Amun: Automatic Capturing of Malicious Software

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

Abstract: This paper describes the low-interaction server honeypot Amun. Through the use of emulated vulnerabilities Amun aims at capturing malware in an automated fashion. The use of the scriping language Python, a modular design, and the possibility to write vulnerability modules in XML allow the honeypot to be easily maintained and extended to personal needs.

1 Introduction

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

With the help of honeypots we are able to capture such autonomously spreading 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 network security. Server based honeypots provide a number of emulated services to lure attackers, monitor the attack, and analyze the exploit code to get hold of the self-propagating 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).

In this paper we introduce *Amun* a highly flexible and lightweight low-interaction honeypot, designed to capture malware that spreads by exploiting server based vulnerabilities. The use of a scripting language allows it to be easily extended and ported to different operating systems. Furthermore, Amun tries to actually "speak" the required protocol of an application an attacker is trying to exploit, making it more successful at fooling attackers.

2 Related Work

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

One of the most well known low-interaction honeypots is *Honeyd* [Pro04]. It is a Linux application that runs in the background, creates virtual hosts on a network and offers different vulnerable services. The virtual hosts can be configured to appear as a certain operating system, such as Microsoft Windows for example. Honeyd features a plug-in system for easy extension and some helpful tools like *Honeycomb* [KC03b, KC03a], that can automatically generate intrusion detection signatures from the captured data. Generated signatures are currently support by Bro [Pax98] and Snort [Koz03]. The focus of Honeyd is mainly on the collection of attack information rather than capturing malware binaries itself.

The following two low-interaction honeypots follow the same scheme as Amun does. The first is called *Nepenthes* [BKH+06]. The second is called *Omnivora* [Tri07]. Just like Amun, both honeypot solutions aim at capturing malware in an automated manner by emulating well-known vulnerabilities. Both solutions perform very well, but require good programming skills either in C++ or Delphi, in order to extend or modify 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 every time. Additionally, Amun uses protocol specific replies instead of random byte replies to be more efficient in fooling attacks.

3 Amun Honeypot Architecture

Amun is written in Python[1], a small and simple scripting language. The honeypot is made up of different components that handle the individual parts needed for vulnerability simulation and shellcode detection, which will be described in this section in more detail. Following is a short list of the most important components of Amun:

- Amun Kernel (Section 3.1)
- Amun Request Handler (Section 3.2)
- Vulnerability Modules (Section 3.3)
- Shellcode Analyzer (Section 3.4)
- Download Modules (Section 3.5)
- Submission Modules (Section 3.6)
- Logging Modules (Section 3.7)

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

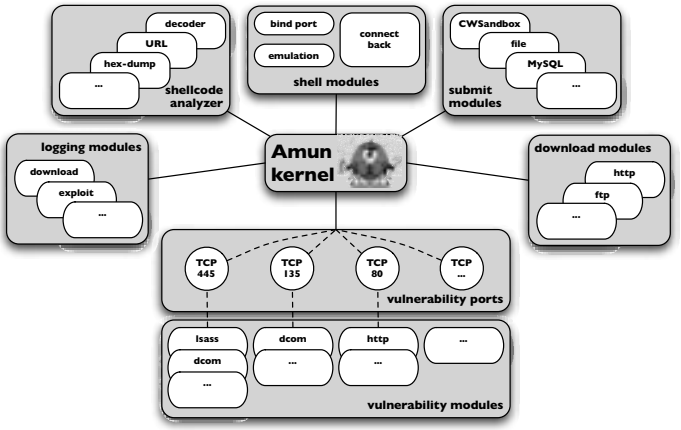

Figure 1: schematic setup of Amun

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 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 routine of the software. Amun is a single threaded application that uses the `select` operator to iterate over all open sockets. Besides these socket operations the Amun Kernel handles downloads, configuration reloads, shell spawning, and event logging in the main loop as well.

During the startup phase, the Amun Kernel initializes the regular expressions that are used for shellcode matching, reads the main configuration file 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 hard disc.

For each loaded vulnerability module Amun starts a TCP server listening on the ports the module has registered for.

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 latter option allows the reconfiguration of Amun during runtime, i.e. without the need to restart the honeypot.

3.2 Amun 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 to port 445: if it is a new connection the Request Handler loads all vulnerability modules registered for port 445. In the next step the incoming traffic is distributed to each of these modules. Each of the vulnerability modules checks if the incoming traffic matches the service that it emulates and either 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, that 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 vulnerability modules, but a reply packet 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.

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.

3.3 Vulnerability Modules

The vulnerability modules make up the emulated services that lure autonomous spreading malware to initiate an exploit. Each module represents a different vulnerable 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.

Vulnerabilities are realized as finite state machines. They usually consist of several stages that lead through the emulated service. Figure 2 shows an example of a finite state machine describing a buffer overflow vulnerability in the ExchangePOP3 v5.0 software. Upon the first connection of an attacker Amun sends the banner of the emulated service and awaits the first command. In this case stage one waits for the command `Mail`, all other input leads to the dropping of the connection.

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.

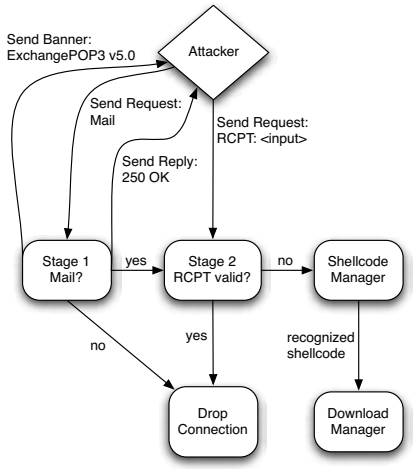

Figure 2: example of a finite state machine

To facilitate the process of writing new vulnerability modules, Amun supports XML to describe a module. This XML file is subsequently 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 3 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 module 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 subsequently passed to the Request Handler and then to the Shellcode Analyzer.

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. 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.

To quickly analyze certain ports for incoming attacks, Amun has a so-called analyzer vulnerability module. This module 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 unrecognized exploits in the wild.

Currently, Amun emulates 43 different vulnerable services on 53 different ports. An ex-

```
<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
            [...]
            \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="6">
   <ReadBytes>160</ReadBytes>
   <Reply position="9">\x00</Reply>
   <Request> [...] </Request>
  </Stage>
 </Stages>
</Vulnerability>
```

Figure 3: simple vulnerability module in XML

tract of the more well known vulnerabilities is displayed in Table A in the Appendices.

3.4 Shellcode Analyzer

If 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 expressions 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.

Shellcode can be distinguished as being either clear text or encoded (obfuscated). 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.

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(s) 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 instructions 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 multi-byte XOR variant is very much the same, but utilizes more than one byte to encode the shellcode.

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 of exploits.

If the analyzed payload is not recognized by any of the regular expressions, a file containing the data is written to hard disc.

3.5 Download Modules

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 methods 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 file transfer (`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 are 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 4 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 4: 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 it, using the parameters for quiet mode (`/q`), 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 all files with the `.exe` extension. 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`).

3.6 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 hard disc for example, or send to a remote service.

In the default configuration Amun only loads the *submit-md5* module. This module stores each downloaded binary to a certain folder on the hard disc. As a filename it uses the MD5 hash of the content of the file.

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

3.7 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 [Göb06]. 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 that handle reoccurring exploits.

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 [Goz07]. 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 soon 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.

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. Amun can only recognize shellcode it already knows of.

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 automatically 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 resulting from data collected at 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.

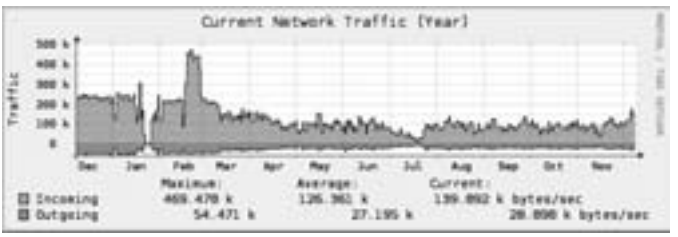

Figure 5: oberserved network traffic

Figure 5 shows the amount of network traffic observed at the honeypot. The data was generated in five minute intervals using RRDTool[2]. The average number of Kilobytes received is 126.36KB, whereas we saw a maximum during February 2009 with 469.48KB. Compared to the 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.

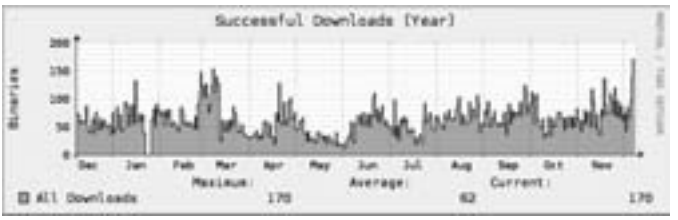

Figure 6: successfull downloads of malicious software

Figure 6 displays the number of successfully downloaded malware binaries seen over the last twelve months. For comparison, Figure 7 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 8 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

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

Figure 7: successfull downloads of malicious software in the last 24 hours

measurement period we have collected a total of 4790 malware binaries. Note that the used uniqueness criteria is not reliable, as polymorphic malware for example produces a different hash each time it is downloaded.

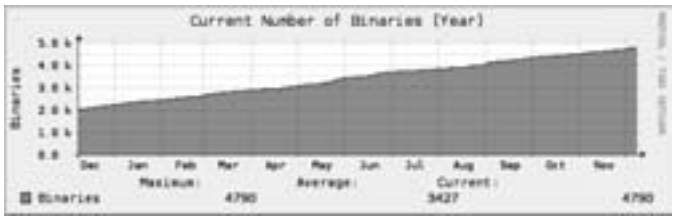

Figure 8: unique malware binaries captured

A more detailed analysis of honeypot data captured with low-interaction honeypots is presented at the DIMVA conference in 2007 [GWH07]. The complete analysis of all collected information during the twelve months period is left as future work.

6 Summary

In this paper we presented the low-interaction server based honeypot Amun. We gave a detailed description of the different parts of the software that are responsible to handle the emulation of vulnerabilities and download the propagating malware. 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 vulnerability modules. Thus, malware analysts are able to collect current malware in the wild and have the opportunity to extend the honeypot 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 falsely generated alarms. 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 to compromise machines in the Internet.

Acknowledgements.

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

References

[BKH+06] 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.

[Göb06] J. Göbel. Advanced Honeynet based Intrusion Detection. Master's thesis, RWTH Aachen University, July 2006.

[Goz07] R. Gozalbo. Honeypots aplicados a IDSs: Un caso practico. Master's thesis, University Jaume I., April 2007.

[GWH07] 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.

[KC03a] C. Kreibich and J. Crowcroft. Automated NIDS Signature Generation using Honeypots. In *SIGCOMM'03: Special Interest Group on Data Communication*, 2003.

[KC03b] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *HotNets'03: Second Workshop on Hot Topics in Networks*, 2003.

[Koz03] J. Koziol. *Intrusion Detection with Snort*. Sams, Indianapolis, IN, USA, 2003.

[Pax98] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *7th USENIX Security Symposium*, 1998.

[Pro04] N. Provos. A Virtual Honeypot Framework. In *13th USENIX Security Symposium*, 2004.

[Tri07] P. Trinius. Omnivora: Automatisiertes Sammeln von Malware unter Windows. Master's thesis, RWTH Aachen University, September 2007.

Appendices

A List of Emulated Vulnerabilities

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

Table 1: excerpt of Amun vulnerability modules