, vulnName, \
               downURL, fexists):
    try:
      self.log_obj = amun_logging.amun_logging(" \
                     submit_MY_MODULE_NAME", smLogger)

      [...]

    except KeyboardInterrupt:
      raise
```

Figure 14: *submission module layout*

function of each submission module:

- *file_data* - actual binary data
- *file_data_length* - length of the file
- *downMethod* - the download protocol, e.g. http
- *attIP* - IP address of the attacking host
- *victimIP* - IP address of the honeypot
- *smLogger* - reference to the submission log
- *md5hash* - MD5 hash of the file
- *attackedPort* - contains the port that was attacked
- *vulnName* - vulnerability modul that was exploitet
- *downURL* - URL where the binary was retrieved from
- *fexists* - does the file already exist on harddisc

When creating a new submission module the directory and files need to have the following form. The new directory needs to be placed within the `submit_modules` directory and the name must be of the form `submit-ModulName`, e.g., `submit-example`. The actual Python code must be placed within this new directory, in a file named `submit_ModulName`, e.g., `submit_example`. Note the underscore in the filename. To load the module add it to the main configuration file, as described in the configuration section.

3.1.8 Logging Modules

The logging modules provide an easy way to generate different kinds of notifications whenever an exploit occurs. Currently Amun offers five modules: *log-syslog*, *log-mail*, *log-mysql*, *log-surfnet*, and *log-blastomat*. The last logging module belongs to an intrusion detection system (IDS) developed at the RWTH Aachen called Blast-o-Mat [2]. The IDS uses honeypots as intrusion sensors to detect attacks in the network.

The log-syslog module sends all incoming attack information to the local syslog daemon. That way it is also possible to send attack information to remote machines, e.g., a central logging server. Another method is to use the log-mail module, that sends information about exploits to a predefined email address. Note, that according to the number of attacks, a lot of emails can be generated and flood the mail server. To prevent this, the block options of the configuration file can be used, as described in the configuration section.

The log-mysql module, allows the logging of attack information to a MySQL database. The layout for the database is stored in the configuration directory of Amun. This module is however still in development.

The log-surfnet module, allows the integration of Amun into the surfnet IDS, also called SURFids [4]. SURFids is an open source Distributed Intrusion Detection System based on passive sensors, like honeypots. SURFids uses PostgreSQL as underlying database.

Logging modules support three main functions to log events: `initialConnection`, `incoming`, and `successfullSubmission`. The first function is triggered upon a connection request of a host to the honeypot. This request must not be malicious at this point in time. The second function is called as soons as an exploit is detected and some kind of download method has been offered. The last function is called whenever a binary was

successfully downloaded, thus, this function receives the same parameters as the incoming function of the submission modules described previously.

```
import psyco ; psyco.full()
from psyco.classes import *

import time
import amun_logging
import amun_config_parser
import psycopg2

class log:
 def __init__(self):
   try:
     self.log_name = "Log MODUL"
     conffile = "conf/log-MODUL.conf"
     config = amun_cfg_parser.ConfigParser(conffile)
     self.sensorIP = config.getSingleValue("sensorIP")
     [...]
   except KeyboardInterrupt:
     raise

 def initialConnection(self, attackIP, attackPort, \
                 victimIP, victimPort, identifier, \
                 initialConnectionsDict, loLogger):
   [...]

 def incoming(self, attackIP, attackPort, victimIP, \
                   victimPort, vulnName, timestamp, \
              downloadMethod, loLogger, attackerID, \
                                   shellcodeName):
   [...]

 def successfullSubmission(self, attIP, attaPort, \
            victimIP, downloadURL, md5hash, data, \
                filelength, downMethod, loLogger, \
                             vulnName, fexists):
   [...]
```

Figure 15: *logging module layout*

Figure 15 shows the basic structure of a logging module. In the initialisation function, the name of the logging module is defined and if needed, a configuration file can be parsed. All operations defined in this function are performed at the start of Amun. Next are the three main logging functions that are executed by the Amun Kernel upon certain events. Most parameters are already described in section 3.1.7. The only new parameter is `attackerID`, which links an initial connection entry to the actual exploit that might happen. Note, that due to the single threaded design of Amun it is not possible to keep track of everything a single attacker performed.

4 Limitations

Although low-interaction server honeypots are a great addition to todays' intrusion detection mechanism, they also have some limitations. The most obvious limitation with low-interaction honeypots in general is the lack of capturing zero day attacks. The reason is that only vulnerabilities can be emulated that we already know of, thus this approach is always one step behind. The same restriction applies to the use of shellcode.

Next is the fact that the vulnerable services are not fully simulated with every feature they offer, but only the parts needed to trigger an exploit. As a result, low-interaction honeypots will not fool any human attacker, but only autonomous spreading malware, which does not verify the functionality of a service in the first place. Although such checks could be easily added, todays' malware is rather poorly written. There exist cases were not even the server reply is checked and the malware sends its shellcode regardless of the attacked service being vulnerable or not.

5 Results

In this section we present some statistics collected from a single Amun honeypot installation with a few thousand IP addresses assigned. The data was collected during the last twelve months, thus ranging from December 2008 till December 2009, with almost no downtime of the sensor.

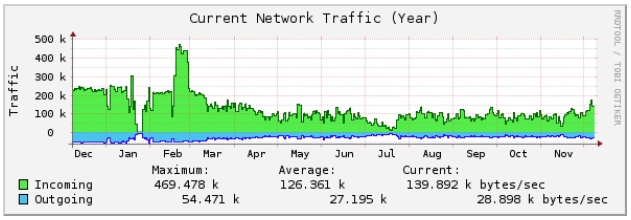

Figure 16: *oberserved network traffic*

Figure 16 shows the amount of network traffic observed at the honeypot. The data was generated in five minute intervals using RRDTool[3]. The average number of Kilobytes received is 126.36KB, whereas we saw a maximum during February 2009 with 469.48KB. Compared to the

[3] http://oss.oetiker.ch/rrdtool/

incoming traffic, the outgoing traffic is rather low, with an average of 27.2KB. The reason for this difference is, that the honeypot also receives the malware binaries, which usually make up the biggest amount of traffic.

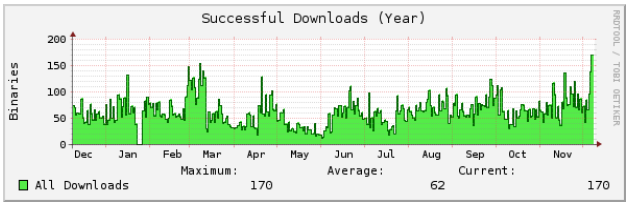

Figure 17: *successfull downloads of malicious software*

Figure 17 displays the number of successfully downloaded malware binaries seen over the last twelve months. For comparison, Figure 18 shows the number of successfully downloaded binaries for the last 24 hours. It shows that we have captured an average of 73 binaries every 5 minutes, with a maximum of 170 binaries, which is very similar to what we have seen over the last twelve months.

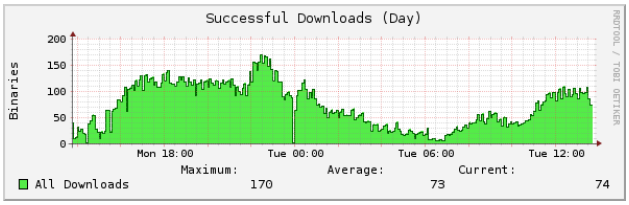

Figure 18: *successfull downloads of malicious software in the last 24 hours*

Figure 19 shows the increasing number of unqiue malware binaries that we have collected. Uniqueness is determined using the MD5 hash value of a binary. During the twelve months measurement period we have collected a total of 4790 malware binaries.

A more detailed analysis of honeypot data captured with low-interaction honeypots is presented here [3]. The complete analysis of all collected information during the twelve months period is left as future work.

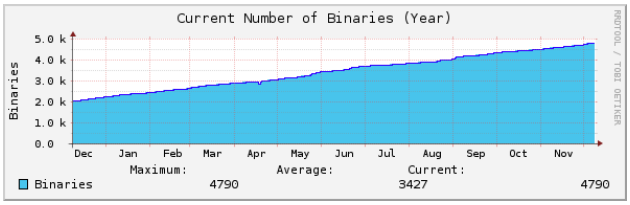

Figure 19: *unique malware binaries captured*

6 Conclusion

In this report we presented the low-interaction server based honeypot Amun. We gave a detailed describtion of the different parts of the software that are responsible to handle the emulation of vulnerabilities. We showed the individual modules for vulnerability emulation, logging, shellcoded analyses, and submission. All coordination between the different modules is handled by the Amun Kernel, which is the core part of Amun.

The main focus of Amun is to provide an easy platform for malware collection , like worms or bots. For this purpose Amun uses the simple scripting language Python and a XML based vulnerability module generation process to support an easy creation of new vulnerabiity modules. Thus, malware analysts are able to collect current malware in the wild and have the opportunity to extend the software without much programming knowledge.

Although low-interaction honeypots do have some limitation regarding zero-day attack detection, they also make up a great addition to todays' network security systems. Considering well placed honeypots throughout a network. These passive sensors can detect any scanning machine and report it to a central IDS, without any false positives. That means an alarm is only raised upon the exploitation of an emulated vulnerability.

Finally, honeypots are an important tool to study and learn about attackers and their procedures.

Acknowledgements.

We would like to thank Philipp Trinius who provided valuable feedback on a previous version of this report that substantially improved its presentation. Markus Engelberth for creating the schematic overview picture at the beginning of this report.

References

[1] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. In *RAID'06: 9th International Symposium On Recent Advances In Intrusion Detection*, 2006.

[2] J. Göbel. Advanced Honeynet based Intrusion Detection. Master's thesis, RWTH Aachen University, July 2006.

[3] J. Göbel, C. Willems, and T. Holz. Measurement and Analysis of Autonomous Spreading Malware in a University Environment. In *DIMVA'07: Detection of Intrusions and Malware, and Vulnerability Assessment*, 2007.

[4] R. Gozalbo. Honeypots aplicados a IDSs: Un caso practico. Master's thesis, University Jaume I., April 2007.

[5] J. Koziol. *Intrusion Detection with Snort*. Sams, Indianapolis, IN, USA, 2003.

[6] C. Kreibich and J. Crowcroft. Automated NIDS Signature Generation using Honeypots. In *SIGCOMM'03: Special Interest Group on Data Communication*, 2003.

[7] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *HotNets'03: Second Workshop on Hot Topics in Networks*, 2003.

[8] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *7th USENIX Security Symposium*, 1998.

[9] N. Provos. A Virtual Honeypot Framework. In *13th USENIX Security Symposium*, 2004.

[10] P. Trinius. Omnivora: Automatisiertes Sammeln von Malware unter Windows. Master's thesis, RWTH Aachen University, September 2007.

[11] C. Willems, T. Holz, and F. Freiling. CWSandbox: Towards Automated Dynamic Binary Analysis. *IEEE Security and Privacy*, 5(2), 2007.