

Attack redirection in Honeypots



**Information Assurance Laboratory
Center for Risk of Reliability
University of Maryland**

Julien VEHENT – April/September 2007
Master Management de la Sécurité des Systèmes Industriels
et des Systèmes d'Information

I would like to deeply thank my adviser Doctor Michel Cukier, Assistant Professor in the Department of Mechanical Engineering and co-director of the Information Assurance Laboratory, and Robin Berthier, Ph.D student in Reliability Engineering. This internship would not have been doable without their supports and advises.

I would also like to salute all the students, researchers and friends I met during the last 6 months. The regular misunderstandings due to our various and awkward accents have made this stay really enjoying.

RÉSUMÉ

La nécessité d'observer, d'analyser et de comprendre les comportements frauduleux est bien connue dans le monde de la gestion des risques : la protection d'un environnement ne peut se faire correctement sans la compréhension des risques qui l'affaiblissent. Concernant les risques liés aux piratages informatiques, les systèmes de Honeypots fournissent probablement le meilleur moyen d'analyse.

L'université du Maryland, et en particulier l'*Information Assurance Laboratory*, travaille sur plusieurs projets et méthodes liés aux réseaux Honeypots. A cause de l'important trafic frauduleux qui circule en permanence sur le réseau Internet, l'analyse des informations que reçoit un Honeypot est rendue difficile. Je vais présenter dans ce rapport le travail que j'ai effectué sur l'analyse, la spécification et la création d'une méthode permettant de sélectionner des attaques *intéressantes* sur lesquelles une analyse approfondie pourra ensuite être effectuée.

ABSTRACT

The need to watch, analyze and understand malicious behaviors is a well known requirement in risk management : the protection of an environment can't be done without the understanding of the risks that weaken it. Regarding computer systems risks, honeypots provide certainly the most efficient analytic system.

The University of Maryland, and more precisely the *Information Assurance Laboratory*, works on several projects and methods linked to Honeypots. Because of the important malicious traffic on the Internet, the analysis of the information that a Honeypot receives is difficult. I will present in this report the work I have done on the analysis, the specification and the creation of a method that allows the selection of interesting attacks on which a detailed analysis could be done later.

Table of contents



Introduction	6
The University of Maryland.....	7
Honeypot theory.....	9
1.1 Why Honeypots ?.....	9
1.1.1 The HoneyNet Project.....	10
1.2 Honeypot Types.....	11
1.2.1 Low-Interaction Honeypots.....	11
1.2.2 High-Interaction Honeypots.....	12
1.2.3 Hybrid Honeypots.....	12
1.3 A New Hybrid Honeypot.....	13
1.3.1 Attack selection.....	14
1.3.2 What is an interesting attack ?.....	15
1.3.3 Unknown attacks selection techniques.....	16
1.3.4 Connection sequences strings analysis.....	17
1.3.5 The interaction limit problem.....	18
1.3.6 Conclusion.....	19
The Argusproxy project.....	20
2.1 The Argusnet Network.....	20
2.1.1 Argusnet experiments.....	22
2.2 ArgusProxy : software engineering.....	22
2.2.1 Requirements.....	22
2.2.2 Activity Diagram.....	23
2.2.3 Deployment Diagram.....	25
2.2.4 Modules interactions.....	26
2.2.4.1 Redirector.....	26
2.2.4.2 Decision Engine.....	27
2.2.4.3 Shared memory.....	28
2.2.5 Documentation.....	29
2.3 Timetable.....	30
2.3.1 Gantt diagram.....	30
2.4 Technical issues.....	31
Tests & future work.....	34
Conclusion.....	35
References.....	36
Figures.....	37
Glossary.....	38
Appendix.....	39
A ⇒ Gantt	39
B ⇒ Connections sequences analysis.....	40
C ⇒ ArgusProxy Network Engine.....	47
D ⇒ ArgusProxy source code.....	50

Introduction



Like the old proverb says : *“know your enemies and know yourself, you will win a hundred times in a hundred of battles”*. This advise is still applicable to the network war that opposes companies, that have computer networks to protect, to hackers, who want to use these very networks for their malicious activities. In this context, a honeypot is a lure that is useful to make the enemy come. Studying honeypots means to study the way an intruder think and the way it will attempt to break into a computer network. During the last ten years, it has been the most used method by anybody who wants to improve the techniques to secure computer resources. Know you enemy, and you will know how to protect yourself from it.

During the last five months, I have worked on a new kind of honeypot architecture at the University of Maryland. Classical honeypots have limitations due to their designs. Those limitations falls in two categories : breadth of coverage and limit of the interaction. The purpose of my internship was to create an innovative architecture that can improve those two limitations. To do so, I had to study the many different types of honeypots, tests them, analyze them and criticize them.

The first chapter of this report explains what honeypots are, their history and the way they are used. Hybrid honeypots will be introduced in the second part of this first chapter. I will also detail some experiments I realized on the analysis of network attacks.

The second chapter deals with the Argusnet architecture set up at the University of Maryland. Then, I will present the ArgusProxy project, its requirements and how I specified it using the Unified Modeling Language.

Finally, in the third chapter, I will detail the current state of ArgusProxy and what will be done until the end of my internship.

An overview of the progress of this internship is available in Appendix A as a Gantt diagram.

The University of Maryland

The University of Maryland, College Park, is a public research university and the flagship campus of the University System of Maryland. Founded in 1856, it is a 1,200 acre suburban campus composed of 270



buildings and located eight miles at the north of Washington, D.C., and 35 miles from Baltimore.

The University consists of around 35,000 students and 6,000 professors, in various majors, including Agriculture and Natural Resources, Architecture, Arts and Humanities, Computer, Math, and Physical Sciences, Education, Journalism, Life Science.

History

The University of Maryland was first an Agricultural College founded in 1856. Only 34 students were enrolled in 1859. This number grew when President Lincoln signed the Morrill Land Grant Act providing federal support for state colleges to teach agriculture, mechanical arts and military tactics. The first women students enrolled in 1916. In 1917 Albert F. Woods



(1866 - 1948) was named president and created seven schools, each with its own dean : Agriculture, Engineering, Arts and Sciences, Chemistry, Education, Home Economics and the graduate school.



In 1932, Testudo the turtle became the official Maryland mascot. Its bronze version stands proudly in front of the McKeldin library, to touch its head is a sign of luck.

In 2006, the revenue of the university reached \$1,244,892,829. Today, the University of Maryland is one of the nation's Top 20 public research universities.

Information Assurance Laboratory

The Information Assurance Laboratory at the University of Maryland is co-directed by Dr. Michel Cukier and Dr. Carol S. Smidts.

The laboratory focuses on four research threads:

1. Probabilistic Risk Assessment;
2. Modeling;
3. Bug and Vulnerability Identification Tools;
4. Empirical Studies.



Among others, these research activities are funded by National Science Foundation (NSF), National Aeronautics and Space Administration (NASA), Nuclear Regulatory Commission (NRC), National Security Agency (NSA), Defense Advanced Research Projects Agency (DARPA), Teradyne, Texas Instruments Inc., Tedco, and Maryland Industrial Partnerships (MIPS).

The team of Michel Cukier

Michel Cukier is an Assistant Professor in the Department of Mechanical Engineering. He received a physics engineering degree from the Free University of Brussels, Belgium, in 1991, and the Doctor in computer science from the National Polytechnic Institute of Toulouse, France, in 1996.

His current research interests include security evaluation, intrusion tolerance, distributed system validation and fault injection with a team of Ph.D, graduate and undergraduate students.



The Glenn Martin Hall, building of the Department of Mechanical Engineering

I took part of the Argusnet project, developed by Susmit Panjwani, Stephanie Tan and Keith Jarrin, and mainly maintained and improved by Robin Berthier as part of his Ph.D. The purpose of this project is to develop an architecture to monitor attackers and collect attack data.

Honey-pot theory

« It is said that if you know your enemies and know yourself, you will not be imperiled in a hundred battles;
if you do not know your enemies but do know yourself, you will win one and lose one;
if you do not know your enemies nor yourself, you will be imperiled in every single battle. »

Sun Tzu, *The Art of War*

Historically, the term “Honey-pot” is not issued from the world of computer security. It's a widely used concept in the spying world that refer to a set of techniques used to grab relevant and secret information from a target by attracting it in a trap (usually a sexual trap, with the help of drugs).

In computer sciences, the term find its roots in the 80's with a book : "The Cuckoo's Egg" from Clifford Stoll. As a system administrator at the Lawrence Berkeley National Laboratory, Stoll has, during almost three years, followed and recorded the behavior of an intruder who was using one of his Unix systems to relay attacks and target several governmental agencies. To locate him, Stoll has spent ten months recording every action on the system and was finally able to isolate the source of the attack. After a while, the intruder, Markus Hess, has been arrested. He was working for the KGB. “The Cuckoo's Egg” relates this story and has built the foundations of the attacks observation techniques.

1.1 Why Honey-pots ?

The first step to understand honey-pots is to define what a honey-pot is. This can be harder than it sounds because, unlike firewalls or Intrusion Detection Systems (IDS), a honey-pot does not solve a specific problem. Instead, honey-pots are highly flexible tools that come in many shapes. They can do everything from detecting encrypted attacks in IPv6 networks to capturing the latest on-line credit card fraud. This flexibility made honey-pots the cornerstone of the research in computer security.

Lance Spitzner, a senior security architect for Sun Microsystems, Inc., and an acknowledged authority

in security and honeypot research, gives the following definition of Honeypots :

"A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource." [6]

The information provided by a honeypot is extremely interesting for anyone whose interest lies in information security. It could be security software companies (anti-virus editors are taking a huge interest in honeypot techniques), Internet providers that need to oversize their networks because of automated attacks that consume their resources (to detect and block those attacks is very interesting for them) or simply companies that consider the understanding of the threats that aim their information systems as important.

But the most important work around honeypots is done by the research community. Some projects, like the HoneyNet project [5], are trying to centralize the efforts and share the techniques.

1.1.1 The HoneyNet Project

"The HoneyNet Project is a non-profit volunteer, research organization dedicated to improving the security of the Internet at no cost to the public. All of our work is released as and we are firmly committed to the ideals of OpenSource. Our goal, simply put, is to make a difference." [5]

The HoneyNet project has been started by Lance Spitzner. This project has three main goals :

1. Awareness : Explaining the threats and vulnerabilities that exist in the Internet today. The goal of the HoneyNet Project is to provide information so people can better understand they are a target, and understand the basic measures they can take to mitigate these threats.
2. Information : They provide details to better secure and defend resources. Historically, information about attackers has been limited to the tools they use. The HoneyNet Project provides critical additional information, such as their motives in attacking, how they communicate, when they attack systems and their actions after compromising a system.
3. Tools : For organizations interested in continuing their own research about cyber threats, the HoneyNet Project provides the tools and techniques they have developed.

Since 2002, the HoneyNet Project has been the central meeting point of all the honeypots researchers

over the world. The community grows in many country but in France, the French HoneyNet Project ended in July 2007.

1.2 HoneyPot Types

Conceptually, almost all honeypots are working the same way. They are a resource that has no authorized activity. They do not have any production value. Theoretically, a honeypot should see no traffic because it has no legitimate activity. This means any interaction with a honeypot is most likely unauthorized or malicious activity. Any connection attempt to a honeypot is more likely a probe, attack, or compromise. While this concept sounds very simple, it is this very simplicity that gives honeypots their advantages.

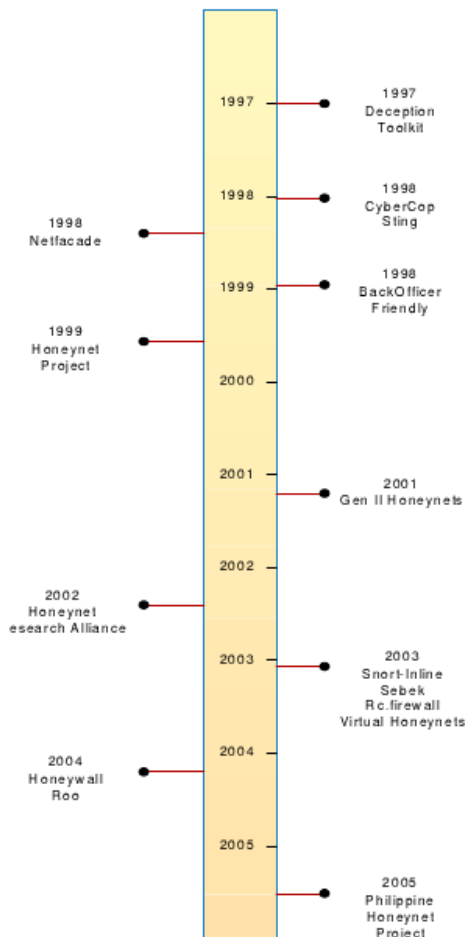


Figure 1: honeypots projects since 1997

1.2.1 Low-Interaction Honeypots

In 1997, the first usable tool allowing to fake computer's vulnerabilities is released. The Deception Kit [11] is a set a Perl scripts that re-create the basic behavior of several well-known daemons. When an intruder seeks for a particular vulnerability and reach a computer that hosts The Deception Kit, he has the vision of an exploitable system and launches his attack. Of course, nothing happens on the system because the kit is just able to reply pre-recorded answers but this is enough to record the communication and learn from it. This gives the administrators a delay to fix their systems. The idea of using a set of scripts to fake a service, limiting the number of possible responses, provides only a low level of interaction between the intruder and the faked system, so we use to call this kind of architecture *Low-Interaction Honeypot*.

1.2.2 *High-Interaction Honeypots*

During the following years, honeypots will become more popular and, by the way, more sophisticated. The use of real systems, dedicated servers set up to be attacked, will improve the gathering of information. In such systems, the interaction is not limited to pre-recorded answers so the intruder goes more deeply and, because everything is recorded, monitored and studied, gives more relevant information on the behavior of the attacker. These systems are called *High-Interaction Honeypots*.

Those two approaches are different and complementary. When *Low-Interaction Honeypots* allow to cover a wide range of connections with few information on each, *High-Interaction Honeypots* provide more accurate information on a specific kind of attack but need to be re-initialized after each attack. Moreover, honeypots are no different than other technologies, they also have an associated risk. Specifically, honeypots have the risk of being taken over by an attacker and being used to attack other systems. This risk is small with *Low-Interaction Honeypots* but high with *High-Interaction Honeypots*. In both cases, it's important to take care about it and assess the risk before setting up a honeypot.

1.2.3 *Hybrid Honeypots*

The idea of combining different types of honeypots has occurred a few years ago. The temptation of using the coverage capabilities of *Low-Interaction Honeypots* and the interaction capabilities of *High-Interaction Honeypots* has driven the research community on a new field of research : *Hybrid Honeypots*.

One of the first work on Hybrid Honeypots has been presented in the paper : « *A Hybrid Honeypot Architecture for Scalable Network Monitoring* », written by the University of Michigan and Google [1]. This new architecture uses a *Low-Interaction Honeypot* as a front end, a *High-Interaction Honeypot* as a back end and two other new components : a **redirection engine** and a **checksum based decision engine**. The goal of the first one is to redirect connections like, what a NAT proxy does and the goal of the second one is to detect new attacks. An incoming connection is handled by the *Low-Interaction*

Honeypot, then the decision engine processes every packet of the connection and, if the connection is interesting enough, sends a redirection order to the redirection engine.

This architecture gives the tremendous advantages of the selection : by computing a checksum on every incoming packets, they limited the number of connections that the *High-Interaction Honeypot* receives and, by the way, deeply decreased the analysis time.

1.3 A New Hybrid Honeypot

The first assumption in *Hybrid Honeypots* is redirecting an active connection from a *Low-Interaction Honeypot* to its new target. This is difficult because the attacker must not detect the redirection and because, in the case of the TCP protocol, a connection is not designed to be broken.

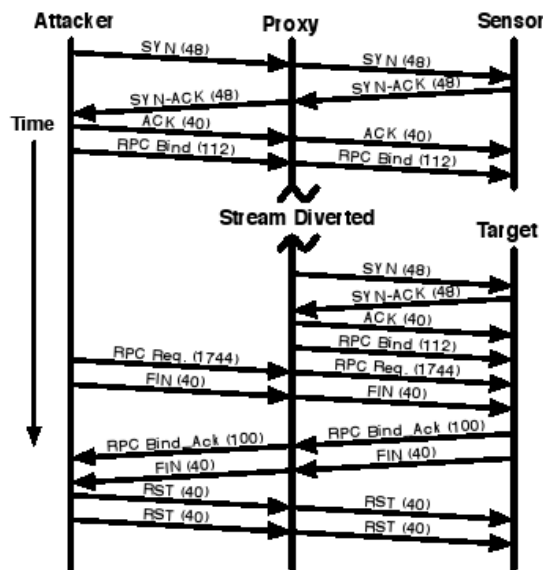


Figure 2: TCP redirection in hybrid honeypot

Figure 2 shows the redirection process like it's presented by the University of Michigan in their work [1]. The Sensor is the *Low-Interaction Honeypot* and the Target is the *High-Interaction Honeypot*.

When the decision to redirect the communication is taken, a new TCP connection is initialized with the Target and the redirection engine replays the packets of the recorded connection. Then, the packets received from the attacker are redirected to the Target and not to the Sensor anymore.

This is a very interesting design but, like I will discover later, it breaks the principle of robustness on which most of the Internet protocols are based. Dynamically redirecting a connection is a difficult problem to solve. When it's quite easy to do with a basic protocol like UDP, a connection oriented protocol like TCP is definitely not designed to be dynamically redirected (sequences numbers, ack packets and so are adding complexity) and a ciphered communication is almost impossible to redirect.

Moreover, there's some basic assumption to take care about (see also [2]) :

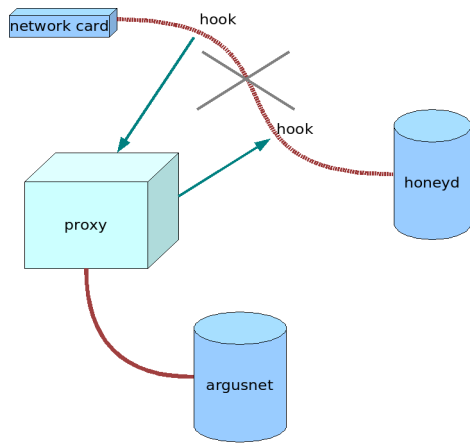


Figure 3: basic hybrid honeypot architecture

→ **Invisibility** : the activity of the redirection engine must be completely transparent to the attacker. This means that the redirected connections must not carry any fingerprint of any manipulation;

→ **Scalability** : even if the number of connections redirected is low, the redirection engine must be designed to handle a high number of connections at the same time, especially when redirecting a connection. If the latency is too high, a skilled attacker could detect it and cut the connection;

→ **Recording** : in order to replay and study attacks later, connections must be recorded. This is very consuming in computing times and memory usage.

Figure 3 shows a basic example of the *Hybrid Honeypot* I have designed during the first days of my internship.

1.3.1 Attack selection

In order to select and redirect attacks, we need to define the concept of attack extremely precisely. When considering network and computer attacks, we can isolate three main models of behaviors classified by their interactivity levels :

- ◆ **One packet attack**

A single datagram (UDP or ICMP most of the time) carries the entire attack. UDP and ICMP protocols do not require the establishment of a connection (handshake) so the first packet received is generally the entire attack.

example : SqlSlammer, a 376 bytes payload in a UDP packet that aims SQL Server 2000.

The detection of this kind of attack doesn't require any interaction, a simple sensor on the network can receive it.

- ◆ **Connection oriented attack**

In this model, a response from the targeted host is required for the attack to succeed. This is

typically the behavior of port scanners : they attempt to connect to a list of TCP ports using synchronization packets and mark as open a port that answers with a synchronization acknowledgment packet.

Because of the absence of payload, faking a basic interactivity is very easy to do by simply binding a port and waiting for connections.

◆ **Application dependent attack**

This targets a high level protocol, usually over the operating system layer. In this model, the attacker needs to establish a complex communication with the target, involving, sometimes, the use of several upper level protocols and fragmented packets.

This means that, to detect these attacks, we need to fake a real service with a most important interactivity level.

1.3.2 *What is an interesting attack ?*

Before connecting a honeypot system to the Internet, it's very important to define what we want to study. The persistent noise on the Internet consists of many different kinds of worms, lost packets, scans and so on. A honeypot will collect all of them. The interesting thing with *Hybrid Honeypots* is that they give the ability to select the information we want to study.

Basically, an interesting attack could be whatever we want. But, regarding the concept of attack, we can reduce this list to the following:

1. An interesting attack belongs to one of the three categories of attack : one packet attack, connection oriented attack and application dependent attack. This means that the selection engine should be configured to select an attack included in these categories.
2. The second parameter is the depth of the analysis : how far does the engine go when analyzing the packet/connection. For example, a security software editor will be interested by an extremely wide coverage capability in order to detect very quickly an unknown attack. In this case, a checksum inspection is efficient enough. A research laboratory could be more interested by selecting attacks that aim a very specific version of the NetBios protocol in order to perform a

study on the new Windows Vista implementation of this protocol. Here, a more sophisticated engine which could include an IDS, a checksum database and so on, should be more relevant.

With those two parameters, the attack level and the analysis depth, we can determine what an interesting attack is for a specific study and build a specific decision engine for it.

This adds a requirement for the decision engine : it has to be as modular as possible, regarding these two parameters.

1.3.3 *Unknown attacks selection techniques*

Considering the initial postulate that we want to build a general decision engine in order to detect unknown attacks, and redirect them from the *Low-Interaction Honeypot* to the *High-Interaction Honeypot*, I will now present a set of techniques to determine if we have already seen or if we already know the behavior of an incoming connection.

1. One of the first techniques easily usable is the **checksum check** : set up a database with checksums of known payloads and compare an incoming payload with the entries of the database. This is quite simple to implement but not really reliable because :
 - ◆ An attack can start with a known payload and then switch to others techniques just after;
 - ◆ A same attack can use several variants, the checksum will not detect them as similar.

The checksum method is efficient when the goal of the study is to detect new worms (including variants) and automated vulnerability scans (Metasploit, Nessus, ...).

2. A solution based on tools could be to use an IDS as a **negative detection engine**. If a communication is not detected as a known attack, we can consider it as an unknown attack and redirect it. This works because, in a honeypot network, the whole traffic is attacks.

IDSs are using powerful techniques to detect attacks, they can also provide useful information regarding the connection (type of the attack, operating system targeted and so on...).

3. While I was working on r-contiguous string inspection [3] and a comparative analysis of vulnerability check and vulnerability exploit, I found that a connection can also be represented by its sequence. This is the *connection sequences strings* method I will explain now.

1.3.4 Connection sequences strings analysis

A packet sequence is a string created using the flags (in the case of TCP), the size and direction of each packets that compose a communication.

```
{ 011000 191 O }011000143I011000230O011000455I011000360O01100093I01100097O
```

A typical sequence string (tcp 445). It's composed of, for each packet, 6 tcp flags (in purple), a size in bytes (in green) and a direction (O = coming from outside, I from inside, in red). The string is composed of all the packets of the communication.

By comparing the sequence of an incoming communication with the recorded sequences, it's possible to regroup connections. I have made some experiments and the results seems to prove that sequence string inspection is relevant (Figure 4).

packets	15722
connections	1850
avg pkt/con	8,5

nb of identical sequences	sequence string	%
257	011000191O011000143I011000230O011000455I011000360O01100093I01100097O01100093I	13,89%
224	011000191O011000143I011000230O011000455I011000368O01100093I01100097O01100093I	12,11%
219	011000191O011000143I011000230O011000455I011000384O01100093I01100097O01100093I	11,84%
135	011000191O011000143I011000230O011000455I011000376O01100093I01100097O01100093I	7,30%
113	011000191O011000143I011000290O011000455I011000424O01100093I	6,11%
108	011000191O011000143I011000230O011000455I011000396O01100093I01100097O01100093I	5,84%
102	011000191O011000143I011000290O011000455I011000428O01100093I	5,51%
66	011000191O011000143I011000222O011000455I011000276O01100093I	3,57%
61	011000191O011000143I011000230O011000455I011000400O01100093I01100097O01100093I	3,30%
60	011000191O011000143I011000230O011000455I011000412O01100093I01100097O01100093I	3,24%
56	011000191O011000143I011000290O011000455I011000440O01100093I	3,03%
53	011000191O011000143I011000230O011000455I011000366O01100093I01100097O01100093I	2,86%
39	011000191O011000143I011000290O011000455I011000444O01100093I	2,11%
30	011000191O011000143I011000260O011000455I011000406O01100093I	1,62%
30	011000191O011000143I011000230O011000455I011000382O01100093I01100097O01100093I	1,62%
29	011000191O011000143I011000290O011000455I011000434O01100093I	1,57%
27	011000191O011000143I011000230O011000455I011000380O01100093I01100097O01100093I	1,46%
24	011000191O011000143I011000290O011000455I011000450O01100093I	1,30%
20	011000191O011000143I	1,08%
15	011000203O011000155I011000234O011000467I011000288O011000105I	0,81%

90,16%

[...]		
nb of unique sequences	110	5,95%

Figure 4: sequence string inspection results for TCP connections on port 445

Those results show that more than 90% of the connections are seen more than one time. But they also show that most of the connection strings are very similar, only a small difference in the sizes seems to differentiate them. Thus, I tried to find the “neighbors” of the most seen sequence by using a growing variance in packets size (Figure 5). This graph shows that by introducing a variation of 15% in packets size, we can match almost 6 times more sequences. The order of the packets in the communications are identical, only the sizes change. To confirm those results, I took four sequences and a variance of 20%,

and I searched for their neighbors in nine months of malicious activity. The results presented in Figure 6 show that more than 80% of the connections are similar.

This method of classification can help to detect and select unknown attacks. But this technique can also return a high number of false positive results. That should be evaluated. A more detailed document about this work is available in Appendix B.

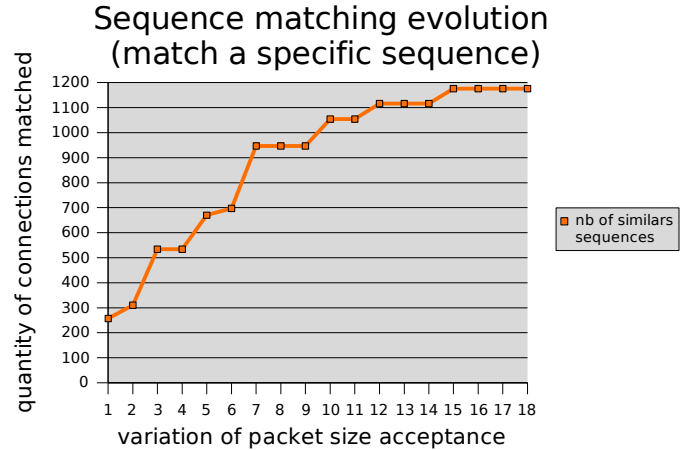


Figure 5: connection sequence matching using a variance in packet size

dump	nb of matched sequences	nb of sequences on port 445	
609	4535	7329	61,88
610	2031	2601	78,09
611	3771	4307	87,56
612	6104	6794	89,84
701	518	744	69,62
702	3280	4771	68,75
703	9304	10142	91,74
704	10176	11528	88,27
705	1208	1398	86,41
		average	80,24

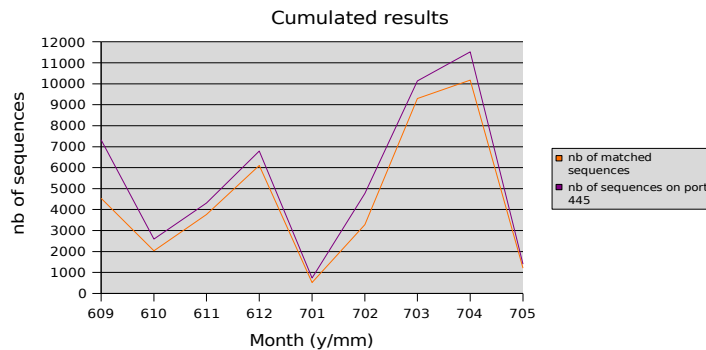


Figure 6: cumulated sequences inspection results

1.3.5 The interaction limit problem

When dealing with *low-interaction honeypots*, an important issue is their interaction limit. While I was working on the differences between vulnerability checks and exploits, I discovered that a software like HoneyD (probably the most popular *low-interaction honeypot*) stops to answer to an attacker when its interaction limit is reached. This means that the honeypot will acknowledge the incoming packets but

will not provide more information nor cut the connection (Figure 7).

A skilled attacker will detect this behavior and suspect that he's actually targetting a honeypot, and then probably cut the communication. So, we need a mechanism to detect when a *low-interaction honeypot* has reached its interaction limit and then intercept the next packet while asking the decision engine to make a decision immediately. This is what we call the interaction limit problem.

Figure 7: interaction limit with Honeyd faking a Windows 2000 SMTP daemon

```

Source      ->Destination  Proto.  Info
128.8.37.111->128.8.37.120  TCP     44488 > smtp [SYN] Seq=0 Ack=0[...]
128.8.37.120->128.8.37.111  TCP     smtp > 44488 [SYN, ACK] Seq=0 Ack=1[...]
128.8.37.111->128.8.37.120  TCP     44488 > smtp [ACK] Seq=1 Ack=1 [...]
128.8.37.120->128.8.37.111 SMTP Response: 220 win2kpro Microsoft ESMTP MAIL [...]
128.8.37.111->128.8.37.120 TCP 44488 > smtp [ACK] Seq=1 Ack=108 [...]
128.8.37.111->128.8.37.120 SMTP    Command: ETRN test.nessus.orgr
128.8.37.120->128.8.37.111 TCP     smtp > 44488 [ACK] Seq=108 Ack=23 [...]
128.8.37.111->128.8.37.120 SMTP    Command: ETRN test.nessus.orgr
128.8.37.120->128.8.37.111 TCP     smtp > 44488 [ACK] Seq=108 Ack=45 [...]
128.8.37.111->128.8.37.120 SMTP    Command: ETRN test.nessus.orgr
128.8.37.120->128.8.37.111 TCP     smtp > 44488 [ACK] Seq=108 Ack=67 [...]

```

(in gray, the TCP handshake, in bold the SMTP banner and then, the 3 same request sent by the attacker).

1.3.6 Conclusion

We can summarize the Honeypot tools as follow : *Low-Interaction Honeypot* provides scalability at the cost of the depth of the information gathered, *High-Interaction Honeypot* provides very detailed information but need to be re-initialized after each attack. Between those two stands the *Hybrid Honeypot*. It mixes the scalability of *Low-Interaction Honeypot* with the depth of *High-Interaction Honeypot*. *Hybrid Honeypot* relies on two engines, one to select interesting attack, the Decision Engine, and one to redirect those attacks to a target of choice, the Redirection Engine. By combining these techniques, *Hybrid Honeypot* provides an efficient method to filter attacks in order to focus only on specific ones, and thus reduce the analysis time and speed up the reactivity.

With this knowledge, I started to work on a Hybrid Honeypot software engineering project called ArgusProxy. The work I did on this project is described in the next chapter.

The Argusproxy project

« *Visualize this thing that you want, see it, feel it, believe in it. Make your mental blue print, and begin to build.* »

Robert Collier

2.1 The Argusnet Network

ArgusNet is an architecture to run live experiments in the field of system and network security. The goal of this architecture is to easily and securely deploy targets for attackers, such as honeypots, in order to collect live attack data and monitor experiments.

ArgusNet was initially developed by Susmit Panjwani, Stephanie Tan, Keith M. Jarrin at the end of 2004. Then in January 2007, an enhanced version of the architecture was built by Robin Berthier and Daniel Ramsbrock to face new challenges, such as better protection for the management network, stronger reliability for data collections, and easier solutions to backup and update the architecture and experiments. The architecture is made of two main parts (see Network 1):

1. **The honeypot network:** where all the experiments are deployed.
2. **The management network:** where all the data collection, data monitoring and administration take place.

The strict separation between these two networks allows operators to gather data and closely monitor the experiments without being detected by attackers. The three major pieces of the architecture are:

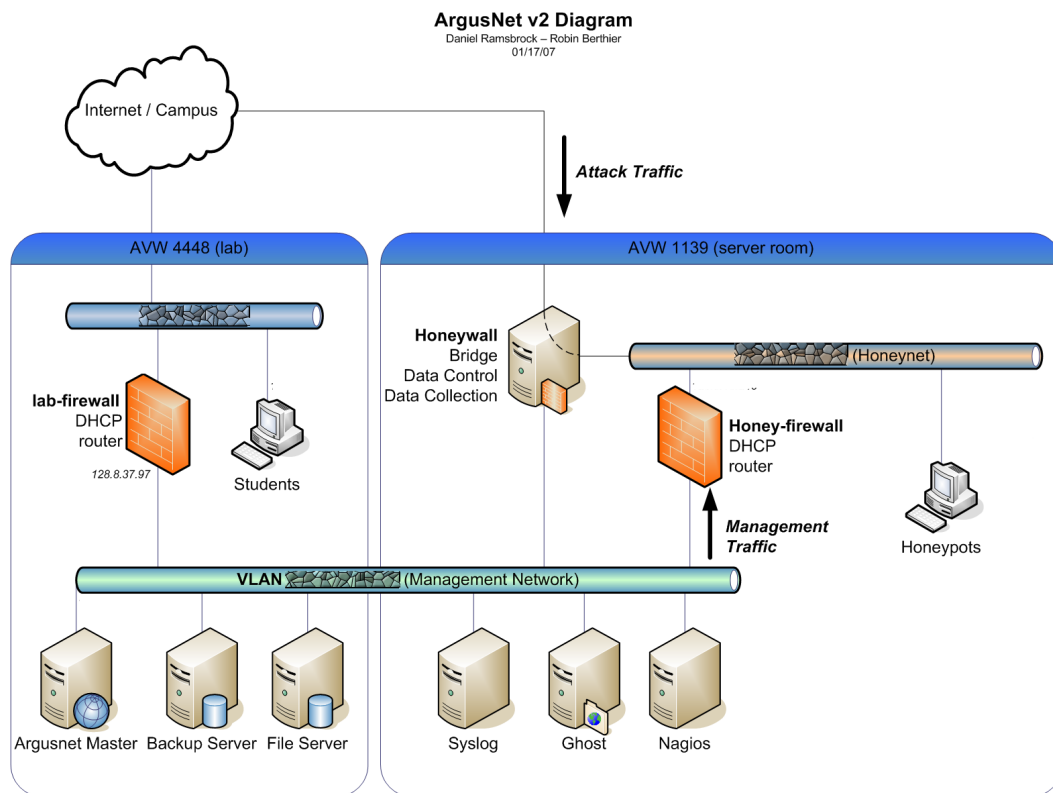
1. the **Honeywall:** from the Honeynet project, it is used for several purposes:
 - ◆ to collect network data (dump files and flows) and system data;
 - ◆ to limit outbound connections from honeypots;
 - ◆ to analyze and monitor the data collection through its protected web interface.

2. the **Honey-Firewall**: based on IPCop, it is used:

- ◆ to provide IP to honeypots with its DHCP server functionality;
- ◆ to hide and protect the management network from the honeypots;
- ◆ to route management traffic to administrate the honeypots;
- ◆ to forward specific traffic (syslog, ghost) from the honeypots to specific management machine;

3. the **Lab-Firewall**: also based on IPCop, it closes the management network perimeter and protects the management machines. It also provides IP addresses to the management network and VPN functionality for remote access.

There are other pieces like a Backup server (for architecture and experimental data), a Syslog server (collect system information), a server to re-image the honeypots, an Argusnet Master (host a Trac website for everything related to Argusnet) and a Nagios server (to monitor honeypots availability).



2.1.1 *Argusnet experiments*

The Argusnet network has been built to run experiments at the University of Maryland. During the last three years, numerous studies have been realized by Dr. Cukier and his team on this network. When I met Dr. Cukier (during the CRISIS '05 at the Bourges school of engineering), he was presenting the results of their works to determine if a correlation exist between port scans and attacks [7]. This study was one of the first done using Argusnet. Some others example of research projects can be read in [8] and [9]. The experiment presented in [9] is particularly interesting because it tend to analyze the human behavior of an attacker that gain an access to a Linux system.

“ To build a profile of attacker behavior, we looked for specific actions taken by the attacker and the order in which they occurred. These actions were: checking the configuration, changing the password, downloading a file, installing/running rogue code, and changing the system configuration.”

Argusnet is a very efficient architecture to study malicious behavior and improve many security techniques. Then, by helping to create a *Hybrid Honeypot* architecture with the ArgusProxy project, my goal was to improve Argusnet and open new research opportunities.

2.2 *ArgusProxy : software engineering*

To describe the specifications of the system, I used the Unified Modeling Language (UML) and created a set of diagrams that represents the different parts of the ArgusProxy software.

2.2.1 *Requirements*

Based on the information I gathered and presented in Chapter 1, I showed that the architecture needs to provide fast network response capabilities that can only be done in the deepest layers of an operating system. That's why I decided to work with the Linux kernel and, more specifically, with the Netfilter engine of the kernel.

Then, I designed a basic architecture from a very high point of view (Figure 8). In this architecture, ArgusProxy is split in two independent modules : the Redirector and the Decision Engine.

The goal of the Redirector is to handle connection, record them, send information to the decision engine

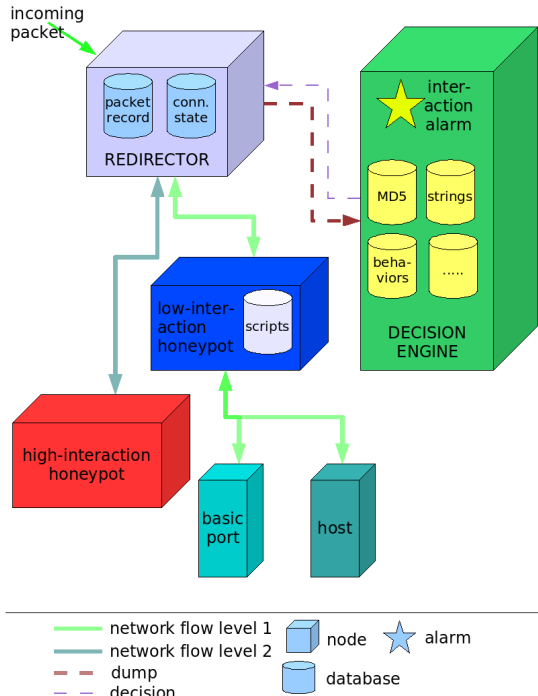


Figure 8: basic hybrid honeypot architecture

and replay a specific connection to a specific host if asked.

The goal of the Decision Engine is to run tests on incoming packets, keep a score for each incoming connections, send a message to the Redirector when the score of a connection exceeds a limit and control the interaction limit of a *low-interaction honeypot*.

Plugged together, those two engines should be able to manage a wide range of connections at the same time, providing a new kind of architecture that, as far as I know, has never been done before.

2.2.2 Activity Diagram

In UML, an activity diagram represents the business and operational step-by-step work flows of components in a system. An activity diagram shows the overall flow of control. This diagram is probably the most useful for a developer. The reading starts at the top left corner with an incoming packet. Going down the diagram, it's easy to follow the different operations processed on the packet (Figure 9).

The first bifurcation separate redirected packets from the rest of the packets. A packet which is part of a redirected connection will follow the steps in the green box. The other packets will be routed to the main part of the engine. In this part, they are first recorded, then forwarded to their destination (the yellow box) while a copy is sent to the redirection engine (the purple box).

When the decision to redirect a connection is taken, we reach the blue box at the bottom of the diagram. The connection is replayed to its destination and its state is recorded in the different tables.

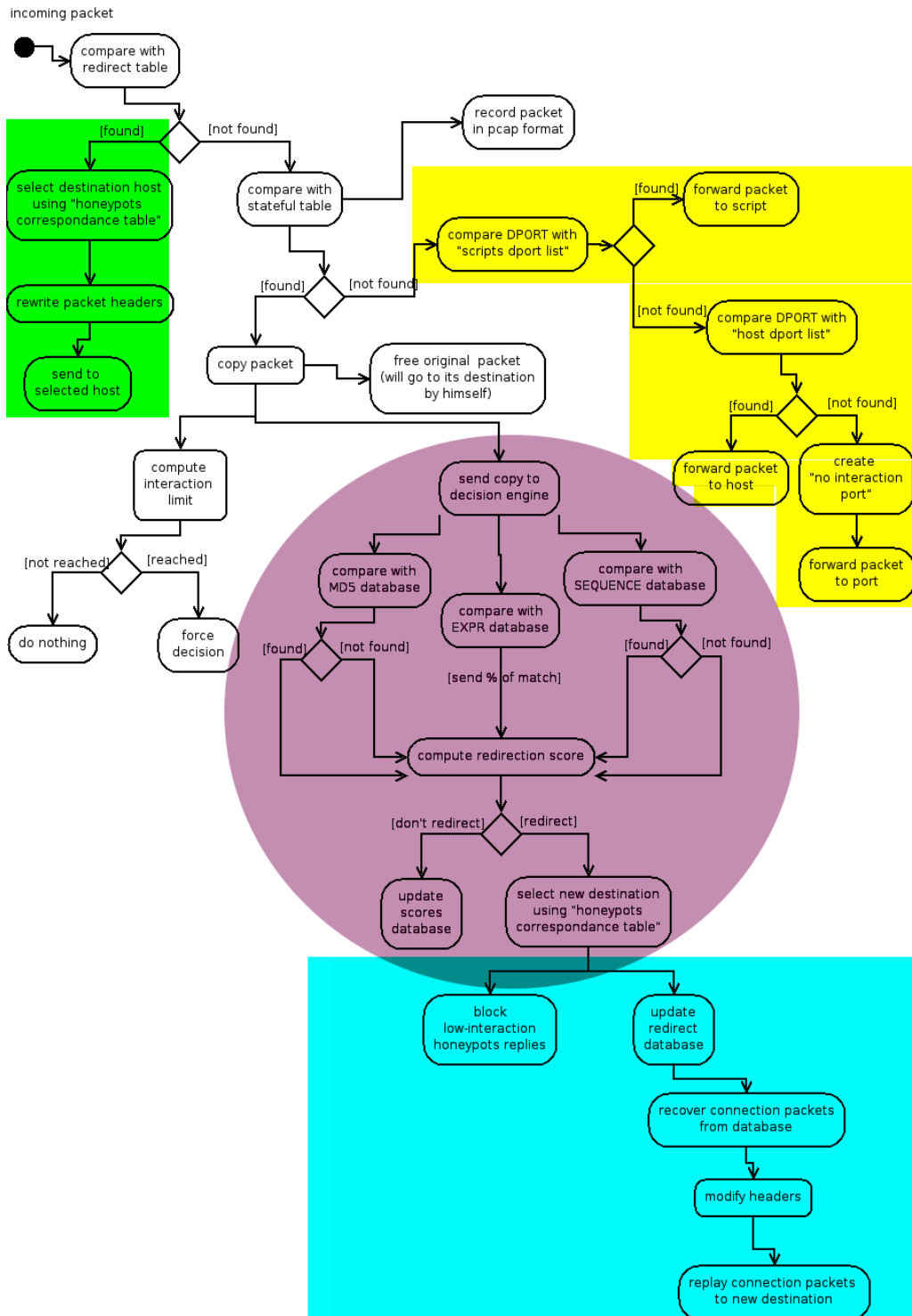


Figure 9: ArgusProxy activity diagram

2.2.3 Deployment Diagram

In the Unified Modeling Language, a deployment diagram serves to model the hardware used in system implementations, the components deployed on the hardware, and the associations between those components. The elements used in deployment diagrams are nodes (shown as a cube), components (shown as a rectangular box, with two rectangles on the left side) and associations.

The deployment diagram below is an improvement of Figure 8, it gives a more accurate presentation of the *hybrid honeypot* architecture.

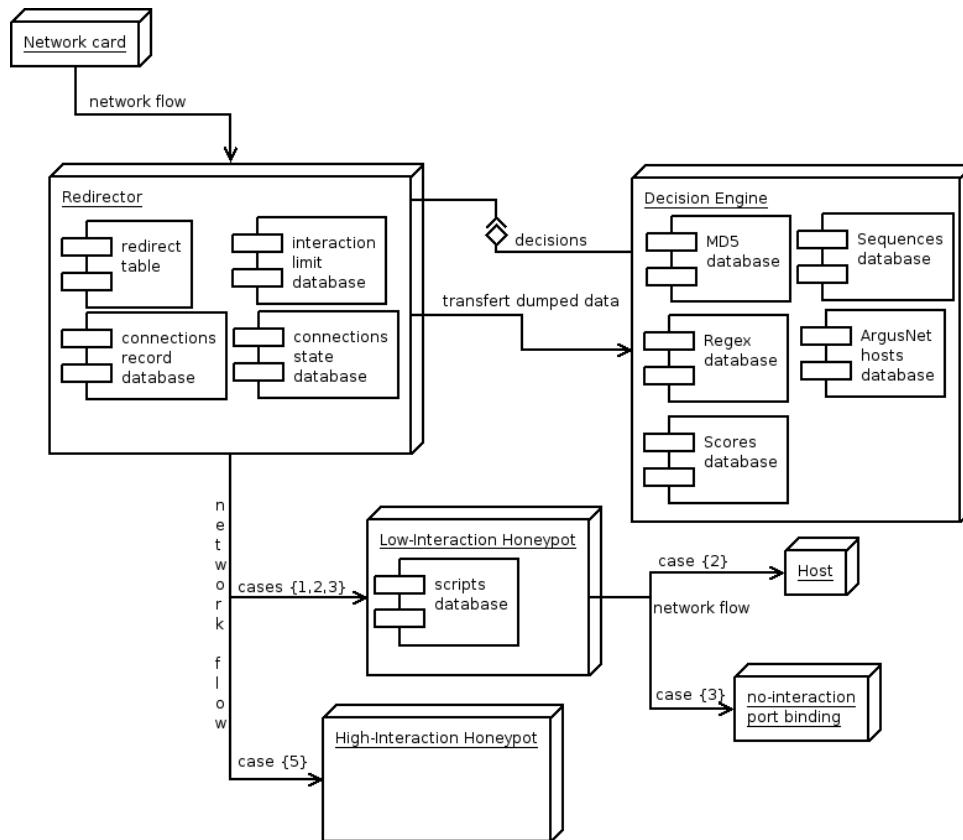


Figure 10: ArgusProxy Deployment diagram

I have also made some smaller diagrams to describe, from a closer point of view, the behavior of the software while processing incoming packets. I will present them in the next part.

2.2.4 Modules interactions

The Redirector and the Decision Engine need to share some information regarding to incoming connections. This implies the needs of a shared space in the system and a set of signals (for the Decision Engine and the Redirection to exchange notifications). The diagrams below present a vision of the architecture from a high level.

2.2.4.1 Redirector

The Redirector receives every incoming packet from the Netfilter Queue system via the library LibNetfilter_Queue. It records every packet both in a PCAP file and in a shared memory. It replays recorded communications when the Decision Engine sends a replay signal. The regular behavior for a non-redirection communication is to forward (or let go) the packet to its destination, which is a low-interaction honeypot, like HoneyD. HoneyD then decides where to redirect the packet (script, host or a simple port). A signal is sent to the Decision Engine to notify it that a new packet has been received and needs to be processed (Figure 11).

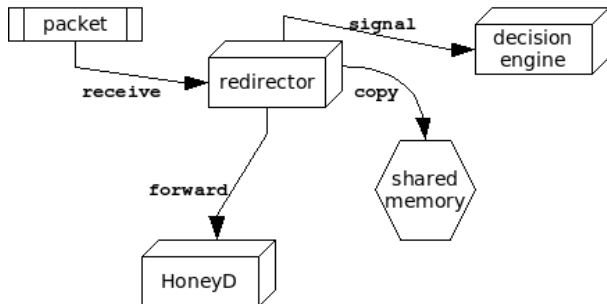


Figure 11: Redirector general behavior

When the Redirector receives a REDIRECT signal, it selects a new destination in the redirect table and replays the whole communication to the new destination in Argusnet. The communication could be recorded in a database or

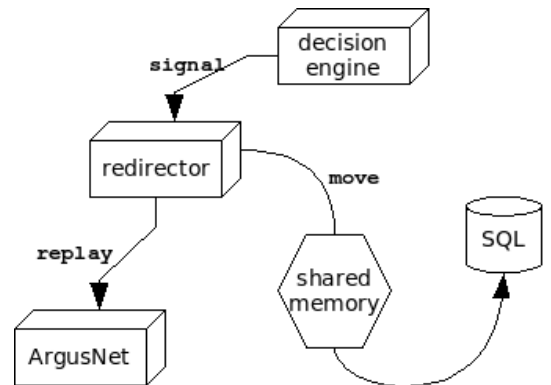


Figure 12: Connection replaying

just dropped from the Redirector memory (Figure 12).

Once the replay is done, the Redirector acts as a NAT (Network Address Translation) proxy in the communication. It handles the incoming packets from the attacker to send them to Argusnet and get answers from Argusnet to send

them to the attacker. This is complex because the attacker must not know that the communication has been redirected and that he's not speaking with the *low-interaction honeypot* anymore. When I did that,

I didn't realize that replaying and faking TCP connections will be, from far, the most difficult part of this project.

2.2.4.2 Decision Engine

The Decision Engine is started each time it receives a notification signal from the Redirector. It gets the new packet from the shared memory and starts the 3 sub-engines (MD5, EXPR, SEQUENCE) to process this packet. When the sub-engines send their status responses, the Decision Engine makes a decision based on the values received and the history of the connection and sends the decision to the Redirector (signals REDIRECT or CLOSE). Figure 13 shows the general architecture of the Decision Engine.

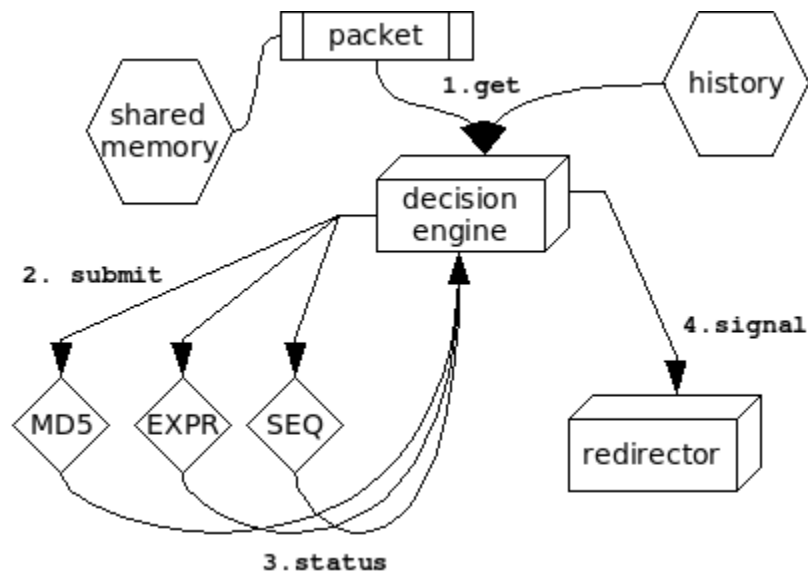


Figure 13: Decision engine general architecture

The modularity of this engine rely on the sub-engines that can be plugged to it. In Figure 13, the three engines are a checksum comparison engine that use the MD5 algorithm, an regular expression module that compare ASCII words with a database of known words and a connections sequences inspection module (like the one I presented in chapter 1.3.4). An incoming packet is first linked to its history and then processed by the chosen tests. The Decision Engine make its decision regarding the results of the tests and sends the corresponding signal to the Redirector.

2.2.4.3 Shared memory

To store information in memory, I chose to use a global Balanced Binary Tree to store connections and one Singly-Linked List (for each connection) to store the packets of the connection. Balanced Binary Trees are slow in writing mode because they need to be equilibrated at each writing but they give a very high speed access to information in reading mode. The time is an important problem because the attacker must not detect the redirection, and this last process imply the reading of data from the shared memory.

Thus, by using together the Balanced Binary Tree and Singly-Linked Lists, I built a reliable architecture to store both packets and information regarding a connection.

The information stored are :

1. Binary Tree Key : a tuple composed of the source IP address, the source port, the destination IP address and the destination port (ex : 82.54.165.9:5122:128.8.37.121:25);
2. Binary Tree meta-information : a boolean value set when a connection is redirected, a boolean value set when a connection has been replayed to Argusnet, a socket to access a connection between ArgusProxy and an Argusnet host, a raw socket used to reply packets to the attacker, the Singly-Linked List to record connection packets, a thread to listen for replied packets from Argusnet and a time value to delete out of date entries.

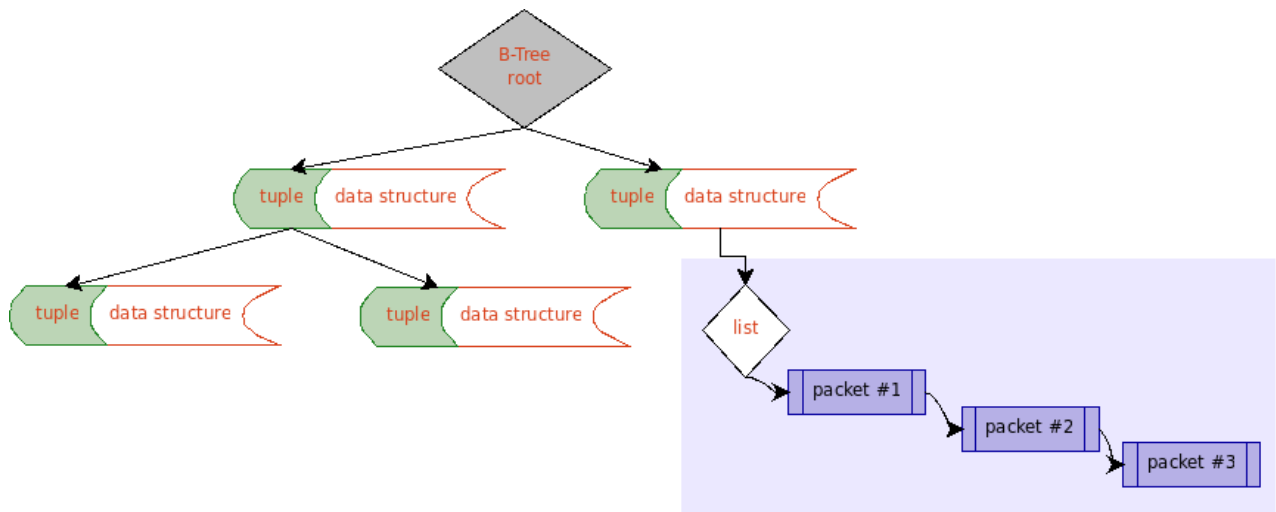


Figure 14: ArgusProxy memory storage using B-Tree and Singly-Linked Lists

2.2.5 Documentation

To ensure that my work would be resume and continued after I left the University, the writing of an exhaustive documentation was part of the requirements.



I have used the doxygen syntax all along the project to build the documentation. Doxygen is probably the only serious documentation project available today. It provides a complete set of tools to transform code comments into a readable documentation.

The figure below represent the “process_pkt” function and sub-functions of the software (Figure 15). This figure has been generated using Doxygen and the comments I have put in the source code.

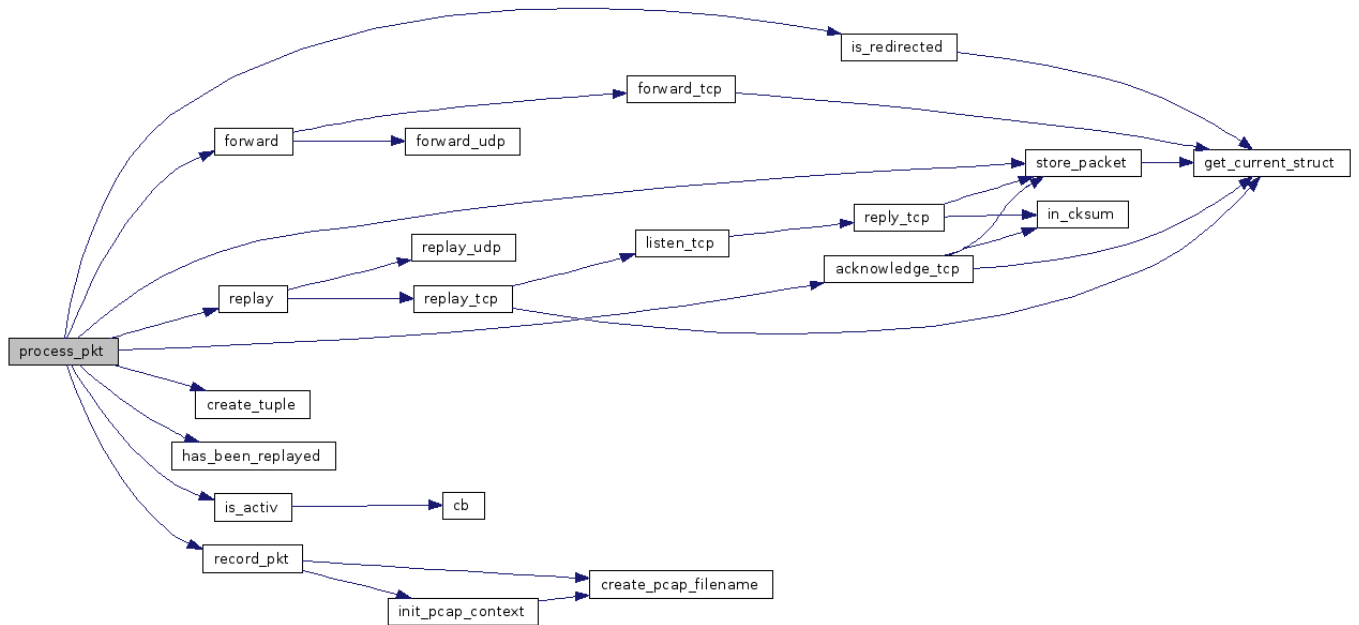


Figure 15: Process_pkt function and sub-functions called for each packet

All the work I did around ArgusProxy is also stored in the Trac website of the laboratory. This includes the subversion repository (that also contains the code I wrote to study the sequence inspection), the wiki pages and the project management information that I will present on the next section of this chapter.

2.3 Timetable

The original version of the development phase has been planned to be done within 45 days. I have first planned to deploy a beta version of the engine at the beginning of August. During the months of June and July, I kept the progression quite according the original timetable. But when I reached the replay engine, I realized I wouldn't do it on time.

I have made a mistake by thinking that I could code a TCP engine in 9 days. But quickly, I realized that coding a TCP engine will not be as easy as what I had thought. So, I had to come back to a new modeling phase and prepare more accurately this part. Trying to code it without a good preparation was definitely too hard. Instead of spending 9 days, I spent almost 30 days on it (see Appendix C).

2.3.1 Gantt diagram

The Gantt diagram below detail the development of the Redirector.

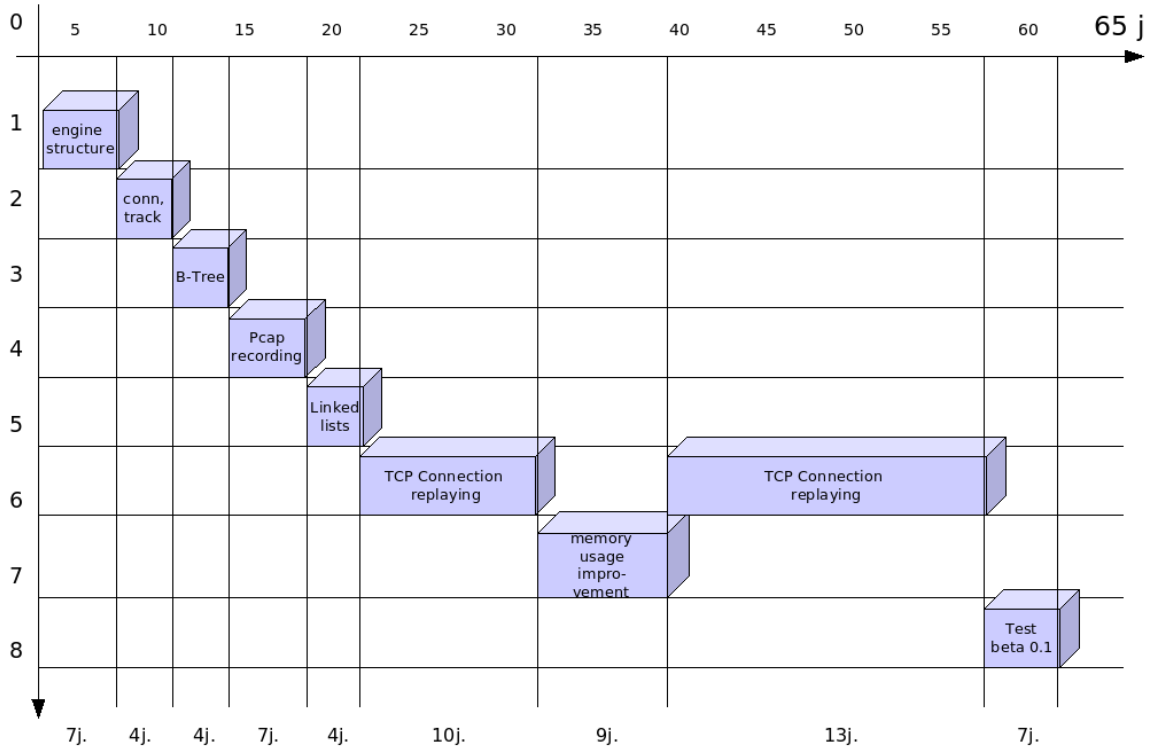


Figure 16: Gantt diagram for the ArgusProxy Redirector

What is not shown on this diagram is the bug correction phase that will stand during the month of september. During this months, I will improve the architecture and implement the last features. So, the Redirection engine of ArgusProxy will finally be a 95 days project.

2.4 *Technical issues*

The goal of this report is not to detail deeply the way I coded ArgusProxy (the most interesting functions are available in Appendix D) but, when coding in C, there's some interesting issues to discuss.

Despite its popularity, the C language has been widely criticized. Such criticisms fall into two broad classes: operations that are too hard to achieve using basic C, and undesirable operations that are too easy to accidentally invoke while using C. Putting this another way, the safe, effective use of C requires more programmer skill, experience, effort, and attention to detail than is required for some other programming languages.

Because ArgusProxy was my first software engineering project of this size, I chose to rely on higher level libraries for the critical sections of code.

For the Connection State checking and the hook of incoming connections, I used the Netfilter library. Despite the fact that Netfilter developers are amazing coders but not very efficient documentation writers, I won a precious time by re-using Netfilter's capabilities.

For example, the code below requests the Netfilter system to check the state of a connection (both TCP and UDP works).

```
/*! nfct_query - send a query to ctnetlink
\param[in] h: library handler
\param[in] NFCT_Q_GET: query type
\param[in] ct: data required to send the query
*
* For query types:
*   NFCT_Q_CREATE: add a new conntrack, if it exists, fail
*   NFCT_Q_CREATE_UPDATE: add a new conntrack, if it exists, update it
*   NFCT_Q_UPDATE: update a conntrack
*   NFCT_Q_DESTROY: destroy a conntrack
*   NFCT_Q_GET: get a conntrack
*
* On error, -1 is returned and errno is explicetely set. On success, 0
* is returned.
*/
errno = 0;
conn_state = nfct_query(cth, NFCT_Q_GET, ct);
```

Regarding the Balanced Binary Tree and the Singly-Linked Lists, I have tried first to code them by



myself but, quickly, I have found that the Gnome Library [4] provides a far more efficient and secure implementation of both of them.

More than that, this library re-implement a large number of widely used functions in a more secure way. I have tried, when it was possible, to rely on them when I have seen they were not slowing down my code.

As another example, the function below is used to store a packet in the Singly-linked list (in bold green) of a specific connection in the Binary Tree (in bold blue).

```

    /*!
    * get the b-tree key
    */
    if (
        TRUE != g_tree_lookup_extended(conn_tree, key_one->str, NULL, NULL)
        &&
        TRUE != g_tree_lookup_extended(conn_tree, key_two->str, NULL, NULL)
    )
    {
        /*! if doesn't exist, create it and init the whole structure
        * as a value for this entry
        */
        struct conn_struct *add_new_data;
        add_new_data = malloc( sizeof(*add_new_data) );

        /*! get current time
        */
        gint *curtime;
        struct tms current;
        curtime = g_strdup_printf("%d", times(&current));

        /*! fill the structure
        */
        add_new_data->access_time = curtime;
        add_new_data->redirected = 0;
        add_new_data->hasbeenreplayed = 0;
        add_new_data->socket = 0;
        add_new_data->rawsocket = 0;
        add_new_data->recordlist = NULL;

        /*! store the address of the payload as a new entry of the list
        */
        add_new_data->recordlist = g_slist_append(add_new_data->recordlist, store);

        /*! add the list to the tree, value contain the address of the first entry of the list
        */
        g_tree_insert(conn_tree, key_one->str, add_new_data);

        g_print("STORING FUNCTION : entry created for %s\n",key_one->str);
    }

```

The Gnome Library really provides a simplified access to high performance functions. It doesn't change the algorithmic nor simplify the specification phase but provides a strong implementation of critical features that usually require high skills to implement them. I can evaluate that the use of this library

has divided by two the amount of time needed to code ArgusProxy.

Regarding the network functions of ArgusProxy, I chose to use the basic network capabilities provided by the Linux operating system. If, for regular TCP communications, this architecture gives a very simple manner to send and receive packets, this is a bit more difficult to do in the case of raw packets used to send packets outside of regular communications.

There's almost no documentation on the Internet about raw sockets, most people are just using them to create SYN flooders. So, I had to read the RFC 793 again and again to recreate a basic TCP stack.

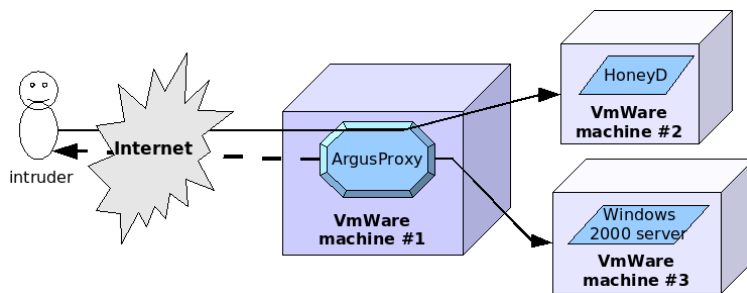
One of the biggest problems was the TCP checksum computation, for two reasons :

- ◆ First, the TCP checksum is computed using a pseudo header that breaks the principle “TCP is not aware of what is done under its layer”. In fact, the TCP pseudo header contains the IP addresses of the communication, and this is a bit tricky to set up.
- ◆ Second, the checksum is a 16 bits word computed on the pseudo header, the TCP header and the payload. The trick is that the main algorithm has been created several years ago and is not very portable. So I spent several days to understand the steps to compute this checksum and, finally, recode it before I can understand my mistakes.

I could also add the memory issues I had to deal with, especially regarding some data communications between the HEAP and the STACK, many allocation problems and so on... I learned from a C language professor here that this was the kind of problems every developers have to deal with one day.

Tests & future work

At the time I'm writing these pages, ArgusProxy has just reached the first workable beta version. The support of UDP protocol is not implemented yet and only one Argusnet destination can be use (I should finish that during the next weeks).



Network 2: ArgusProxy VmWare network

I'm actually working with Robin Berthier to set up a test bed (see Network 2) using virtual machines. Before the end of the month of August, we should have a workable network connected to the Internet.

Once the integration qualification done, I will work on operational qualification by studying the behavior of ArgusProxy in a real environment. According to the V cycle development process (Figure 17), this will certainly imply to re-code some parts of the software and correct some others.

Finally, the first final version of ArgusProxy should be released at the end of September if the software successfully pass the performance qualification. The decision engine used will still be minimal but this is a part of the project we decided to ignore from the beginning.

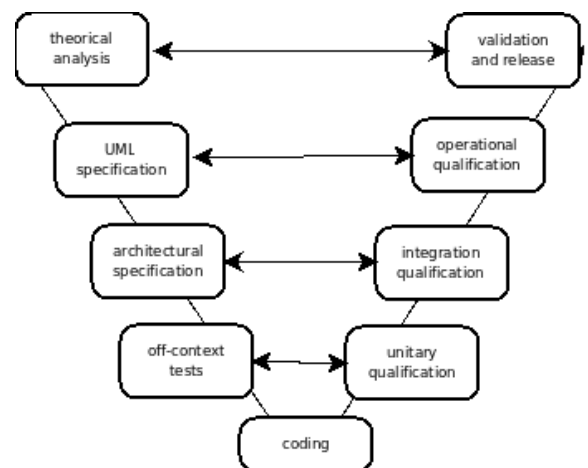


Figure 17: V cycle development diagram

Conclusion



This internship has been an extremely interesting experience to me. During the last five months, I had the opportunities to do things I wanted to do for years.

First, I have worked on a research project, which was new to me because of my major and previous experiences. I am glad to have had the opportunity to discover “the other side” of computer security. It was a bit difficult to learn how to think as a researcher. I understood, after a few weeks, that the methodology to solve a problem is different from what I learned during my Master's degree. In a company, the right way to go from A to B is to use the most efficient paths spending as little money as possible. In a research laboratory, the same problem is solved when all the possible paths have been studied and when it is proven that one of them is good.

Secondly, I have directed my first software engineering project from the very first steps of the study to the release of the final version. This was an interesting experience and it gave me the occasion to reuse what I learned in software engineering, programming, networks and systems classes during the last two years. Creating a software from scratch has been challenging but I acquired a better understanding of the need of precise specification to create secure softwares. This appear more important to me because I experienced by myself the many problems that occurred during the development phase.

Thirdly, this internship has been an amazing social experience. I have met so many people from so many countries, it was (and it still is) really enjoying to share our points of views, experiences, and cultures. Beyond the American way of life, I learned a lot from a social point of view.

The overall outcome of this internship is very positive, and I would like to thank one more time Dr. Michel Cukier and Robin Berthier for having offered me the opportunity to work with them at the University of Maryland.

References




- [1] Michael Bailey, Evan Cooke, David Watson, Farnam Jahanian, Niels Provos. *A Hybrid Honeypot Architecture for Scalable Network Monitoring*. University of Michigan, Google Inc. October 27, 2004
- [2] Elie Bursztein. *TCP Timestamp To count Hosts behind NAT*. phrack #63. http://www.phrack.org/archives/63/p63-0x03_Linenoise.txt
- [3] David Duncombe, George Mohay, Andrew Clark. *Synapse: Auto-correlation and Dynamic Attack Redirection in an Immunologically-inspired IDS*. Queensland University of Technology
- [4] *Glib Reference Manual*, <http://developer.gnome.org/doc/API/2.2/glib/index.html>
- [5] *The Honeynet Project*, <http://www.honeynet.org>
- [6] *Honeypots*, by Lance Spitzner, <http://www.spitzner.net/honeypots.html>
- [7] Susmit Panjwani, Stephanie Tan, Keith M. Jarrin, Michel Cukier: *An Experimental Evaluation to Determine if Port Scans are Precursors to an Attack*. DSN 2005: 602-611
- [8] Michel Cukier, Robin Berthier, Susmit Panjwani, Stephanie Tan: *A Statistical Analysis of Attack Data to Separate Attacks*. DSN 2006: 383-392
- [9] Daniel Ramsbrock, Robin Berthier, Michel Cukier: *Profiling Attacker Behavior Following SSH Compromises*. DSN 2007: 119-124
- [10] Niels Provos, Thorsten Holz, *Virtual Honeypots - From Botnet Tracking to Intrusion Detection*, Addison Wesley Editions
- [11] *The Deception Toolkit*, by Fred Cohen, <http://all.net/dtk/dtk.html>

Figures



Figure 1: honeypots projects since 1997.....	10
Figure 2: TCP redirection in hybrid honeypot.....	12
Figure 3: basic hybrid honeypot architecture.....	13
Figure 4: sequence string inspection results for TCP connections on port 445.....	16
Figure 5: connection sequence matching using a variance in packet size.....	17
Figure 6: cumulated sequences inspection results.....	17
Figure 7: interaction limit with Honeyd faking a Windows 2000 SMTP daemon	18
Figure 8: basic hybrid honeypot architecture.....	22
Figure 9: ArgusProxy activity diagram.....	23
Figure 10: ArgusProxy Deployment diagram.....	24
Figure 11: Redirector general behavior.....	25
Figure 12: Connection replaying.....	25
Figure 13: Decision engine general architecture.....	26
Figure 14: ArgusProxy memory storage using B-Tree and Singly-Linked Lists.....	27
Figure 15: Process_pkt function and sub-functions called for each packet.....	28
Figure 16: Gantt diagram for the ArgusProxy Redirector.....	29
Figure 17: V cycle development diagram.....	33

Glossary

- 
- **Balanced Binary Tree** : A binary search tree that attempts to keep its height, or the number of levels of nodes beneath the root, as small as possible at all times.
 - **Checksum** : A checksum is a form of redundancy check, a simple way to protect the integrity of data by detecting errors in data that are sent through a network.
 - **HoneyD** : HoneyD is an open source computer program that allows a user to set up and run multiple virtual hosts on a computer network. These virtual hosts can be configured to mimic several different types of servers, allowing the user to simulate an infinite number of computer network configurations.
 - **IDS** : An intrusion detection system is used to detect many types of malicious network traffic and computer usage that can't be detected by a conventional firewall. This includes network attacks against vulnerable services, data driven attacks on applications, host based attacks such as privilege escalation, unauthorized logins and access to sensitive files, and malware (viruses, trojan horses, and worms).
 - **IP** : Internet Protocol, the computer networking protocol used on the Internet
 - **Payload** : In communication, telecommunications and information science, the payload is the data, such as a data field, block, or stream, being processed or transported by a protocol (like TCP or UDP).
 - **Singly-Linked Lists** : A singly-linked list is a sequence of nodes, each containing arbitrary data fields and a reference ("link") pointing to the next node.
 - **Subversion** : Subversion (SVN) is a version control system (VCS) initiated in 2000 by CollabNet Inc. It allows users to keep track of changes made over time to any type of electronic data. Typical uses are versioning source code, web pages or design documents.
 - **TCP** : Transmission Control Protocol, a transportation protocol that is one of the core protocols of the Internet protocol suite. This protocol guarantees reliable and in-order delivery of data from sender to receiver.
 - **Trac** : Trac is an open source, minimalist, web-based project management and bug-tracking tool, inspired by CVSTrac. It is developed and maintained by Edgewall Software.
 - **UDP** : User Datagram Protocol (UDP) is one of the core protocols of the Internet protocol suite. UDP does not guarantee reliability or ordering in the way that TCP does. Datagrams may arrive out of order, appear duplicated, or go missing without notice.
 - **Nessus** : Nessus is a comprehensive vulnerability scanning program. Its goal is to detect potential or confirmed weaknesses on the tested machines.
 - **Metasploit** : The Metasploit Project is an open source computer security project which provides information about security vulnerabilities and aids in penetration testing and IDS signature development.

Appendix

A ⇒ Gantt

Company: University of Maryland, Department of Mechanical Engineering, Center for risks and reliability
 Manager: Julien VEHENT
 Start: April 3, 2007
 Finish: September 28, 2007

WBS	Nom	Démarrer	Terminer	Travail	Durée	Mou	Coût	Assigné à
1	State of the art	avr 3	avr 6	4j	4j		0	jv
2	doc1 : attack redirection, state of the art and requirements	avr 9	avr 9	4h	4h	1j 4h	0	jv
3	Weekly meeting	avr 9	avr 9	N/A	N/A	124j	0	jv, mc, rb
4	Docs & Tests : tcp redirection	avr 9	avr 16	5j	5j	1j 4h	0	jv
5	Weekly meeting	avr 17	avr 17	N/A	N/A	118j	0	jv, mc, rb
6	Docs & Tests : honeypd and snort	avr 16	avr 24	6j	6j	1j 4h	0	jv
7	Weekly meeting	avr 23	avr 23	N/A	N/A	114j	0	jv, mc, rb
8	attack selection in honeypots	avr 24	avr 30	4j	4j	1j 4h	0	jv
9	Weekly Meeting	avr 30	avr 30	N/A	N/A	109j	0	jv, mc, rb
10	Comparative analysis of Checks and Exploits	avr 30	mai 7	5j	5j	1j 4h	0	jv
11	Weekly Meeting	mai 7	mai 7	N/A	N/A	104j	0	jv, mc, rb
12	ArgusProxy: Algorithmic and UML specification	mai 7	mai 21	10j	10j	1j 4h	0	jv
13	Weekly Meeting	avr 3	avr 3	N/A	N/A	128j	0	jv, mc, rb
14	Connections Sequence: Analysis and Correlations	mai 21	jun 6	12j	12j	1j 4h	0	jv
15	▼ Argusproxy	jun 6	sep 26	95j	80j	1j 4h	0	jv
15.1	Structure of the engine	jun 6	jun 15	7j	7j	4h	0	jv
15.2	Check connection state	jun 15	jun 21	4j	4j	1j 4h	0	jv
15.3	Storage techniques for dynamic informations	jun 21	jun 27	4j	4j	1j 4h	0	jv
15.4	Pcap recording	jun 27	jui 6	7j	7j	4h	0	jv
15.5	Replay/Rewrite connections	jui 6	aoû 17	30j	30j	4h	0	jv
15.6	Basic Decision Engine	aoû 17	aoû 23	4j	4j	1j 4h	0	jv
15.7	Network engine tests & validation	aoû 23	aoû 30	10j	5j	1j 4h	0	jv, rb
15.8	UDP support	aoû 30	sep 5	4j	4j	1j 4h	0	jv
15.9	Multi-Argusnet destination	sep 5	sep 12	5j	5j	1j 4h	0	jv
15.10	Operationnal qualification	sep 12	sep 26	20j	10j	1j 4h	0	jv, rb
16	internship report	aoû 31	aoû 31	N/A	N/A	20j	0	jv, mc, rb

I used the Open Source software "Planner" to manage this project. The diagram generated by planner is too big to be copied here.

B ⇒ Connections sequences analysis

Introduction

In this document, I will show that the use of connections sequences could be useful to regroup connections and reduce eliminate non-interesting ones.

First : What is a connection sequence ?

A connection sequence is the simplified representation of the packets exchanged between two hosts during a communications. A **sequence string** is used to represent the connection sequences. In a classical TCP communication, a sequence string is composed of :

foreach packets of the connection :

- packet flags
- urg
- ack
- psh
- rst
- syn
- fin
- full packet size in bytes
- direction (O => coming from outside; I => coming from inside)

A typical connection string on port 445 look like this :

```
00001062001001062I0100006000110001910011000143I0110002300011000455I011000360001100093I01100097001100093I01000160001000160I000100600000100600
```

Thus, by representing every connections in a dump file in this format, we can easily regroup identicals sequences.

The ACK packets problem

In most of TCP communications, packets are usually following a specific order. But ACK packets are different. The Acknowledgment rythms depend on the network stack, the size of the window (which depend of the operating system), and so on... thus, ACK packets are rarely received in the same order and that's a problem for this experiment because ACK packets change the sequence string. Handshake and connection closing are also useless because always identical.

That's why the connections strings used in the experiments below are limited to PUSH packets. We don't care about the handshakes, acknowledgements and closings.

Source datas

The source datas used are [ArgusNet](#) dump from the IIS server. The period begin in 02192007 until 03272007. {see the [repository](#)}

Process pcap files

I use a little C program to process pcap files. The code is very simple. It produce a file where each line is a packet summary, like that :

```
201.11.66.2:4704:192.168.2.45:139:0:1:0:0:0:0:60:I
201.11.66.2:4704:192.168.2.45:139:0:1:0:1:0:0:60:I
213.22.112.42:2465:192.168.2.45:139:0:0:0:0:0:1:0:78:O
```

The external IP address is always the first argument of the line, followed by the external port, the local IP address, the local port, the TCP flags, the packet size and the direction (coming from inside or outside). The piece of code which do this is :

```
#!/ SEARCH PATTERN ON SRC ADDR TO FIND THE DIRECTION */
char *r = strstr(inet_ntoa(ip_h->ip_src), "192.168");

/*! if pattern found, packet is coming from the inside */
if (r != NULL )
{
    fprintf(out, "%s:%u:", inet_ntoa(ip_h->ip_dst), ntohs(tcp_h->dest));
    fprintf(out, "%s:%u:%u:%u:%u:%u:%u:%u:%d:I\n", inet_ntoa(ip_h->ip_src),
ntohs(tcp_h->source), tcp_h->urg, tcp_h->ack, tcp_h->psh, tcp_h->rst, tcp_h->syn,
tcp_h->fin, header->len);
}
else /*! pattern not found, packet is coming from the outside */
{
    fprintf(out, "%s:%u:", inet_ntoa(ip_h->ip_src), ntohs(tcp_h->source));
    fprintf(out, "%s:%u:%u:%u:%u:%u:%u:%u:%d:O\n", inet_ntoa(ip_h->ip_dst),
ntohs(tcp_h->dest), tcp_h->urg, tcp_h->ack, tcp_h->psh, tcp_h->rst, tcp_h->syn,
tcp_h->fin, header->len);
}
```

Then, I use a Perl program to process this file and regroup packets in connections.

```
# Regroup connections sequences in %global hash table
open FILE, "parsed.txt" or die print "unable to open file parsed.txt\n$!\n";
while (<FILE>){
    # process lines using ":" separator
    # example line : 87.91.58.222:49585:192.168.2.45:139:0:0:0:0:1:0:66:O
    chomp($_);
    my @line = split(/:/, $_);

    # create ID (IP:PORT:IP:PORT) and Sequence string
    my $id = $line[0].":".$line[1].":".$line[2].":".$line[3];
    my $string =
$line[4].$line[5].$line[6].$line[7].$line[8].$line[9].$line[10].$line[11];

    # add $string at the end of the string sequence
    $global{$id} .= $string;
}
}
```

The Perl program is build to do far more than that, it will call the pcap parser itself and process the data to produce a result file. Launch it without arguments to print the help information.

Count identical sequences

The goal of this experiment is to count identical sequences (limited to PUSH packets) that are directed to local port 445, which's Microsoft CIFS default port.

The tables below shows the results of this experiment.

packets	15722
connections	1850
avg pkt/con	8.5

nb of identical sequences	of scheme	%
257	0110001910011000143I0110002300011000455I011000360001100093I01100097001100093I	13.89%
224	0110001910011000143I0110002300011000455I011000368001100093I01100097001100093I	12.11%
219	0110001910011000143I0110002300011000455I011000384001100093I01100097001100093I	11.84%
135	0110001910011000143I0110002300011000455I011000376001100093I01100097001100093I	7.30%
113	0110001910011000143I0110002900011000455I011000424001100093I	6.11%
108	0110001910011000143I0110002300011000455I011000396001100093I01100097001100093I	5.84%
102	0110001910011000143I0110002900011000455I011000428001100093I	5.51%
66	0110001910011000143I0110002220011000455I011000276001100093I	3.57%
61	0110001910011000143I0110002300011000455I011000400001100093I01100097001100093I	3.30%
60	0110001910011000143I0110002300011000455I011000412001100093I01100097001100093I	3.24%
56	0110001910011000143I0110002900011000455I011000440001100093I	3.03%
53	0110001910011000143I0110002300011000455I011000366001100093I01100097001100093I	2.86%
39	0110001910011000143I0110002900011000455I011000444001100093I	2.11%
30	0110001910011000143I0110002600011000455I011000406001100093I	1.62%
30	0110001910011000143I0110002300011000455I011000382001100093I01100097001100093I	1.62%
29	0110001910011000143I0110002900011000455I011000434001100093I	1.57%
27	0110001910011000143I0110002300011000455I011000380001100093I01100097001100093I	1.46%
24	0110001910011000143I0110002900011000455I011000450001100093I	1.30%
20	0110001910011000143I	1.08%
15	0110002030011000155I0110002340011000467I0110002880011000105I	0.81%
14 to 2	-----	3.89%

nb of unique sequences (seen just once)	110	5.95%
--	-----	-------

Those results are interesting because they shows that 94.05% of the connections are seen more than 1 time. So this means that we could probably reduce by 20 times the number of connections handled by [ArgusNet](#) hosts. But this experiment doesn't care about False Positive, so this could be a limitation.

Relevancy of variation in packets size

As we can see in the previous table, many sequences are similars and differs just by a few bytes in one or two packets. Thus, I have made another experiment to evaluate the relevancy of a variation in packets sizes. I have took the first sequence string of the previous result :

0110001910011000143I0110002300011000455I011000360001100093I01100097001100093I

and I have redone the same experiment but with an increasing variation from 1% to 100% in packets sizes.

This is the pseudo code :

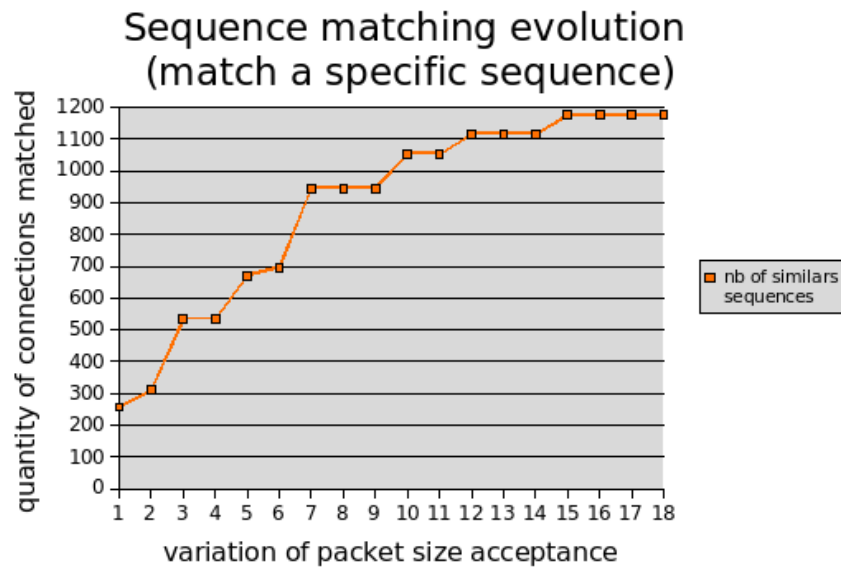
```

For each sequence string {
    if nb of packets and direction is equal to reference
    then {
        for each packet size in sequence string {
            # ref packet size is taken from the reference sequence string
            if packet size is upper to (ref packet size + variation) or lower to
(ref packet size - variation)
            then { mark sequence string as NOT EQUAL }
        }
        if sequence string is not NOT EQUAL { mark sequence as EQUAL }
    }
}

```

Then, by doing this for each variation value between 1 and 18, we have the following results :

variance (%)	nb of similars sequences
1	257
2	310
3	534
4	534
5	670
6	697
7	946
8	946
9	946
10	1054
11	1054
12	1116
13	1116
14	1116
15	1176
16	1176
17	1176
18	1176
...



Since 18 to 100, the number of similar sequences matched is always 1176. The connection matching is almost 5 times better with a variation of 15% of packets sizes...This could be an interesting technique to increase the performances of a decision engine based on sequences strings inspection, but again this could produce a lot of False Positive....

Connections sequences validity period

I'm no longer working with the dump_IIS.pcap file but with a set a 9 pcap files (one for each month) covering all [ArgusNet](#) communication since September 2006 to the May, 9th 2006. In this last experiment, I have took 4 sequences strings from 3 differents months.

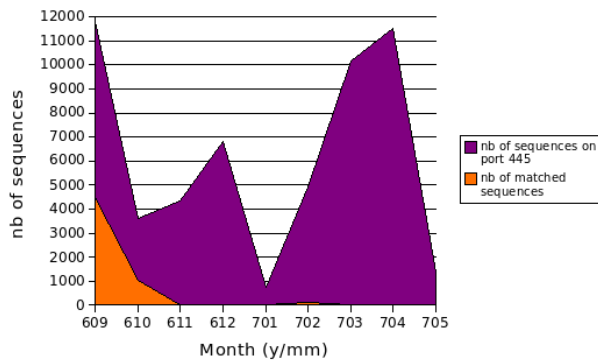
- **September 2006 :**
0110000;011000I;0110000;011000I;0110000;011000I;0110000;011000I;:191;143;230;335;392;93;97;93;
- **December 2006 :** 0110000;011000I;:191;93;
- **April 2007 :**
0110000;011000I;0110000;011000I;0110000;011000I;0110000;011000I;:191;143;230;455;360;93;97;93;
- **April 2007 :**
0110000;011000I;0110000;011000I;0110000;011000I;0110000;011000I;:191;143;230;335;392;93;97;93;

And I have tried to find those sequences strings in differents dump files. Again, sequences are limited to PUSH packet and again I'm only dealing with destination port 445. The analyser also use a variation value of 20%, which is, based on the previous result, big enough to include the most variation we can.

September 2006

dump	nb of matched sequences	nb of sequences on port 445
609	4479	7329
610	1019	2601
611	37	4307
612	0	6794
701	0	744
702	129	4771
703	0	10142
704	0	11528
705	0	1398

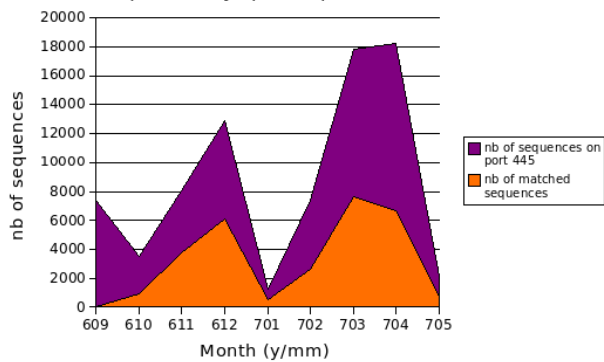
Matched sequences by specific pattern and variance of 20%



December 2006 : 0110000;011000I;:191;93;

dump	nb of matched sequences	nb of sequences on port 445
609	56	7329
610	922	2601
611	3732	4307
612	6104	6794
701	518	744
702	2639	4771
703	7662	10142
704	6698	11528
705	759	1398

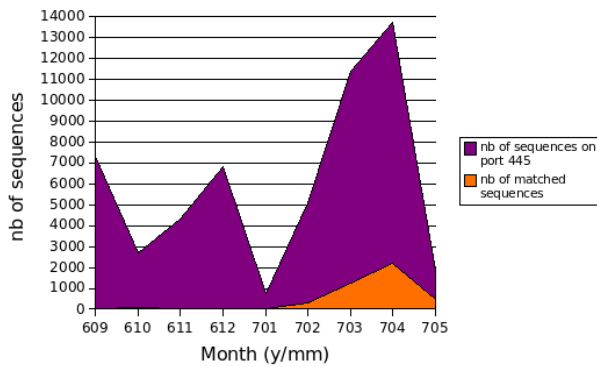
Matched sequences by specific pattern and variance of 20%



April 2007

dump	nb of matched sequences	nb of sequences on port 445
609	0	7329
610	88	2601
611	0	4307
612	0	6794
701	0	744
702	299	4771
703	1242	10142
704	2194	11528
705	444	1398

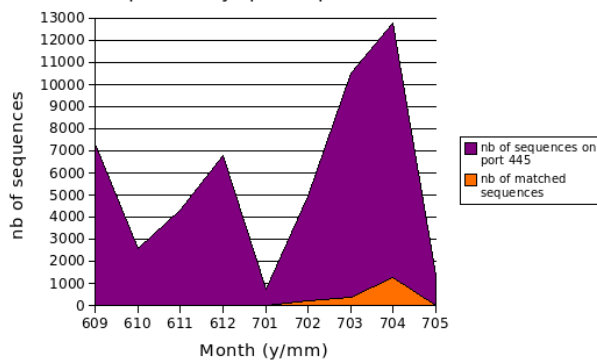
Matched sequences by specific pattern and variance of 20%



April 2007

dump	nb of matched sequences	nb of sequences on port 445
609	0	7329
610	2	2601
611	2	4307
612	0	6794
701	0	744
702	213	4771
703	400	10142
704	1284	11528
705	5	1398

Matched sequences by specific pattern and variance of 20%



Except for the sequences string of december, which is very short, all the sequences strings have a limited life time. This means that a sequences strings recorded in september is useless after october, and we can then limit the storage time to approximatively a months and a half.

C ⇒ ArgusProxy Network Engine

Hybrid honeypot requirements

The need of forwarding connections between honeypots implies to build a non-conventional network engine. The needs are :

- ◆ Low latency replay time;
- ◆ Transparency to the intruder;
- ◆ UDP and TCP complacency (but I will not take interest on UDP in this document);

So, I tried for a while to think about it as a NAT problem. Considering that ArgusProxy will act as a NAT router in a Linux environment, it could have been interesting to re-use some of the Netfilter features to simplify the architecture. But our architecture implies to update the redirection rules in the middle of the communication, and Netfilter doesn't commit the NAT rules changes before the next communication (at least, for TCP), so it doesn't works...

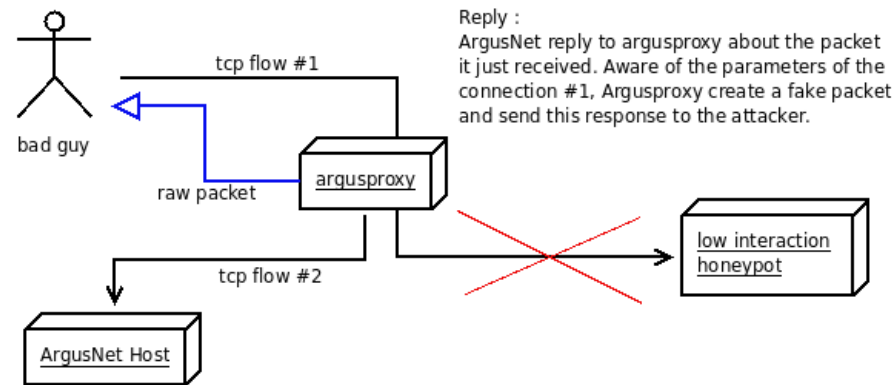
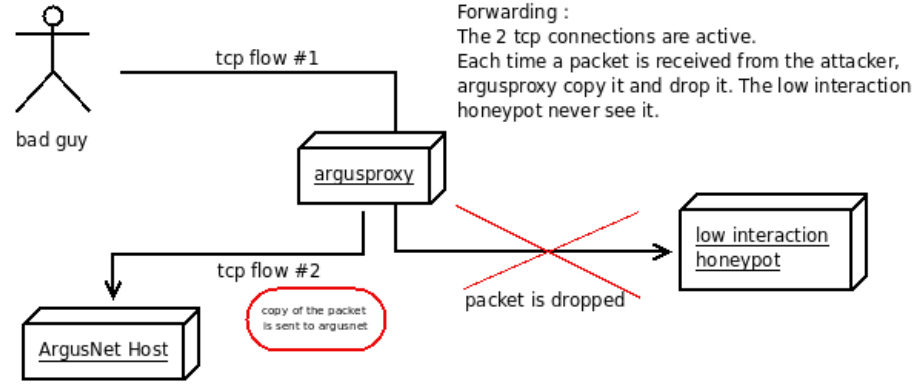
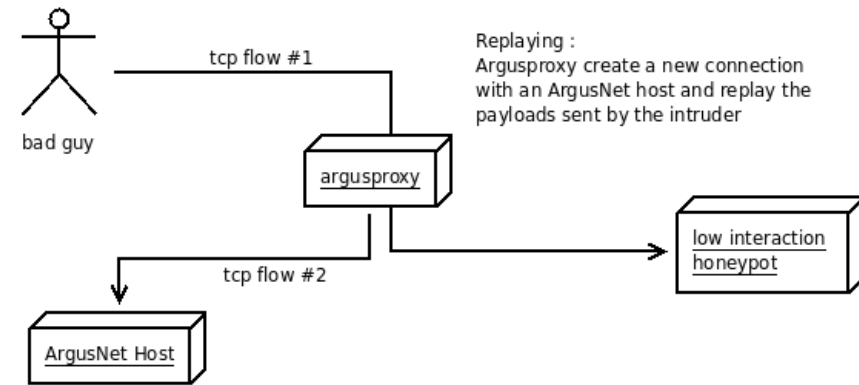
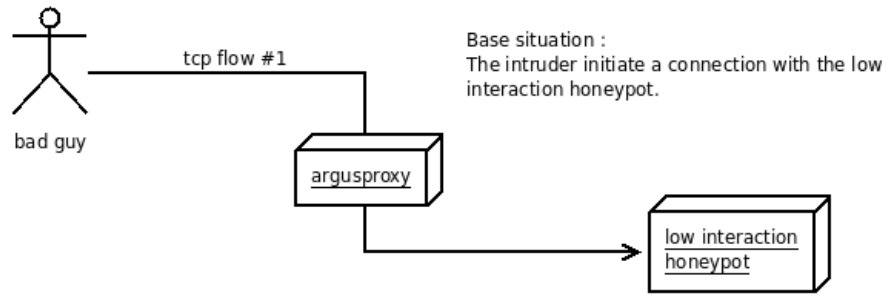
So, I have decided to rely fully on raw sockets and using fully crafted connections using my own packets. Rebuilding a complete network engine from root using raw packets could have been a good idea, but the Linux kernel doesn't like when its not aware of the creation of a connection. When I try to create a TCP handshake, my SYN packet is sent and receive by the destination, that reply with a SYN/ACK packet but then the linux kernel send a RST packet to the destination, because for it the connection doesn't exist. Again, it doesn't works...

Solution chose

The solution I'm actually working on use both raw sockets, Netfilter and regular kernel networks methods. The idea is to create and maintain a TCP connection with the Argusnet host for each redirected connection. Each connection will be represented by a socket stored in the redirected connections binary tree.

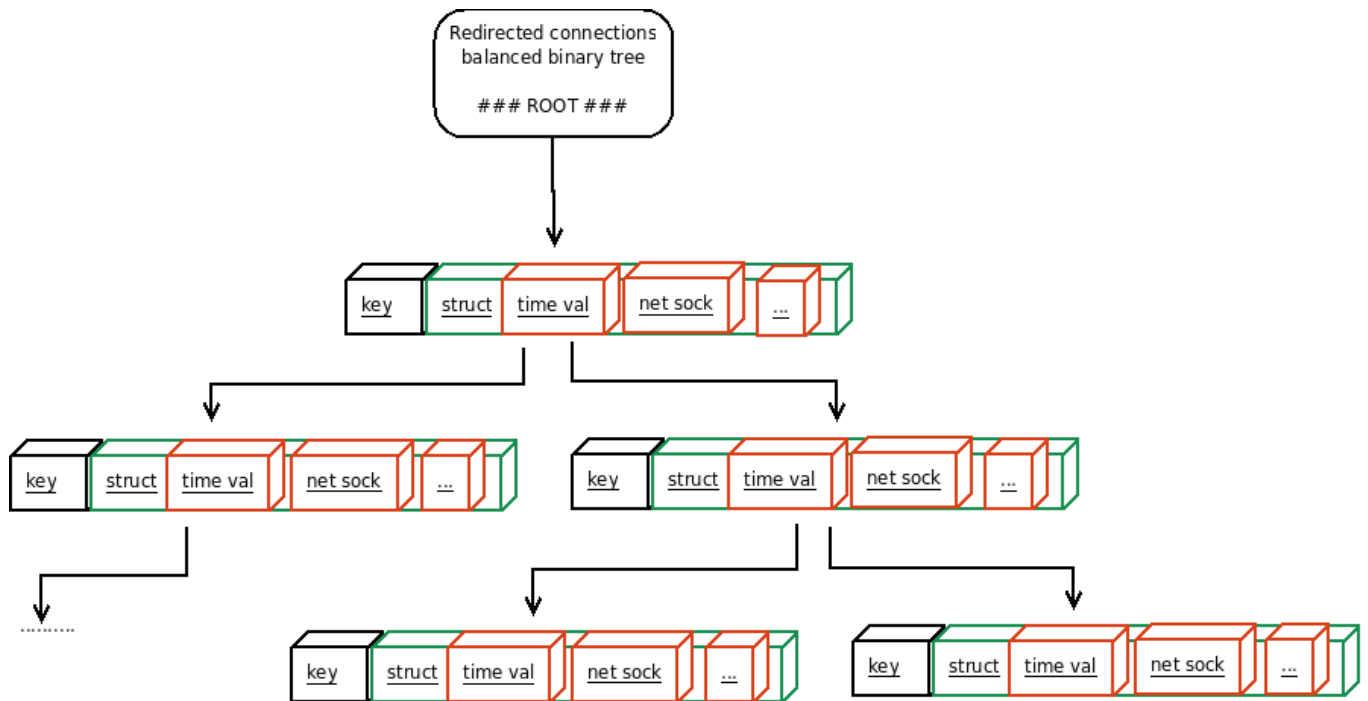
When a packet is received from an attacker, its payload is send to the ArgusNet destination using the active communication. When ArgusProxy receive the response from the Argusnet host, it build a RAW packet (we need to keep some information about the original TCP connection to do so) and send it to the attacker.

The following pictures shows these four steps :



This architecture should work quite efficiently but it obliges us to keep alive a number of TCP connections. I don't know yet if it will be a problem. The Binary Tree will give a quick access to the sockets if the kernel doesn't limit it or close it when it is called from another memory space....

The B-Tree will look like that :

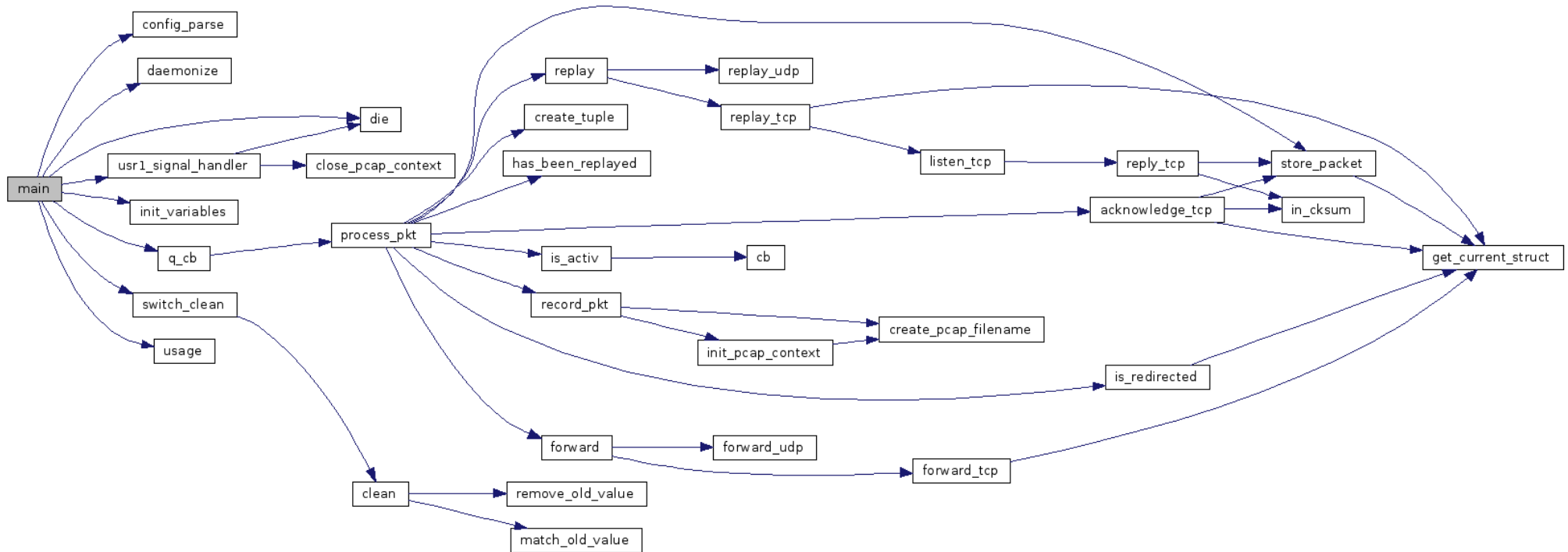


“struct” is the data part of an entry, it will contain a time value (to delete out of date entries) and the network socket. If we need to add more information in this structure, it will still be possible to do it after.

Time table

I thought the network engine will take me less time than that, but with the better vision I have now, I think I will need 3 weeks to do it completely if everything goes well. This means that a beta version of ArgusProxy could not be installed before the beginning of September, because I also need to code a basic decision engine.

D ⇒ ArgusProxy source code



Process_pkt

```

/*! process_pkt
*
\brief Function called for each received packet, this is the core of the redirection engine
\param[in] tb a Netfilter Queue structure that contain both the packet and the metadatas
\return statement = 0 if the packet should be accept or 1 if the packet has to be dropped
*/
static u_int32_t process_pkt (struct nfq_data *tb)
{
    int statement = 1;
    char *nf_packet;

    /*! extract ip header from packet payload
    */
    nfq_get_payload(tb, &nf_packet);

    struct iphdr *iph = ((struct iphdr *) nf_packet);

    /*! compute the two tuples key_one and key_two
    *
    * if packet is not TCP nor UDP, return dontcare = 1
    */
    int dontcare = create_tuple(nf_packet);

    if (dontcare == 0)
    {
        /*! Is the packet a part of a redirected connection ??? */
        if ( is_redirected() == 0 )
        {
            /*! in a redirected connection, do not process packets
            * from the low-interaction honeypot network
            *
            * just drop them
            */
            if (NULL == strstr( inet_ntoa(iph->saddr), (char *) g_hash_table_lookup(config, "net_prefix")) )
            {

                /*! store actual packet
                */
                if ( 0 != store_packet(nf_packet, NULL, NULL))
                {
                    syslog(LOG_ERR | LOG_USER, "%s", "MAIN : Unable to store the packet in memory");
                    g_print("Unable to store the packet in memory \n");
                    return -1;
                }

                /*! if packet need to be acknowledged, do it
                */
                if( iph->protocol == 6)
                {
                    struct tcphdr *tcph = (struct tcphdr *) (nf_packet + iph->ihl*4);

                    if(tcph->ack == 1 && tcph->psh == 1)
                        acknowledge_tcp(nf_packet);
                }

                /*! The connection is redirected but
                * if it has not been replayed already,
                * replay it now
                */
                if ( has_been_replayed() != 0 )
                {

                    /*! replay the connection
                    */
                    if ( ( replay(iph->protocol) ) != 0 )
                    {
                        syslog(LOG_WARNING | LOG_USER, "%s", "Error while replaying connection");
                    }
                }
            }
            else {
                /*! otherwise forward it to its honeypot destination
                * get the reply from the forwarded destination

```

```

        * and send it to the source in a raw packet
        */
        if ( 0 != ( forward(iph->protocol, nf_packet) ) )
        {
            syslog(LOG_WARNING | LOG_USER, "%s", "Forwarding failed");
        }
    }

    /*! if record mode is activated
    */
    if ( NULL != strstr(g_hash_table_lookup(config, "record_redirected"), "1" ) )
    {
        /*! record packet in pcap format in mode 'redirected' */
        if ( 0 != record_pkt(tb, NULL, 1) )
        {
            syslog(LOG_ERR | LOG_USER, "%s", "MAIN : Pcap recording failed !!!");
            g_print("Unable to record the redirected packet in PCAP format\n");
            return -1;
        }
    }

    }

    /*! switch statement to 0 to drop the packet
    */
    statement = 0;
}

/*! Is the packet part of an activ connection ? (0 = yes)*/
else
{
    if ( is_activ(nf_packet) == 0 )
    {
        /*! forward packet to dest
        if ( 0 != ( forward(m, redirect_to_dest) ) )
        {
            syslog(LOG_WARNING | LOG_USER, "%s", "Forwarding failed to destination ", redirect_to_dest);
        }
        */
    }

    /*! if record mode is activated
    */
    if ( NULL != strstr(g_hash_table_lookup(config, "record"), "1" ) )
    {
        /*! record packet in pcap format */
        if ( 0 != record_pkt(tb, NULL, 0) )
        {
            syslog(LOG_ERR | LOG_USER, "%s", "MAIN : Pcap recording failed !!!");
            g_print("Unable to record the packet in PCAP format !!!\n");
            return -1;
        }
    }

    /*! store packet in memory to replay the connection later
    */
    if ( 0 != store_packet(nf_packet, NULL, NULL) )
    {
        syslog(LOG_ERR | LOG_USER, "%s", "MAIN : Unable to store the packet in memory");
        g_print("Unable to store the packet in memory \n");
        return -1;
    }

    }

    /*! free keys memory */
    g_string_free(key_one, FALSE);
    g_string_free(key_two, FALSE);

    return statement;
}

```

Is_activ

```

///! Is_activ function, Search if the packet is part of an active connection in the netfilter conntrack table
///!
\param[in] nf_packet is the raw packet
\return 0 if found, anything else if not
///
int is_activ(char *nf_packet)
{
    struct nf_conntrack *ct;
    struct nfct_handle *cth;
    struct tcphdr *tcp;
    struct udphdr *udp;

    ///! default value is -1 => state not active ///
    int conn_state = -1;

    ///! extract ip header from packet payload ///
    struct iphdr *iph = ((struct iphdr *) nf_packet);

    g_print( "NETFILTER_CONNTRACK : Connection check called for %s",key_one->str);

    ///! nfct_conntrack_new - allocate a new conntrack
    *
    * In case of success, this function returns a valid pointer to a memory blob,
    * otherwise NULL is returned and errno is set appropriately.
    ///
    ct = nfct_new();

    ///! nfct_set_attr_u[8,16,32] - set the value of a certain conntrack attribute
    *
    \param[in] ct: pointer to a valid conntrack
    \param[in] type: attribute type
    \param[in] value: unsigned [8,16,32] bits attribute value
    ///
    nfct_set_attr_u8(ct, ATTR_ORIG_L3PROTO, AF_INET);
    nfct_set_attr_u32(ct, ATTR_ORIG_IPV4_SRC, iph->saddr);
    nfct_set_attr_u32(ct, ATTR_ORIG_IPV4_DST, iph->daddr);
    nfct_set_attr_u8(ct, ATTR_ORIG_L4PROTO, iph->protocol);

    int dontcare = 0;
    switch(iph->protocol)
    {
        case 6 :
            tcp = (struct tcphdr *) (nf_packet+ iph->ihl*4);

            nfct_set_attr_u16(ct, ATTR_ORIG_PORT_SRC, tcp->source);
            nfct_set_attr_u16(ct, ATTR_ORIG_PORT_DST, tcp->dest);
            break;

        case 17 :
            udp = (struct udphdr *) (nf_packet + iph->ihl*4);

            nfct_set_attr_u16(ct, ATTR_ORIG_PORT_SRC, udp->source);
            nfct_set_attr_u16(ct, ATTR_ORIG_PORT_DST, udp->dest);
            break;

        default :
            ///! don't care about this packet ///
            dontcare = 1;
    }

    ///! continue with packet only if protocol is UDP or TCP ///
    if ( dontcare == 0 )
    {
        ///! nfct_open - create a conntrack handler
        ///
        cth = nfct_open(CONNTRACK, 0);

        if (!cth)
            syslog(LOG_WARNING | LOG_USER, "%s","Can't open Conntrack handler");

        ///! nf_callback_register - register a callback
        *

```

```

\param[in] cth: library handler
\param[in] NFCT_T_ALL : message type
\param[in] cb: callback used to process contrack received
\param[in] NULL : data used by the callback, if any.
*
* This function register a callback to handle the contrack received,
* in case of error -1 is returned and errno is set appropriately, otherwise
* 0 is returned.
*
* Note that the data parameter is optional, if you do not want to pass any
* data to your callback, then use NULL.
*/
if ( 0 != nfct_callback_register(cth, NFCT_T_ALL, cb, NULL))
    syslog(LOG_WARNING | LOG_USER, "%s", "Failure in callback function");

/*! nfct_query - send a query to ctnetlin
\param[in] h: library handler
\param[in] NFCT_Q_GET: query type
\param[in] ct: data required to send the query
*
* For query types:
*   NFCT_Q_CREATE: add a new contrack, if it exists, fail
*   NFCT_Q_CREATE_UPDATE: add a new contrack, if it exists, update it
*   NFCT_Q_UPDATE: update a contrack
*   NFCT_Q_DESTROY: destroy a contrack
*   NFCT_Q_GET: get a contrack
*
* On error, -1 is returned and errno is explicitly set. On success, 0
* is returned.
*/
errno = 0;
conn_state = nfct_query(cth, NFCT_Q_GET, ct);

/*! close the handler */
nfct_close(cth);
}

if (conn_state == 0)
    g_print( " ==> ESTABLISHED\n");

else
    g_print( " ==> NEW\n");

/*! return state value */
return conn_state;
}

```

Store_packet

```

/*! store_packet function
  \brief Store the current packet as part of the connection to replay it later. If this is the first
  packet of a communication, init its structure in the main B-Tree.
  *
  \param[in] m is the IP raw packet
  \param[in] mode, set to one if the another context than the current one must be used
  *
  \return 0 in case of success, anything else otherwise
  */
int store_packet(char *nf_packet, int mode, char alt_key[43])
{
    /*! copy the packet
    */
    struct iphdr *printip = (struct iphdr *) nf_packet;
    gpointer *store = malloc(ntohs(printip->tot_len));

    /*! the b-tree will index the position of the "store" value in memory
    *
    * so don't free it !!!
    */
    memcpy(store, nf_packet, ntohs(printip->tot_len));

    /*!
    * search the entry
    */
    if (
        TRUE != g_tree_lookup_extended(conn_tree, key_one->str, NULL, NULL)
        &&
        TRUE != g_tree_lookup_extended(conn_tree, key_two->str, NULL, NULL)
    )
    {
        /*! if doesn't exist, create it and init the whole structure
        * as a value for this entry
        */
        struct conn_struct *add_new_data;
        add_new_data = malloc( sizeof(*add_new_data) );

        /*! get current time
        */
        gint *curtime;
        struct tms current;
        curtime = g_strdup_printf("%d", times(&current));

        /*! fill the structure
        */
        add_new_data->access_time = curtime;
        add_new_data->redirected = 0;
        add_new_data->hasbeenreplayed = 0;
        add_new_data->socket = 0;
        add_new_data->rawsocket = 0;
        add_new_data->recordlist = NULL;

        /*! store the address of the payload as a new entry of the list
        */
        add_new_data->recordlist = g_slist_append(add_new_data->recordlist, store);

        /*! add the list to the tree, value contain the address of the first entry of the list
        */
        g_tree_insert(conn_tree, key_one->str, add_new_data);

        g_print("STORING FUNCTION : entry created for %s\n",key_one->str);
    }
    else {
        /*! the tuple already exist in the B-Tree
        *
        * add the packet at the end list and update the time value
        */

        /*! if we don't use the current context
        */
        if( mode == 1)

```

```

{
    /*! find the good entry in the b-tree
    */
    struct conn_struct * connection_data;

    if (TRUE != g_tree_lookup_extended(conn_tree, alt_key, NULL, (gpointer *) &connection_data))
        g_print("TABLE : Wrong B-Tree key in alternativ context\nkey = %s\n", alt_key);

    /*! store the packet
    */
    connection_data->recordlist = g_slist_append(connection_data->recordlist, store);

    g_print("STORING FUNCTION : %dst packet of %s stored in
memory\n", g_slist_length(connection_data->recordlist), alt_key);
}
else
    /*! use current context
    */
    if( 0 == get_current_struct() )
    {
        current_connection_data->recordlist = g_slist_append(current_connection_data->recordlist,
store);

        g_print("STORING FUNCTION : %dst packet of %s stored in
memory\n", g_slist_length(current_connection_data->recordlist), key->str);

        /*! for test purpose only !!!
        *
        * when the fourth packet of a connection is reached, ask for a redirection
        * next time a packet is seen
        */

        if ( g_slist_length(current_connection_data->recordlist) == 17 )
            current_connection_data->redirected = 1;
    }
}

return 0;
}

```

B-Tree cleaning functions

```

/*! match_old_value function : called for each entry in the B-Tree, if a time value is upper to 5
minutes, entry is deleted
\param[in] key, a pointer to the current B-Tree key value
\param[in] value, a pointer to the current B-Tree associated value
\param[in] trash, user data, unused
\return FALSE, to continu to traverse the tree (if TRUE is returned, traversal is stopped)
*/
int match_old_value(gpointer key, struct conn_struct *cur_conn, gpointer trash)
{
    struct tms actual;

    gint *valtime;
    valtime = g_strdup_printf("%d",times(&actual));

    if( (valtime - cur_conn->access_time) > 30000 )
    {
        g_ptr_array_add(entrytoclean, key);
    }
    return FALSE;
}

/*! remove_old_value function : called for each entry in the pointer array, each entry is a key that is
deleted from the B-Tree
\param[in] key, a pointer to the current B-Tree key value stored in the pointer table
\param[in] trash, user data, unused
*/
void remove_old_value(gpointer key, gpointer trash)
{
    g_print("REDIRECT TABLE CLEANER : entry %s removed\n", (char *) key);
    if (TRUE != g_tree_remove(conn_tree,key))
    {
        g_print("REDIRECT TABLE CLEANER : error while removing %s !\\ KEY NOT FOUND IN B-TREE !\\
\n", (char *) key);
    }
}

/*! watchman for the b_tree, wake up every 5 minutes and check every entries */
void clean()
{
    while (1)
    {
        /*! wake up every 5 minutes */
        sleep(300);

        g_print("Cleaning process started\n");

        /*! First, clean the redirect table
******/
        entrytoclean = g_ptr_array_new();

        /*! call the clean function for each value, delete the value if TRUE is returned*/
        g_tree_traverse( conn_tree,(GHRFunc) match_old_value, G_IN_ORDER, NULL );

        /*! remove each key listed from the btree */
        g_ptr_array_foreach(entrytoclean, (GFunc) remove_old_value, NULL);

        /*! free the array */
        g_ptr_array_free(entrytoclean, TRUE);

        /*! Second, clean what ???
******/
    }
}

```


Record_pkt

```

/*! record_pkt
 *
 *brief record a packet in the current pcap file descriptor
 *
 *param[in] nfq_data *tb, raw packet ( used with nfqueue)
 *param[in] *payload, packet to record (used outside of nfqueue)
 *param[in] mode, 0 for non redirected connection, 1 for redirected connections, 2 for redirected outside
nfqueue
 *
 *return 0 on success, anything else otherwise
 */
int record_pkt(struct nfq_data *tb, char *p, int mode)
{
    /*! if pcap desc doesn't exist, init pcap context
    */
    if (!pcap_main_desc)
    {
        if (0 != init_pcap_context())
        {
            syslog(LOG_ERR | LOG_USER, "%s", "PCAP_TOOL : Error while initializing pcap");
            g_print("exit with error code -1\n");
            return -1;
        }
    }

    pcap_dumper_t *DumpDescriptor;

    /*! switch the descriptor regarding to the mode (redirected or not)
    */
    if (mode == 0 )
        DumpDescriptor = pcap_output_current;
    else
        DumpDescriptor = pcap_output_redirected;

    /*! if the actual pcap output file is bigger than 10mo, create a new one
    */
    if (ftell((FILE *)DumpDescriptor) > 10485760){

        /*! close the current descriptor */
        pcap_dump_close(DumpDescriptor);

        /*! create output filename based on the conf and the time value */
        GString *file_name;
        file_name = create_pcap_filename(mode);

        /*! open the new file descriptor
        */
        if (NULL == (DumpDescriptor = pcap_dump_open(pcap_main_desc, file_name->str)))
        {
            syslog(LOG_ERR | LOG_USER, "%s", "PCAP_TOOL : Error while creating pcap output file");
            g_print("exit with error code -1\n");
            return -1;
        }
        g_print("PCAP_TOOL : file descriptor opened at %s\n", file_name->str);

        /*! store new descriptor in global descriptor
        */
        if (mode == 0 )
            pcap_output_current = DumpDescriptor;
        else
            pcap_output_redirected = DumpDescriptor;
    }

    /*! create pcap specific header
    */
    struct pcap_pkthdr phdr;

    GTimeVal t;
    g_get_current_time(&t);

    phdr.ts.tv_sec      = t.tv_sec;
    phdr.ts.tv_usec    = t.tv_usec;

```

```
if( mode == 2)
{
    /*! mode 2 is used when we need to record a packet received outside of
    * the netfilter queue
    */
    struct iphdr *ip = (struct iphdr *) p;
    phdr.caplen = ntohs(ip->tot_len); /*! +1 because the '\0' is not included */
    phdr.len = phdr.caplen;

    /*! pcap_dump, write pcap header and packet data to the output file
    \param[in] pcap_output_current, descriptor to the current output file
    \param[in] &phdr, pcap header
    \param[in] payload, packet data
    */
    pcap_dump(DumpDescriptor, &phdr, (const u_char *)p);
}
else
{
    char *nf_packet;
    phdr.caplen = nfq_get_payload(tb, &nf_packet);
    phdr.len = phdr.caplen;

    /*! pcap_dump, write pcap header and packet data to the output file
    \param[in] pcap_output_current, descriptor to the current output file
    \param[in] &phdr, pcap header
    \param[in] (const u_char *)nf_packet, packet data from netfilter queue
    */
    pcap_dump(DumpDescriptor, &phdr, (const u_char *)nf_packet);
}

g_print("PCAP_TOOL : Current packet writed to output pcap file (mode = %d)\n", mode);

return 0;
}
```

Netcode functions

```

/*! \file netcode.c
    \brief Network functions file

    \Author J. Vehent
*/

#include <glib.h>
#include <syslog.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <linux/netfilter.h>
#include <libnetfilter_queue/libnetfilter_queue.h>

#include "netcode.h"
#include "tables.h"

/*! in_cksum
    \brief compute the internet checksum (TCP & UDP)
    *
    \param[in] ptr, a layer 4 packet + payload
    \param[in] n16bitwords, size of the data in 16bits words
    *
    \return answer, a 16 bits checksum
*/
unsigned short in_cksum(unsigned short *ptr,int n16bitwords)
{
    register long sum;
    u_short oddbyte;
    register u_short answer;

    sum = 0;
    while (n16bitwords > 0) {
        sum += *ptr++;
        n16bitwords--;
    }

    /*! padding
    */
    if (n16bitwords > 0) {
        oddbyte = 0;
        *((u_char *) &oddbyte) = *(u_char *)ptr;
        sum += oddbyte;
    }

    /*!
    * complement to 1
    */
    sum = (sum >> 16) + (sum & 0xffff);
    answer = ~sum;
    return(answer);
}

/*! forward_tcp function
    *
    \brief forward a packet to its destination in the high-interaction honeypot
    *
    * When a connection is replayed to the high-interaction honeypot, the socket
    * used to replay the connection is recorded in the 'replayed_connections' B-Tree
    *
    * This function search for the good socket, get it and use it to forward a received
    * payload from an attacker to argusnet
    *
    * If the packet to forward is a connection close request, the socket is closed
    *
    \return 0 if the connection has been successfully replayed, anything else if not
    */
int forward_tcp(char *nf_packet)
{
    if( 0 != get_current_struct() )
        return -1;
}

```

```

/*! separate the payload
*/
struct iphdr *ip = (struct iphdr *) nf_packet;
struct tcphdr *tcp = (struct tcphdr *) (nf_packet + (ip->ihl << 2));

if (tcp->psh == 1)
{
    char *payload;
    int PAYLOAD_SIZE = ntohs(ip->tot_len) - ((ip->ihl << 2) + (tcp->doff << 2));

    payload = calloc((size_t)1, (size_t) (PAYLOAD_SIZE));

    /*! set the related memory to 0
    */
    memset(payload, 0x0, PAYLOAD_SIZE);

    /*! get the payload from the list
    */
    memcpy(payload,
        nf_packet + (ip->ihl << 2) + (tcp->doff << 2),
        PAYLOAD_SIZE
    );

    int bytes_sent = -1;

    /*! send(), send a message on a socket
    \param[in] s, a socket file descriptor
    \param[in] payload, buffer that contain the packet
    \param[in] PAYLOAD_SIZE, size of the packet
    \param[in] flags, set to 0
    */
    \return the number of bytes sent on succes, a negative value on error
    */
    bytes_sent = send(current_connection_data->socket,
        payload,
        PAYLOAD_SIZE,
        0
    );

    if (bytes_sent < 0)
    {
        syslog(LOG_ERR | LOG_USER, "%s", "NETCODE : Forwarding packet failed");
        g_print("NETCODE: Unable to forward packet to destination\n");
        return -1;
    }
    else
        g_print("NETCODE : packet forwarded to destination (%d bytes sent)\n", bytes_sent);
}
else
{
    /*! if the packet to forward is a close request (FIN flag), close the socket
    */
    */
    if (tcp->fin == 1)
        close(current_connection_data->socket);

    /*! ADD A CONNECTION CLOSING WITH THE SOURCE HERE
    */
}
return 0;
}

/*! listen_tcp
*/
\brief get the data from a tcp connection with a high-interaction honeypot
*/
\param[in] curkey, the key of the binary tree that refer to the connection between the attacker and the
low-interaction honeypot
*/
void listen_tcp(char curkey[43])
{
    int bytes_recv = -1;

    /*!
    \def socket

```

```

*
* copy the socket to a static value because current_connection_data will be switcher to another value
*
\brief store the socket related to this thread
*
\param socket, the socket
*
*/
int socket = current_connection_data->socket;

/*!
\def packet
*
\brief Buffer to store the datas of a packet received from a listening thread
*
\param replied_data[BUFSIZE], datas
*
*/
char replied_data[BUFSIZE];

/*! loop the incoming packets
*/
while ((bytes_rcv = recv(socket,
    replied_data,
    BUFSIZE,
    0)) != -1 )
{
    g_print("NETCODE : reply received from the destination (%d bytes)\n", bytes_rcv);

    if(bytes_rcv > 0)
        /* send the answer to the source in a raw packet
        */
        reply_tcp(replied_data, bytes_rcv, curkey);

    bytes_rcv = -1;
}

/*! reply_tcp
\brief Forward a TCP PSH packet received from argusnet to its destination using a raw tcp packet
*
\param[in] replied_data, the data received
\param[in] PKTSIZE, the number of bytes received
\param[in] curkey, a pointer to the B-Tree key that refer to the connection between the intruder and the
low-interaction honeypot
*
*/
void reply_tcp(char replied_data[BUFSIZE], int PKTSIZE, char curkey[43])
{
    g_print("NETCODE : Send reply in connection %s (size = %d)\n", curkey, PKTSIZE);
    /* search in the records the last IP packet sent from the inside
    * to get the ip information
    */
    struct conn_struct * connection_data;

    if (TRUE != g_tree_lookup_extended(conn_tree, curkey, NULL, (gpointer *) &connection_data))
        g_print("NETCODE : Wrong B-Tree key... unable to reply the packet\nkey = %s\n", curkey);

    int i = g_slist_length(connection_data->recordlist);

    struct iphdr *last_sent_ip = (struct iphdr *) g_slist_nth_data( connection_data->recordlist, --i);

    while(NULL == strstr( inet_ntoa(last_sent_ip->saddr), (char *)
g_hash_table_lookup(config, "net_prefix")))
        last_sent_ip = (struct iphdr *) g_slist_nth_data( connection_data->recordlist, --i);

    /* get the tcp header
    */
    struct tcphdr *last_sent_tcp = (struct tcphdr *) (g_slist_nth_data( connection_data->recordlist, i) +
(last_sent_ip->ihl << 2));

    /* create the new ip header
    */
    struct packet p;
    memset(&p, 0x0, sizeof(struct packet));

    p.ip.version = 4;
    p.ip.ihl = IPHSIZE >> 2;

```

```

p.ip.tos      = 0;
p.ip.tot_len  = htons(PKTSIZE + IPHSIZE + TCPHSIZE);
p.ip.id       = htons(ntohs(last_sent_ip->id) + 1);
p.ip.frag_off = 0;
p.ip.ttl      = last_sent_ip->tll;
p.ip.protocol = IPPROTO_TCP;
p.ip.saddr    = last_sent_ip->saddr;
p.ip.daddr    = last_sent_ip->daddr;
p.ip.check    = 0; /*!filled by the ip stack of the kernel */

/*! create the tcp header (push packet)
 * size is always 20 bytes
 */
p.tcp.source  = last_sent_tcp->source;
p.tcp.dest    = last_sent_tcp->dest;
p.tcp.ack_seq = htonl(0);
p.tcp.doff    = TCPHSIZE >> 2;
p.tcp.urg     = 0;
p.tcp.ack     = 0;
p.tcp.psh     = 1;
p.tcp.rst     = 0;
p.tcp.syn     = 0;
p.tcp.fin     = 0;
p.tcp.window  = last_sent_tcp->window;
p.tcp.check   = 0; /*! set to 0 for later computing */
p.tcp.urg_ptr = 0;

/*! the seq number is the ack_seq number of the last tcp packet received from the attacker
 *
 * so we need to get the last rcv packet
 */
int j = g_slist_length(connection_data->recordlist);
struct iphdr *last_rcv_ip = (struct iphdr *) g_slist_nth_data( connection_data->recordlist, --j);

/*! this time, we search for a packet that doesn't match the strstr condition */
while(NULL != strstr( inet_ntoa(last_rcv_ip->saddr), (char *)
g_hash_table_lookup(config,"net_prefix")))
    last_rcv_ip = (struct iphdr *) g_slist_nth_data( connection_data->recordlist, --j);

/*! get the tcp header
 */
struct tcphdr *last_rcv_tcp = (struct tcphdr *) (g_slist_nth_data( connection_data->recordlist, j) +
(last_rcv_ip->ihl << 2));

/*! Thus, the actual seq number is the former ack_seq number
 */
p.tcp.seq     = last_rcv_tcp->ack_seq ;

/*! add the payload to the raw packet
 */
memcpy(p.payload, replied_data, PKTSIZE);

/*! pseudo tcp header for the checksum computation
 */

struct pseudo_tcp p_tcp;
memset(&p_tcp, 0x0, PSEUDOTCPHSIZE);

p_tcp.saddr   = p.ip.saddr;
p_tcp.daddr   = p.ip.daddr;
p_tcp.mbz     = 0;
p_tcp.ptcl    = IPPROTO_TCP;
p_tcp.tcpl    = htons(TCPHSIZE + PKTSIZE);

memcpy(&p_tcp.tcp, &p.tcp, TCPHSIZE);

/*! add the payload to the pseudo header
 */
memcpy(p_tcp.payload, replied_data, PKTSIZE);

/*! in_cksum
 \brief compute the tcp checksum
 *
 \param[in] ((TCPHSIZE + PKTSIZE + PSEUDOTCPHSIZE) /2) + ((TCPHSIZE + PKTSIZE + PSEUDOTCPHSIZE) %2) the
 size of the data to compute
 */

```

```

    \return check, a 16 bits checksum
    */
    p.tcp.check = in_cksum((unsigned short *)&p_tcp, ((TCPHSIZE + PKTSIZE + PSEUDOTCPHSIZE) >> 1) +
((TCPHSIZE + PKTSIZE + PSEUDOTCPHSIZE) % 2));

    /* reuse the raw socket created to ack the first packet after the redirection
    * to send this raw packet to its destination
    */
    int bytes_sent = sendto(connection_data->rawsocket,
        &p,
        IPHSIZE + TCPHSIZE + PKTSIZE,
        0,
        (struct sockaddr *) &connection_data->rawsin,
        sizeof (connection_data->rawsin)
    );

    if (bytes_sent < 0)
    {
        g_print("NETCODE : error while replying packet\n-----\n");
        g_print("ERENNO == %d\n", errno);
    }
    else
    {
        g_print("NETCODE : packet replied to its destination, size = %d bytes\n", bytes_sent);
    }

    /* store the ack packet in memory
    */
    char pkt[IPHSIZE + TCPHSIZE + PKTSIZE];
    memcpy(pkt, (char *) &p, IPHSIZE + TCPHSIZE + PKTSIZE);

    if ( 0 != store_packet(pkt, 1, curkey))
    {
        syslog(LOG_ERR | LOG_USER, "%s", "NETCODE : Unable to store the sent packet");
        g_print("Error while trying to store the raw ack packet sent in to the attacker\n");
    }

    /* record packet in pcap format in mode 'redirected w/o nfqueue' */
    if (0 != record_pkt(NULL, pkt, 2))
    {
        syslog(LOG_ERR | LOG_USER, "%s", "NETCODE : Pcap recording failed !!!");
        g_print("Unable to record the replied packet in PCAP format\n");
    }

    g_print("-----\n");
}

/* acknowledge_tcp
*
* \brief Sent a raw built acknowledgment to the attacker
*
* \param[in] nf_packet, the entire packet received by the netfilter queue
*
*/
void acknowledge_tcp(char *nf_packet)
{
    struct iphdr *recv_ip = (struct iphdr *) nf_packet;
    struct tcphdr *recv_tcp = (struct tcphdr *) (nf_packet + (recv_ip->ihl << 2));

    /* search in the records the last IP packet sent from the inside
    * to get the ip information
    */
    if( 0 != get_current_struct() )
        g_print("NETCODE : Unable to get current struct\n");

    g_print("NETCODE : Acknowledge packet for connection %s\n", key->str);

    int i = g_slist_length(current_connection_data->recordlist) - 1 ;

    struct iphdr *last_sent_ip = (struct iphdr *) g_slist_nth_data( current_connection_data->recordlist,
i);

    while( NULL == strstr((char *)inet_ntoa(last_sent_ip->saddr), (char *)
g_hash_table_lookup(config, "net_prefix")))
        last_sent_ip = (struct iphdr *) g_slist_nth_data( current_connection_data->recordlist, --i);
}

```

```

struct tcphdr *last_sent_tcp = (struct tcphdr *) (g_slist_nth_data( current_connection_data-
>recordlist, i) + (last_sent_ip->ihl << 2));

/* create the ip header
*/
struct packet p;
memset(&p, 0x0, sizeof(struct packet));

p.ip.version = 4;
p.ip.ihl = (IPHSIZE >> 2);
p.ip.tos = 0;
p.ip.tot_len = htons(IPHSIZE + TCPSIZE);
p.ip.id = htons(ntohs(last_sent_ip->id) + 1);
p.ip.frag_off = 0;
p.ip.ttl = last_sent_ip->ttl;
p.ip.protocol = IPPROTO_TCP;
p.ip.saddr = last_sent_ip->saddr;
p.ip.daddr = recv_ip->saddr;
p.ip.check = 0; /*filled by the ip stack of the kernel */

/* create the tcp header (ack packet)
* size is always 20 bytes
*/
p.tcp.source = recv_tcp->dest;
p.tcp.dest = recv_tcp->source;
p.tcp.seq = recv_tcp->ack_seq;

/* take the received seq number, add the size of the datas received and use it as ack_seq
*/
p.tcp.ack_seq = htonl((unsigned long)ntohl(recv_tcp->seq) + (unsigned long)(ntohs(recv_ip->tot_len)
- (recv_ip->ihl << 2) + (recv_tcp->doff << 2)));

/* this is a very basic tcp header, just 20 bytes for the ack packet
*/
p.tcp.doff = TCPSIZE >> 2;
p.tcp.urg = 0;
p.tcp.ack = 1;
p.tcp.psh = 0;
p.tcp.rst = 0;
p.tcp.syn = 0;
p.tcp.fin = 0;
p.tcp.window = last_sent_tcp->window;
p.tcp.check = 0;
p.tcp.urg_ptr = 0;

/* pseudo tcp header for the checksum computation
*/
struct pseudo_tcp p_tcp;
memset(&p_tcp, 0x0, sizeof(struct pseudo_tcp));

p_tcp.saddr = last_sent_ip->saddr;
p_tcp.daddr = recv_ip->saddr;
p_tcp.mbz = 0;
p_tcp.ptcl = IPPROTO_TCP;
p_tcp.tcpl = htons(TCPSIZE);
memcpy(&p_tcp.tcp, &p.tcp, TCPSIZE);

/* compute the tcp checksum
*
* TCPSIZE is the size of the tcp header
* PSEUDOTCPHSIZE is the size of the pseudo tcp header
* we divide by 2 because the checksum is computed on 16 bits words and not 8 bits
*/
p_tcp.check = in_cksum((unsigned short *)&p_tcp, ((TCPSIZE + PSEUDOTCPHSIZE) >> 1) + ((TCPSIZE +
PSEUDOTCPHSIZE) % 2));

/* create the raw socket if not exist
*/
if ( current_connection_data->rawsocket == 0)
{
    g_print("NETCODE : create the raw socket\n");
    current_connection_data->rawsocket = socket (PF_INET, SOCK_RAW, IPPROTO_TCP);

    int on=1;
    if (setsockopt(current_connection_data->rawsocket, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) <

```



```

0)
    g_print("NETCODE : Unable to free the raw tcp socket (setsockopt error)\n");

    current_connection_data->rawsin.sin_family    = AF_INET;
    current_connection_data->rawsin.sin_port     = p.tcp.dest;
    current_connection_data->rawsin.sin_addr.s_addr = p.ip.daddr;
}

/*! send the packet
*/
int bytes_sent = sendto(current_connection_data->rawsocket,
    &p,
    ntohs(p.ip.tot_len),
    0,
    (struct sockaddr *) &current_connection_data->rawsin,
    sizeof (current_connection_data->rawsin)
    );

if (bytes_sent < 0)
    g_print("NETCODE : error while acknowledging packet\n");
else
    g_print("NETCODE : acknowledgment sent, size = %d bytes\n",bytes_sent);

/*! store the ack packet in memory
*/
char ackpkt[ntohs(p.ip.tot_len)];
memcpy(ackpkt, (char *) &p, ntohs(p.ip.tot_len));

if ( 0 != store_packet(ackpkt, NULL, NULL))
{
    syslog(LOG_ERR | LOG_USER, "%s","NETCODE : Unable to store the acknowledgment sent");
    g_print("Error while trying to store the raw ack packet sent in to the attacker\n");
}

/*! record packet in pcap format in mode 'redirected w/o nfqueue' */
if (0 != record_pkt(NULL, ackpkt, 2))
{
    syslog(LOG_ERR | LOG_USER, "%s","NETCODE : Pcap recording failed !!!");
    g_print("Unable to record the acknowledgment packet in PCAP format\n");
}

}

int forward_udp(char *nf_packet)
{
    g_print("UDP forwarding is not available yet...\n");
    return 0;
}

/*! replay_tcp function
*/
\brief replay a recorded tcp connection to an identified host
*/
* When the engine wants to replay a tcp connection, this function is called.
* It search the recorded packets list in the B-Tree, then it search for a destination
* in the argusnet correspondance list
* A TCP ( SOCK_STREAM ) connection is created with this host and the recorded payloads
* are replayed to it
*
* Then, the socket of this connection is stored in the 'redirected_connection' B-Tree
* in order to reuse it to forward the next packets of the communication
*
* \return 0 if the connection has been successfully replayed, anything else if not
*/
int replay_tcp()
{
    if( 0 != get_current_struct() )
        return -1;

    /*! get the first packet recorded in an ip structure
*/
    struct iphdr *first_ip = (struct iphdr *) g_slist_nth_data(current_connection_data->recordlist, 0);
    struct tcphdr *first_tcp = (struct tcphdr *) (g_slist_nth_data(current_connection_data->recordlist, 0)
+ (first_ip->ihl << 2));

    /*! we need to find which IP:PORT are the attacker's one, so we search

```

```

    * the local network value in the ip packet and store the result in the
    * following variables
    */
gchar *attacker_ip;
int attacker_port, argusnet_port;

if (NULL != strstr( inet_ntoa(first_ip->saddr), (char *) g_hash_table_lookup(config, "net_prefix")))
{
    /*! if the two saddr matches, saddr is the local addr
    *
    * so, we copy in source_ip the attacker address
    */
    attacker_ip = g_strdup_printf("%s", (char *) inet_ntoa(first_ip->daddr));
    attacker_port = ntohs(first_tcp->dest);
    argusnet_port = ntohs(first_tcp->source);
}
else {
    /*! pattern was not found so the attacker IP is the source addr
    */
    attacker_ip = g_strdup_printf("%s", (char *) inet_ntoa(first_ip->saddr));
    attacker_port = ntohs(first_tcp->source);
    argusnet_port = ntohs(first_tcp->dest);
}

/*! We are ready to replay the connection !
*
* init the tcp socket
*/
current_connection_data->socket = socket (PF_INET, SOCK_STREAM, 0);

int on = 1;
if (setsockopt(current_connection_data->socket, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0)
    g_print("NETCODE : Unable to free the tcp socket (setsockopt error)\n");

/*! the sockaddr_in that contains the dest. address is used
* with send() function
*/
current_connection_data->connsin.sin_family = AF_INET;
current_connection_data->connsin.sin_port = htons (argusnet_port);

///this value is for test purpose, should replace it with a research in a correspondance table
current_connection_data->connsin.sin_addr.s_addr = inet_addr ("1.1.1.3");

/*! connect(), create a tcp session
\param[in] s, a socket file descriptor, define the 14 protocol
\param[in] sin, sockaddr structure, contain the dest port and dest ip addr
\param[in] size of the sockaddr structure
*
\return -1 on error
*/
if( -1 == (connect( current_connection_data->socket,
    (struct sockaddr *) &current_connection_data->connsin,
    sizeof (struct sockaddr)))
{
    syslog(LOG_ERR | LOG_USER, "%s", "NETCODE : Unable to connect to target host");
    g_print("error while connecting to host\n");
    return -1;
}

/*! For each packet received from the attacker, get the payload and replay it to
* the high-interaction honeypot
*
* skip packets from the low-interaction honeypot to the attacker
*/
int i = 0;
while( g_slist_nth_data(current_connection_data->recordlist, i) != NULL )
{
    /*! get the packet back
    */
    struct iphdr *ip =
        (struct iphdr *) g_slist_nth_data(current_connection_data->recordlist, i);

    /*! if packet source addr does not match the low-inter honeypot addr, replay it
    */
    if(NULL == strstr( inet_ntoa(ip->saddr), (char *) g_hash_table_lookup(config, "net_prefix")))
    {

```

```

    struct tcphdr *tcp = (struct tcphdr *) (g_slist_nth_data(current_connection_data->recordlist, i)
+ (ip->ihl << 2));

    /*! get payload if tcp push flag is set to 1
    */
    if (tcp->psh == 1)
    {
        char *payload;
        int PAYLOAD_SIZE = ntohs(ip->tot_len) - ((ip->ihl << 2) + (tcp->doff << 2));

        payload = calloc((size_t)1, (size_t)(PAYLOAD_SIZE));

        /*! set the related memory to 0
        */
        memset(payload, 0x0, PAYLOAD_SIZE);

        /*! get the payload from the list
        */
        memcpy( payload,
                g_slist_nth_data(current_connection_data->recordlist, i) + (ip->ihl << 2) + (tcp->doff <<
2),
                PAYLOAD_SIZE
                );

        int bytes_sent = -1;

        /*! send(), send a message on a socket
        \param[in] s, a socket file descriptor
        \param[in] payload, buffer that contain the packet
        \param[in] PAYLOAD_SIZE, size of the packet
        \param[in] flags, set to 0
        *
        \return the number of bytes sent on succes, a negative value on error
        */
        bytes_sent = send( current_connection_data->socket,
                payload,
                PAYLOAD_SIZE,
                0
                );

        if (bytes_sent < 0)
        {
            syslog(LOG_ERR | LOG_USER, "%s", "NETCODE : Packet has not been send... an error has
occured");
            g_print("error while sending packet #d\n",i);
            return -1;
        }
        else
            g_print("replaying : pkt # %d sent, size = %d bytes\n",i,bytes_sent);
    }
    else {
        /*! if a close request is reached, close the connection
        *
        * that should not happend because the connection is supposed
        * to be alive, otherwise there's no point to redirect it
        */
        if (tcp->fin == 1)
            close(current_connection_data->socket);
    }
}

/*! store the ack packet in memory
*/
char pkt[ntohs(ip->tot_len)];
memcpy(pkt, (char *) g_slist_nth_data(current_connection_data->recordlist, i), ntohs(ip->tot_len));

/*! record packet in pcap format in mode 'redirected w/o nqueue' */
if (0 != record_pkt(NULL, pkt, 2))
{
    syslog(LOG_ERR | LOG_USER, "%s", "NETCODE : Pcap recording failed !!!");
    g_print("Unable to record the packet in PCAP format\n");
}
/*! increment to process next packet
*/
i++;

```

```

}

/* job's done, don't redo it later
*/
current_connection_data->hasbeenreplayed = 1;

/* create a thread that will listen to packets from this connection and forward them to
* the attacker
*/
if( 0!= pthread_create (&current_connection_data->thread_listen, NULL, (void *)listen_tcp, key->str) )
{
    g_print("Error while creating the listening thread\n");
    return -1;
}
else
    g_print("NETCODE : Listen_TCP Thread created for this connection\n");

pthread_detach(current_connection_data->thread_listen);

/* return a success value
*/
return 0;
}

int replay_udp()
{
    g_print("UDP replaying is not available yet...\n");
    return 0;
}

int forward(unsigned short proto, char *nf_packet)
{
    /* select TCP or UDP replay mode
    */
    switch(proto)
    {
        case 6 :
            forward_tcp(nf_packet);
            break;

        case 17 :
            forward_udp(nf_packet);
            break;

        default :
            syslog(LOG_ERR | LOG_USER, "%s", "NETCODE : Cannot forward unsupported protocol");
            g_print("You're asking me to forward a packet which's not tcp nor udp... that won't work !\n");
            return -1;
    }
}

return 0;
}

int replay(unsigned short proto)
{
    /* select TCP or UDP replay mode
    */
    switch(proto)
    {
        case 6 :
            replay_tcp();
            break;

        case 17 :
            replay_udp();
            break;

        default :
            syslog(LOG_ERR | LOG_USER, "%s", "NETCODE : Cannot replay unsupported protocol");
            g_print("You're asking me to replay a packet which's not tcp nor udp... that won't work !\n");
            return -1;
    }
}

return 0;
}

```