# Shellcode Detection in IPv6 Networks with HoneydV6

Sven Schindler[1], Oliver Eggert[1], Bettina Schnor[1] and Thomas Scheffler[2]
[1]*Department of Computer Science, University of Potsdam, Potsdam, Germany*
[2]*Department of Electrical Engineering, Beuth University of Applied Sciences, Berlin, Germany*
{*sschindl, oeggert, schnor*}*@cs.uni-potsdam.de, scheffler@beuth-hochschule.de*

Keywords: Honeypot, IPv6, Shellcode.

Abstract: More and more networks and services are reachable via IPv6 and the interest for security monitoring of these IPv6 networks is increasing. Honeypots are valuable tools to monitor and analyse network attacks. HoneydV6 is a low-interaction honeypot which is well suited to deal with the large IPv6 address space, since it is capable of simulating a large number of virtual hosts on a single machine. This paper presents an extension for HoneydV6 which allows the detection, extraction and analyses of shellcode contained in IPv6 network attacks. The shellcode detection is based on the open source library *libemu* and combined with the online malware analysis tool Anubis. We compared the shellcode detection rate of HoneydV6 and Dionaea. While HoneydV6 is able to detect about 25 % of the malicious samples, the Dionaea honeypot detects only about 6 %.

## 1 INTRODUCTION

Current measurements of IPv6 adoption[1] show that the new Internet protocol is seeing increased deployment and in some countries approaches almost 10% of all network traffic.

The deployment of IPv6-enabled security tools on such networks is necessary in order to gain a deep understanding how attackers might adapt to IPv6 and what strategies are used to target the new network protocol. It is therefore necessary to have adequate security tools, like honeypots, available that allow a deep analysis of usage and attack patterns in IPv6 networks.

Honeypots are very different from most traditional security mechanisms. They provide a security resource whose value lies in being actively probed, attacked, or compromised (Spitzner, 2002). The honeypot system has no production value and is configured to monitor unused address space. Every communication with the honeypot is considered potentially hostile traffic and has to be analysed. A honeypot can be something like a computer or even a mobile phone which is set up to attract attackers. By using various monitoring mechanisms, honeypots can provide valuable data about an attack and the strategies used by the attacker. This information can later be used to develop appropriate countermeasures.

[1]http://www.google.com/ipv6/statistics.html

Intrusion detection systems like *snort* (Beale et al., 2007) passively observe the network communication between multiple hosts and send alarms if they detect patterns that indicate malicious activities. Honeypots, on the other hand, are interacting with an adversary directly and therefore have a lot more control and insight into the communication. This makes even end-to-end encrypted communication channels available, such as between a Command-and-Control-Server and its bots, that could not be analysed by a network-based intrusion detection system.

In many cases, network attacks try to exploit vulnerabilities of the target system to run arbitrary code under the privileges of the attacked service (Polychronakis et al., 2006). A typical target action is to spawn a shell on the remote system. This is the reason why the transferred machine code is commonly called shellcode. Attacking networks with exploits carrying shellcode has become comparatively easy due to the availability of tools like the Metasploit framework (Metasploit, nd), which has recently started to target IPv6-specific risks.

There currently exist two general-purpose honeypots which are capable of simulating different IPv6 network services: HoneydV6 (Schindler et al., 2013) and Dionaea (Dionaea, nd). Other special-purpose honeypots like the ssh honeypot *kippo* (kippo, nd) might be used in IPv6 networks but are not generic enough to capture a wide range of attack scenarios. Dionea already provides the ability to detect shellcode attacks through the *libemu* shellcode detection library (Baecher and Koetter, nd). HoneydV6, on the other hand, did not have the capability to detect and to extract shellcode binaries.

Since HoneydV6 handles certain IPv6 specific properties, like the huge address space of an IPv6 network, better than Dionaea (Schindler et al., 2013), we decided to implement shellcode detection into HoneydV6 and compare the achievable detection rates.

The following section provides an overview over HoneydV6. In Section 3 we introduce mechanisms to detect shellcode contained in network traffic. Next, we present the architecture of our HoneydV6 extension in Section 4. We evaluated our implementation by sending various shellcode samples to HoneydV6 and compared the detection rate of our implementation with the detection rate of Dionaea. The results of this experiment are presented in Section 5. The paper concludes with a summary.

## 2 HONEYDV6

In (Schindler et al., 2013), the authors presented the low-interaction honeypot HoneydV6 which can be used to simulate large IPv6 networks with thousands of routers and hosts. HoneydV6 is an extension of the well-known low-interaction honeypot *Honeyd* (Honeyd, nd) focusing on attack detection in IPv6 networks.

By using the network packet capture library pcap (McCanne and Jacobson, nd), HoneydV6 is able to simulate multiple machines and even entire networks including network latency and simulated packet loss on a single machine. HoneydV6 implements a custom IPv6 network stack instead of using the network stack of the underlying operating system.

The honeypot is a framework for network services and relies on external scripts to emulate services like telnet, ssh, or http servers. HoneydV6 provides a basic logging functionality to log events like ICMPv6 echo requests or connection attempts.

The honeypot does not further inspect the communication between script and adversary, except for some basic statistical analyses. There is no implementation of any attack detection or classification mechanism available. This makes it difficult to differentiate between interesting traffic caused by new attacks and potentially uninteresting traffic of well-known attacks that may be seen on a daily basis.

Our intention is to eliminate the lack of detection capabilities by extending HoneydV6 with shellcode detection mechanisms as discussed in the next section.

## 3 SHELLCODE DETECTION AND ANALYSIS

Shellcode detection allows honeypots and intrusion detection systems to identify network traffic that contains malicious code that tries to compromise the targeted system.

A simple approach for the detection of a shellcode attacks is the matching of network traffic against known code-samples. The well-known intrusion detection system *snort*, for example, runs a signature-based shellcode detection which compares packets against a database containing fingerprints for various attacks. Signature-based shellcode detection is easy, but it has a number of disadvantages. It is not possible to detect new attacks for which no fingerprints are available. The fingerprint database needs to be up to date in order to achieve any substantial detection rates. When new signatures are distributed, they become publicly known and malware developer can test their code samples against the signature base to evade detection. This has lead to the development of so-called polymorphic and metamorphic malware which dynamically changes its signature and behaviour to work around signature-based detection (Polychronakis et al., 2006).

One approach of polymorphic malware is to encrypt the machine code before sending it to the victim. Once the encrypted code is executed on the target machine, it decrypts itself using an attack-specific key. Changing the encryption key produces a new malware signature which cannot be matched against existing fingerprints of a honeypot or an intrusion detection system.

In contrast to polymorphic malware, metamorphic malware does not only change its structure but also its behaviour. One example for metamorphic malware is *Win95/Regswap* which uses different CPU registers to store essential computation values for each instance (Ször and Ferrie, 2001).

These examples show that a simple signature-based shellcode detection is not sufficient for modern honeypot architectures to automatically detect new malware samples. An alternative approach is to use shellcode detection libraries which locate, execute and monitor machine code contained in network traffic in a secure sandbox. The following sections give an introduction into emulation-based shellcode detection libraries. We will focus on the functionality of the open-source library *libemu* which is used by Dionaea before we discuss alternatives.

## 3.1 Shellcode Detection by Emulation

Polychronakis et al. are using a different approach to detect even polymorphic malware (Polychronakis et al., 2006). Instead of matching malware against a precomputed fingerprint, they propose to execute and monitor potential malicious code as long as possible. While there are a number of scientific documents about this topic available, we could find only one open source library, the *libemu* (Baecher and Koetter, nd), which safely executes and evaluates malicious machine code. Other implementations are either not public or turned into a commercial product.

Libemu was developed in 2007 by Paul Baecher and Markus Koetter in the C programming language. The same authors developed Dionaea which includes *libemu* as shellcode detection component. Libemu implements an x86 emulator with the corresponding registers, program counter, virtual memory and a machine code disassembler. This way, libemu is able to run shellcode in a safe and insulated environment on the emulated processor.

Besides executing and checking machine code, *libemu* is able to detect machine code in an arbitrary byte sequence. Malicious shellcode that was transferred over a network and installed on the target system is usually located somewhere in the target system's memory. In order to work, many malware types need to execute functions which determine its own location within the memory. Code implementing those functions is called GetPC code, derived from the expression "get program counter". An example for GetPC code is an x86 *CALL* command followed by a statement that accesses the return address which is stored on the stack due to the *CALL* command execution. Libemu provides a function called `emu_shellcode_test()` to locate GetPC code in a given byte sequence. If a GetPC code could be found then `emu_shellcode_test()` returns the position of the detected code sequence.

Listing 1 shows the *libemu* code excerpt that inspects a given byte sequence for call-based GetPC code. The code block is entered if the currently inspected byte represents an assembly `call` instruction. Libemu creates a copy of the current stack pointer of the emulated CPU `c` and starts emulating the `call` and subsequent assembly instructions. If *libemu* detects that the stack pointer points to the same location as before executing the instructions then it returns 1, indicating that a GetPC code sequence has been found. This is the case if the shellcode popped the current code location, which was previously pushed on the stack due to the `call` instruction, from the stack.

Besides the call-based GetPC code, *libemu* is able to detect malware using the `fstenv` instruction. The instruction can be used to store debug information about the state of the floating point unit into memory. These debug information also contain the current program counter.

```
/* call */
case 0xe8:

  /*...*/

  uint32_t espcopy = emu_cpu_reg32_get(c,esp);
  int j;
  for (j=0;j<64;j++)
  {
    int ret = emu_cpu_parse(emu_cpu_get(e));

    if (ret != -1)
    {
      ret = emu_cpu_step(emu_cpu_get(e));
    }

    /*...*/

    if (emu_cpu_reg32_get(c, esp) == espcopy)
      return 1;
  }

  return 1;
  break;
```

Listing 1: Excerpt of *libemu*'s emu_get_pc function.

After locating possible malware in a byte sequence, *libemu* can be used to emulate the potential shellcode and try to detect accesses to system functions and libraries. This monitoring process is called malware profiling. Libemu is able to create a call graph which visualises malware behaviour.

An alternative malware profiling tool called *Shellzer* was presented in 2011 by Fratantonio et al. (Fratantonio et al., 2011). The tool also emulates malware and analyses accesses to system calls. However, Shellzer is limited to the analysis of JavaScript-, Flash and PDF malware. Since it is no general purpose malware detection library, it is not suitable for integration into the low-interaction honeypot.

The next subsections will present alternative online and offline profiling approaches which may be included into an honeypot architecture.

## 3.2 Execution on a Real OS

Libemu emulates system resources like CPU or operating system calls in order to monitor malware behaviour. An alternative approach is the execution of malware on a real operating system. Of course, this requires a new operating system setup after each run of a malware sample. Virtual machines provide a convenient tool to setup isolated machines which can be reset to an initial state after each malware execution.

This approach is used by Cuckoobox (Sandbox, nd), a project that was developed by Claudio Guarnieri within the scope of the Google Summer of Code 2010. Cuckoobox executes malware and monitors system function calls, created files, downloads, and network traffic. Furthermore, it is able to create screenshots during the malware execution. Cuckoobox consists of a host and multiple guest machines. The host machine is responsible for receiving the malware samples and distributing the samples to available guest machines. A guest machine executes the malware sample and monitors the execution, e.g. file accesses and system calls. The analysis results of this approach are much more specific than the results produced by *libemu*. However, this approach needs a lot more system resources because a set-up containing a real operating system is needed.

### 3.3 Online Malware Analysis

The additional administration and system requirements of Cuckoobox can be avoided by using online malware analysis services like Malwr (Malwr, nd) or Anubis (Anubis, nd). Malwr provides a web interface for a Cuckoobox backend which is running on servers managed by Malwr. Anubis is a similar web service which allows to upload Windows-based malware and which monitors access to the file system, the Windows Registry as well as network and process activities. In contrast to Malwr, Anubis provides an interface to automatically upload malware samples. The resulting protocols can be downloaded in HTML-, XML, PDF- or ASCII text format. The captured network traffic is also provided as a pcap file.

### 3.4 Discussion

The malware profiling capabilities of the sandboxed *libemu* execution can mainly detect code-sequences that represent entry points for malicous activities.

Malware analysis on real operating systems can provide a much higher level of interaction with system components and services and makes classification of malware behaviour easier. However, it also requires a sophisticated test architecture that is not easy to build and maintain for every honeypot installation.

Therefore, we decided to extend HoneydV6 with *libemu* to detect *potential* malware in IPv6 network traffic. Due to its modular structure and convenient high-level functions, *libemu* can be easily integrated into HoneydV6's network flow. Once potential malware has been identified, we can extract this code and send it to an operating system-based analysis service like Anubis. By doing so, we combine the strengths of both approaches in the analysis of network attacks.

## 4 ARCHITECTURE AND IMPLEMENTATION

This section presents the integration of *libemu* and Anubis into HoneydV6. Furthermore, we introduce an extended database logging facility which simplifies analysing monitored attacks. The presented modifications enable HoneydV6 to collect information about IP address, port number, and protocol type, as well as payload and timestamp of all established connections. In order to detect network scans, ICMP and ICMPv6 traffic is also logged in a database.

Figure 1 depicts the basic components of HoneydV6 (white boxes) and the malware detection extensions (blue boxes). Incoming network packets are captured by HoneydV6 using the network packet capture library *libpcap*.

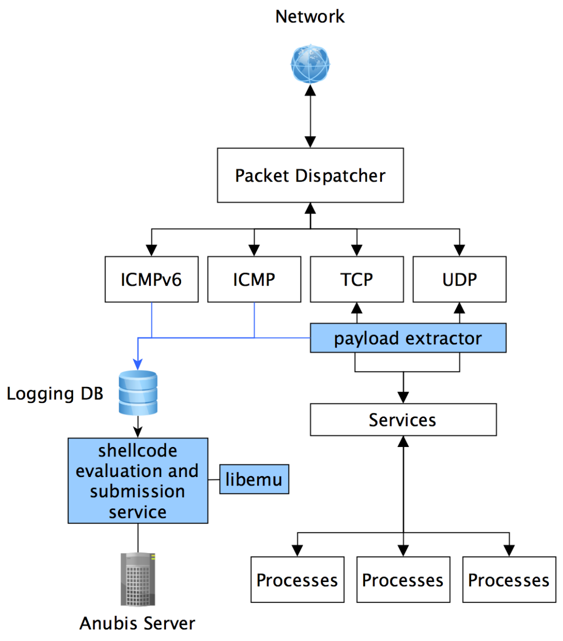

Figure 1: HoneydV6 with Malware Detection Components.

Depending on the packet type, the packet dispatcher passes the packets to a TCP, UDP, ICMP or ICMPv6 processor. ICMP and ICMPv6 packets are processed by an internal component, while TCP and UDP traffic may be forwarded to a configured service script (see Section 2).

A copy of the traffic that is transferred from an adversary to a service script is handed over to the shellcode emulation library *libemu*. This approach allows HoneydV6 to process the traffic as early as possible without any modifications to the service scripts.

The necessary modification to HoneydV6 are described in the next section.

## 4.1 Connecting Libemu to HoneydV6

The first step towards shellcode inspection is the localisation and extraction of shellcodes contained in the network traffic. In order to accomplished that, the shellcode detection library *libemu* needs access the entire honeypot traffic.

HoneydV6 uses the following functions to process TCP packets:

**tcp_recv_cb46** - This callback function is called by the packet dispatcher for incoming TCP packets. It checks whether HoneydV6 emulates the requested service and sends an ICMP or ICMPv6 error if not. HoneydV6 maintains a data structure which contains all established connections. For each incoming TCP packet, the honeypot looks up the corresponding connection entry of type *tcp_con* and writes the packet payload into an assigned file descriptor.

**tcp_new46** - If *tcp_recv_cb46* cannot find an existing connection entry for an incoming TCP packet, it needs to create a new connection entry by using this function.

**cmd_tcp_read and cmd_tcp_write** - HoneydV6 uses the library libevent to monitor read and write operations to the file descriptors that are assigned to a connection. As soon as something is written into the in- or output-stream of a connection, one of these two callback functions is called. Therefore, the entire TCP payload passes these functions which provides a convenient point to add our traffic inspection.

**tcp_free** - When a TCP connection is closed, *tcp_free* removes the connection entry from the connection data structure.

For each function listed above, there exists an UDP counterpart. HoneydV6 writes incoming traffic to the file descriptors of the corresponding service script without doing any further inspection. The honeypot does not keep a reference to the exchanged traffic. As soon as the payload is passed to a service script, it is out of scope of the honeypot.

If we want to inspect incoming traffic for malicious shellcode, we have to keep a reference to the exchanged payload which can be passed to *libemu*. Therefore, we extended the existing *tcp_con* and *udp_con* structures with a pointer to a shellcode buffer which gets initialised when a new connection entry is created. We modified *cmd_tcp_write* and *cmd_udp_write* to copy incoming data into the buffer before it is forwarded to the service emulation script. We restricted the maximum size of the buffer to a configurable value in order to avoid DoS attacks. Currently, the maximum payload size is set to a default value of 1024 bytes. The influence of the buffer size is investigated in Section 5.4. In case of larger payloads which exceed the buffer size, only the first 1024 bytes are stored and used for inspection.

It is important to note that malware profiling, which is done later using Anubis when *libemu* could detect a malware sample, may produce wrong results if the shellcode binary is cropped. If precise profiling results are required then the buffer size should be set to the maximum size of the expected shellcode binaries.

The inspection with *libemu* and Anubis requires a lot of time. For this reason we process the payload in a separate thread in a non-blocking manner. The *libemu* function `emu_getpc_check()` is used to identify malicious payloads. All payloads that are classified malicious are uploaded to Anubis for further analysis. The next section describes this process in detail.

## 4.2 Payload Inspection with Anubis

In contrast to Dionaea, HoneydV6 uses the advanced online malware analysis service Anubis instead of *libemu* to conduct malware profiling. Traffic that is classified malicious by *libemu* can automatically be uploaded to Anubis by an internal HoneydV6 process.

In order to use Anubis, we had to convert our malware samples into the executable Windows *.exe* file format. Furthermore, we needed to implement a HoneydV6 module which automatically uploads the generated binaries via HTTP to Anubis.

Due to performance reasons, we do not run a shellcode inspection for TCP and UDP traffic instantly. Instead, we extract the received payload and store it in a newly created SQLite logging database. A separate thread periodically inspects new binary samples stored in the database and triggers an upload to Anubis. Figure 2 depicts the structure of the new logging database.

The table *connection* holds information about all incoming connections. This includes start time and end time, source and destination address as well as the layer 3 protocol type. The *protocol_id* refers to an entry in the *icmp* or *tcp_udp* tables. All incoming ICMP and ICMPv6 messages are logged in the *icmp* table. Furthermore, we are logging source and

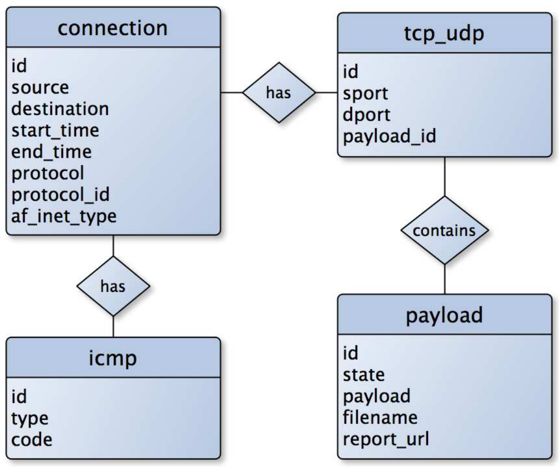

Figure 2: Logging database tables.

destination ports of UDP and TCP connections in the *tcp_udp* table. An entry in the *tcp_udp* table contains a *payload_id* which refers to payload stored in the *payload* table. Besides the actual payload, a payload table entry contains a report URL which points to the Anubis report. We also store the processed malware sample in the filesystem and use the filename-entry in the database to point to it, which allows later access and analysis.

Besides its capability to allow asynchronous malware processing, the database has great advantages over HoneydV6's old log file format. It is now possible to create sophisticated SQL queries to gather statistical information about accessed ports, addresses etc. Furthermore, the database provides a central payload store which is independent from capturing and logging capabilities of service scripts.

We used the Metasploit tool *msfencode* (Metasploit, nd) to transform the collected malware samples into the Windows *.exe* file format. Besides creating executable files, *msfencode* can be used to create malicious payloads because it supports various machine code modifications, e.g. removing null bytes or encrypting machine code. We configured *msfencode* to generate unencrypted x86 binaries to keep the inspection for Anubis as simple as possible.

In our experiments, the process of generating an *.exe* file took about 5 to 10 seconds on a 2,3 GHz Quad-Core Intel Core i7 with 16 GByte RAM using a solid-state drive (SSD). In order to ensure a non-blocking service on slower machines, we moved the file creation into a separate process, using the *fork* system call. The process communication is accomplished via pipes that are created using the *popen* system call.

We only process malware samples which have not yet been inspected by an earlier connection. This is achieved by generating an MD5 checksum for the sample that represents a folder used to store the sample on the filesystem. If a folder with the same name already exists on the filesystem, we assume that the malware sample has already been processed and we skip the inspection.

After the malware file has been successfully created, we use the network library *libcurl* to upload the file to Anubis. We attach the malware sample to an HTTP POST request which we send to the Anubis website. The server replies with a website containing the URL pointing to the inspection result. HoneydV6 extracts this URL and stores it in the logging-database. From there it can be accessed later to manually evaluate the inspection results.

## 5 EVALUATION

In order to validate our implementation, we compared the shellcode detection rate of the extended HoneydV6 to Dionaea's detection rate by sending a large amount of malicious traffic to both honeypots. Because both honeypots are using the same shellcode detection library (libemu), we expected an equal behaviour on both systems.

### 5.1 Generating Malicious Code Samples

We used the Metasploit Framework version 4.4.0-release (Metasploit, nd) to generate malicious code samples that we can send to both honeypots.

Such malicious code sample consists of an exploit, usually a specially crafted byte sequence which exploits a software vulnerability, and shellcode which should be placed on the target system. Metasploit contains many different exploits targeting various software vulnerabilities and allows it to combine various exploits with different shellcodes. Since we only want to test the shellcode detection capabilities of our approach, we only vary and send the shellcode binaries and skip the exploits.

The tools *msfpayload* and *msfencode* of the Metasploit Framework were used to create the test samples. We wrote a simple bash script which calls `msfpayload -l` to create a file containing a list of all available Metasploit shellcodes (274 at the time of writing). For each element in the list, a second bash script iteratively calls *msfpayload* and *msfencode* to create shellcode binaries as follows:

```
msfpayload $1 R | msfencode -a x86
  -e generic/none -t raw >
  $outfile 2>/dev/null
```

Table 1: Distribution of payload sizes for the transmitted shellcode samples.

| **Size in bytes** | 17-32 | 33-64 | 65-128 | 129-256 | 257-512 | 513-1024 | 1025-2048 | 2049-4096 | 4097-8192 |
|---|---|---|---|---|---|---|---|---|---|
| **#Samples** | 2 | 3 | 39 | 28 | 24 | 3 | 4 | 4 | 2 |

In our case, *msfencode* takes the shellcode output of *msfpayload* and converts it into an x86 binary. We used the command line parameters `-e` and `-t` to retrieve an unencrypted raw binary file.

For a number of entries in the generated payload list, *msfpayload* needed non-default custom parameters which are not provided by our script. We skipped these entries and obtained 107 valid shellcode binaries that can be used to evaluate the honeypot detection capabilities.

## 5.2 Honeypot Setup

We adapted the default Dionaea configuration to accept HTTP connections by adding the entries `http` and `httpd` to the section `processors/filteremu/config/allow` in the default Dionaea configuration *dionaea.conf*. In order to enable a verbose logging, we started Dionaea with the parameters `"-l all,debug -L '*'"`. Dionaea stores all connection information in an SQLite database called *logsql.sqlite*. The database contains the tables *connections* and *emu_profiles* to store connection details and malware samples.

HoneydV6 was installed using *./configure* and *make* from the GNU Autotools. We configured HoneydV6 to simulate a single virtual IPv6 host running a web server. The newly implemented shellcode detection was enabled by adding the necessary *dbfile*, *shellcodedir* and *submit* entries to Honeyd's configuration file. Detected shellcode is stored in the database file *honeyd.db*.

## 5.3 Shellcode Transmission

We used the command line tool *netcat* (Netcat, nd) to send our malware samples via HTTP GET requests to HoneydV6 and Dionaea. A script iterates over the list of the previously generated shellcode binaries and creates an HTTP GET request for each binary. The script uses a different TCP source port, starting from 5000, for each connection request to allow the correlation of the logged requests. The sending process was done consecutively in two steps. In the first step, we sent all traffic to Dionaea before we repeated this process for HoneydV6 in a second step.

After finishing the sending process we inspected the Dionaea and HoneydV6 databases to determine which connections had been identified to contain malicious traffic.

Table 2: HoneydV6 detection rate for different shellcode buffer sizes.

| **Buffer Size** | 16 | 32 | 64 | 128 | 256 - 8192 |
|---|---|---|---|---|---|
| **#Detected** | 0 | 12 | 23 | 25 | 26 |

## 5.4 Results

We sent 107 different malware samples to Dionaea and HoneydV6. Since all of the metasploit samples are considered to be malicious, we can not observe false positives. The samples are of different size as shown in Table 1. For the first test run, the shellcode buffer size of HoneydV6 was set to the default size of 1024 Bytes (see Section 4.1). Figure 3 shows the number of detected and undetected samples for both honeypots.

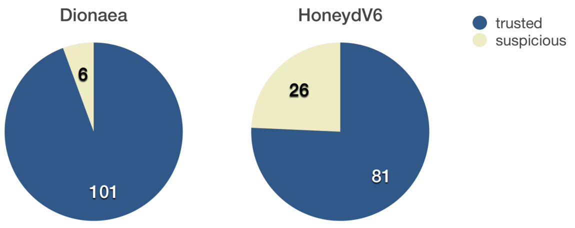

Figure 3: Detection rate of HoneydV6 and Dionaea.

Dionaea creates a database entry for 6 of the 107 shellcodes (about 6 %). HoneydV6 was able to detect 26 malicious samples (about 25%). All shellcodes detected by Dionaea were also detected by HoneydV6.

This different result is surprising, since both honeypots are using the same shellcode detection library *libemu*. Hence, both honeypots are using *libemu's* `emu_getpc_check` function to locate GetPC instructions and to decide whether a byte stream is malicious or not.

The only difference is that HoneydV6 logs all samples with a GetPC occurrence into its database. Dionaea, however, uses further malware profiling functions of *libemu* to create a complete malware profile. A database entry will only be created, when the profile creation could be completed successfully. In case of a profile creation failure, no database entry will be created even if a GetPC instruction could be located. In contrast, HoneydV6 detects as many potential attacks as possible and uses the advanced analysis function of Anubis for the further examination of the potential malware.

Further, we investigated the influence of the size of the shellcode buffer on the detection rate. Therefore, we varied the buffer size from 16 Bytes up to 8192 Bytes. Table 2 shows the number of detected samples for different shellcode buffer sizes. A buffer size of at least 31 bytes was needed to detect the first sample. Increasing the buffer size up to 32 bytes was sufficient to detect 12 shellcode samples. Even with bigger buffer sizes up to 8192 Bytes, *libemu* detected at most 26 malware samples.

## 6 SUMMARY

Attack detection in IPv6 networks is still in an early stage. Currently, there are only two general purpose low-interaction honeypots available: Dionaea and HoneydV6.

Shellcode detection in low-interaction honeypots is an important feature for network health monitoring and zero-day attack detection. Dionaea already provides an integrated shellcode detection mechanism. However, since Dionaea can not be usefully applied to very large IPv6 networks, we decided to extend HoneydV6 with a similar shellcode detection component.

We achieved this goal by integrating the open-source detection library *libemu* into HoneydV6. Beyond the detection and extraction of shellcode binaries, we integrated the online malware analysis service Anubis into HoneydV6 to allow an advanced profiling of malware samples.

In our evaluation, we compared the shellcode detection rate of Dionaea and HoneydV6. Since both honeypots are using the same shellcode detection library *libemu*, we expected a similar detection rate for both honeypots. We found that Dionaea detected about 6%, while HoneydV6 can detect about 25% of all our generated malware samples. This difference in the detection rates can be explained by an additional profiling mechanism only present in Dionaea, which is used as an additional alert filter criteria.

What is noticeable, though, is the general low detection rate for shellcode attacks. This shows that there is still a lack of more advanced open-source shellcode detection libraries that could mitigate these kind of network attacks.

Still it is not possible to inspect traffic that is encrypted on service layer. This problem can be partly solved by extending HoneydV6 with an internal TLS implementation so that HoneydV6 is responsible for encrypting and decrypting the communication and therefore keeps control also over the encrypted communication.

## REFERENCES

Anubis (nd). Anubis: Analyzing Unknown Binaries. Available from: http://anubis.iseclab.org.

Baecher, P. and Koetter, M. (nd). libemu – x86 Shellcode Emulation. Available from: http://libemu.carnivore.it/.

Beale, J., Baker, A. R., Esler, J., and Northcutt, S. (2007). *Snort: IDS and IPS toolkit*. Jay Beale's open source security series. Syngress.

Dionaea (nd). dionaea catches bugs. Available from: http://dionaea.carnivore.it/.

Fratantonio, Y., Kruegel, C., and Vigna, G. (2011). Shellzer: A tool for the dynamic analysis of malicious shellcode. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 61–80, Berlin, Heidelberg. Springer-Verlag.

Honeyd (nd). Honeyd Virtual Honeypot. Available from: http://www.honeyd.org.

kippo (nd). kippo - SSH Honeypot. Available from: https://code.google.com/p/kippo/.

Malwr (nd). Malwr - Malware Analysis by Cuckoo Sandbox. Available from: https://malwr.com.

McCanne, S. and Jacobson, V. (nd). tcpdump & libpcap. Available from: http://www.tcpdump.org/.

Metasploit (nd). Metasploit: Penetration Testing Software. Available from: http://www.metasploit.com.

Netcat (nd). The GNU Netcat project. Available from: http://netcat.sourceforge.net.

Polychronakis, M., Anagnostakis, K. G., and Markatos, E. P. (2006). Network level polymorphic shellcode detection using emulation. In *Proceedings of the Third International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA'06, pages 54–73, Berlin, Heidelberg. Springer-Verlag.

Sandbox, C. (nd). Cuckoo Sandbox. Available from: http://www.cuckoosandbox.org.

Schindler, S., Schnor, B., Kiertscher, S., Scheffler, T., and Zack, E. (2013). HoneydV6: A low-interaction IPv6 honeypot. In *Proc. of the 10th International Conference on Security and Cryptography (SECRYPT 2013)*, Reykjavik, Iceland.

Spitzner, L. (2002). *Honeypots: Tracking Hackers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Ször, P. and Ferrie, P. (2001). Hunting for metamorphic. In *In Virus Bulletin Conference*, pages 123–144.