eForensics Magazine

OPEN

Open Source Forensic Tools

VOL.07 NO.10
ISSUE 10/2018 (101) OCTOBER
ISSN 2300 6986

TEAM

Editor-in-Chief
Joanna Kretowicz
joanna.kretowicz@eforensicsmag.com

Managing Editor:

Dominika Zdrodowska
dominika.zdrodowska@eforensicsmag.com

Aleksandra Solka
aleksandra.solka@eforensicsmag.com

Editors:

Marta Sienicka
sienicka.marta@haking.com

Marta Strzelec
marta.strzelec@eforensicsmag.com

Bartek Adach
bartek.adach@pentestmag.com

Senior Consultant/Publisher:
Paweł Marciniak

CEO:
Joanna Kretowicz
joanna.kretowicz@eforensicsmag.com

Marketing Director:
Joanna Kretowicz
joanna.kretowicz@eforensicsmag.com

DTP
Dominika Zdrodowska
dominika.zdrodowska@eforensicsmag.com

Cover Design
Hiep Nguyen Duc

Publisher
Haking Media Sp. z o.o.
02-676 Warszawa
ul. Postępu 17D
Phone: 1 917 338 3631

www.eforensicsmag.com

word from the team

Dear Readers,

We're proud to present our first free edition in a long time! You were totally in love with our Open Source Tools and Their Developers from 2018, so we decided to revisit the topic of open source forensic tools.

Just like last time, we invited tool developers to prepare tutorials that explain how their tools work and what they can do! You can easily add these open source tools and programs to your toolbox, and learn plenty of new, interesting things. And what's awesome - you can do it for free! Every tool presented can be found on GitHub.

We would also want to thank all authors, developers, reviewers, and proofreaders for contributing to this edition. Without further ado, download the issue and have fun!

Regards,

Dominika Zdrodowska

and the eForensics Magazine Editorial Team

TABLE OF CONTENTS

TSURUGI LINUX

by Giovanni 'sug4r' Rattaro, Marco 'blackmoon' Giorgi, and Davide 'rebus' Gabrini

Introduction

The Tsurugi Linux project is a new open source project that was officially presented in November 2018 at AvTokyo security conference in Japan and this is one of the main reasons for the Japanese name Tsurugi (剣) that refers to a legendary Japanese double-bladed sword used by ancient Japan monks.

The project is mainly focused on DFIR (Digital Forensics & Incident Response) but it's also possible to perform OSINT (Open Source INTelligence) activities, malware analysis and Computer Vision investigations and has been built by a team composed of a bunch of Backtrack and Deft Linux veterans, professionals united by the idea of developing a new DFIR Operating System.

This project is and will be totally free, independent without involving any commercial brand. Our main goal is to share knowledge and "give back to the community".

[v1.0]

Our team is highly motivated and we spent a lot of time building our open source project, testing it and spreading the word travelling all around the world to present talks and technical workshops in many renowned cyber security conferences:

- **AvTokyo** (*3 November 2018 / JAPAN - Tokyo*)
- **BLACKHAT USA** (*8 August 2019 / USA - Las Vegas*)
- **European SANS DFIR Summit** (*30 September 2019 / CZECH REPUBLIC - Prague*)
- **Brucon** (*11 October 2019 / BELGIUM - Ghent*)
- **HackInBO** (*9 November 2019 / ITALY - Bologna*)
- **BLACKHAT EUROPE** (*5 December 2019 / UK - London*)
- **CoRI&IN** (*28 January 2020 / FRANCE - Lille*)

3 projects in 1

The project is divided into three parts, each one with specifics characteristics and different goals:

- Tsurugi Linux
- Tsurugi Acquire
- BENTO Toolkit

TSURUGI LINUX

Tsurugi Linux, also named Tsurugi LAB, is a heavily customized Linux distribution designed to support DFIR investigations, malware analysis and OSINT (Open Source INTelligence) activities.

The system is based on a 64 bits Ubuntu LTS (*Long Time Support*) and we preferred initially to use the 16.04 version to have a stable system with more supported tools, but in our roadmap the major upgrade to the new 20LTS version will be delivered with the release 2020.2 in Q3/Q4. A special "spring

edition" release will be published in March with a full system upgrade, new tools, a few fixes and many tool updates.

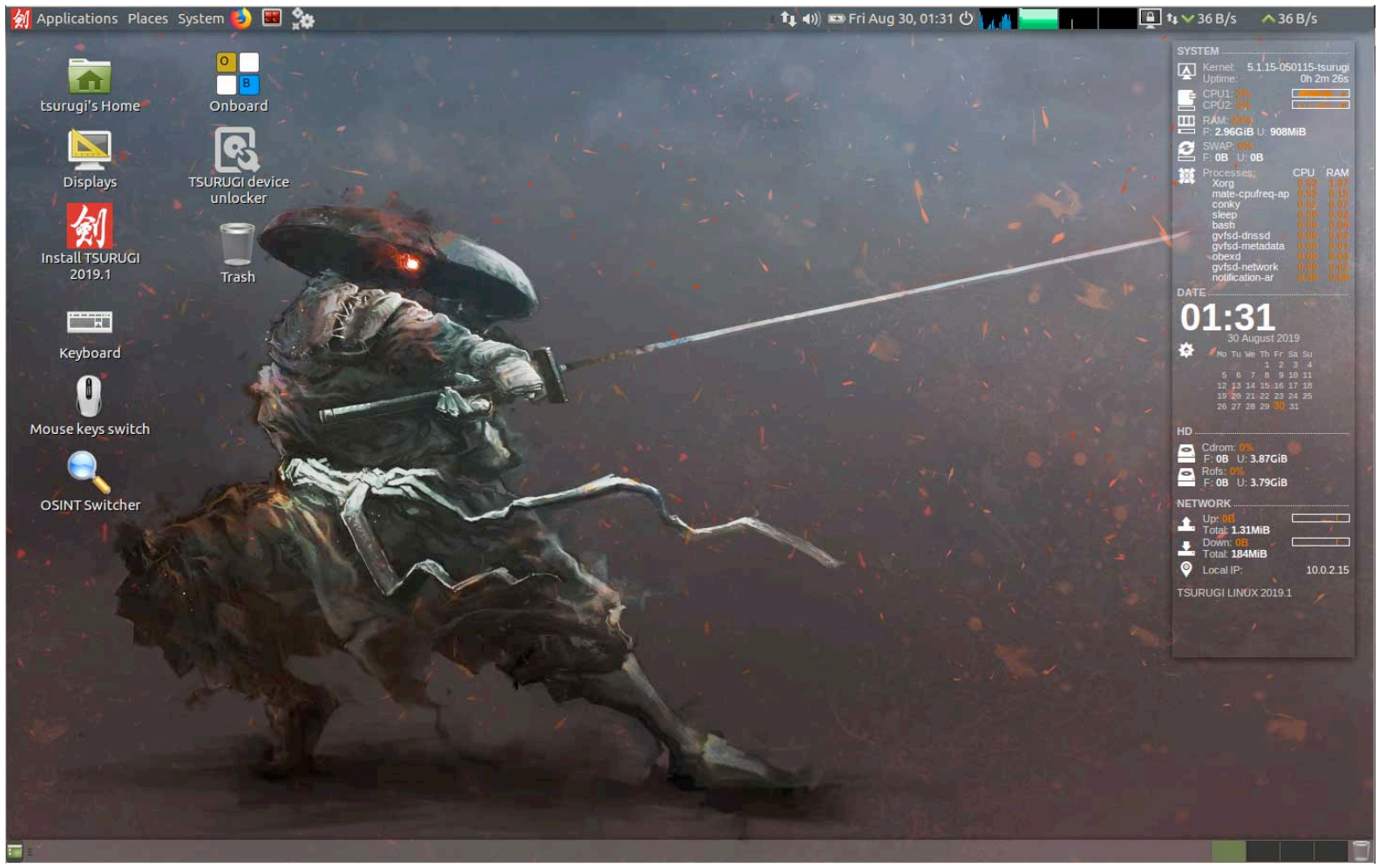
Two repositories (master and development) have been created to be able to deliver bugfixes, improvements and custom updates. Other security updates are guaranteed by the official Ubuntu repositories.

The main idea behind the Tsurugi Linux project is about simplicity but some of the topics can be really complex. Basic Linux skills are mandatory to be able to work correctly and make the most of it.

It's possible to use Tsurugi Linux [LAB] in live mode but its main goal is to be installed and become the default forensics lab.

This is a well-tested project because the first three public releases have been built with about 200 development ISO…

Below is a screenshot of Tsurugi Linux in live mode where many basic but important features are readily available on the main default desktop.

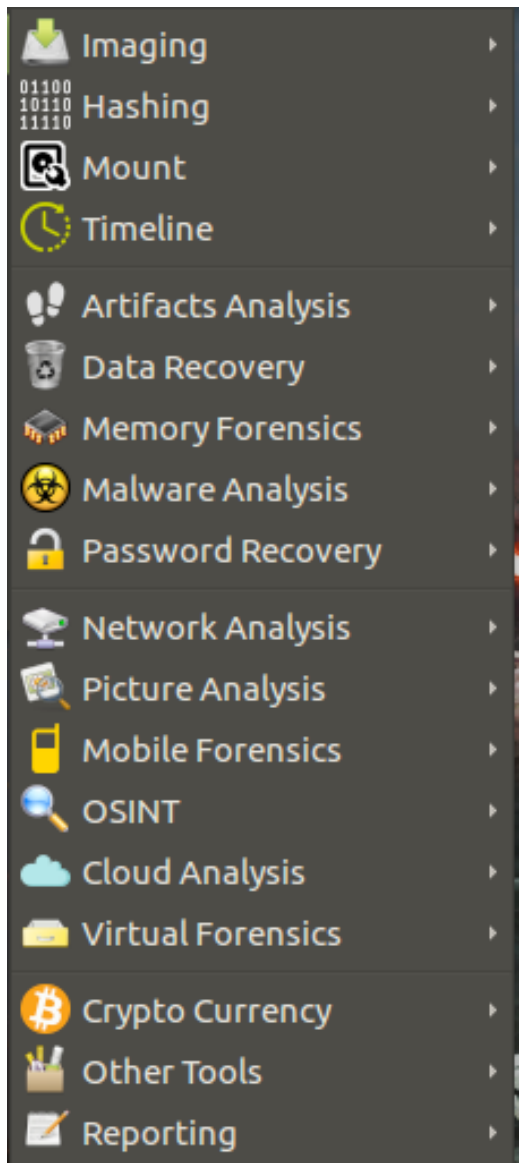


Special features

This distribution includes the latest versions of the most famous tools needed to conduct an in-depth forensic or incident response investigation with several hidden and special features.

Main menu sorting & tools classification

The main Tsurugi menu has been sorted and classified with a particular logic based in part on the six investigation phases (Identification, Preservation, Collection, Examination, Analysis & Presentation) so it's possible to begin the menu and scroll it down during the analysis.



The tools are put in several menus following their specific features, so if a tool has several features it could be present in more than one menu. With this idea it will be possible to find the right tool by searching for a specific investigation task and maybe discovering new specific tools for the same kind of operation.

IMAGING MENU

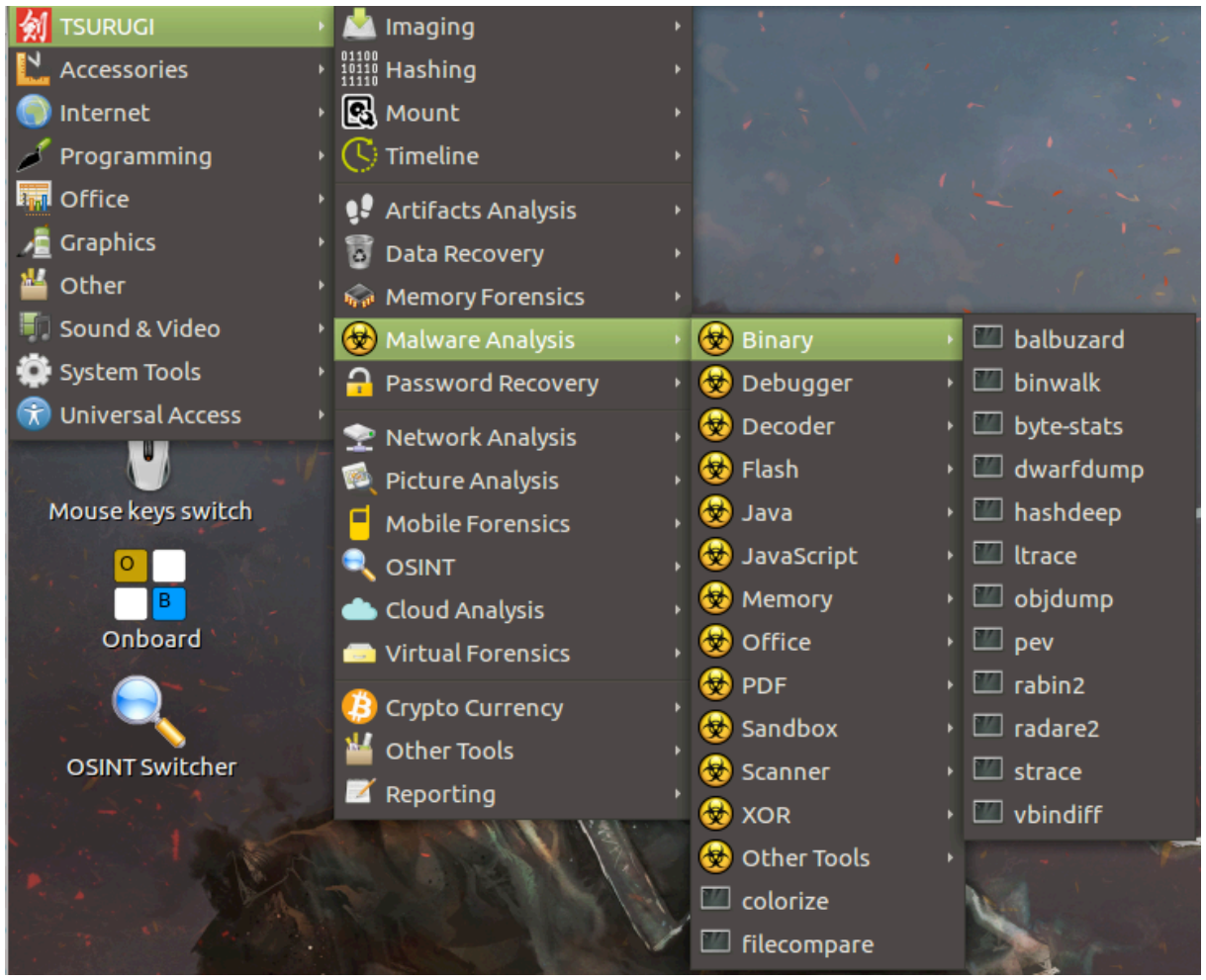
Here are some Tsurugi menus of "IMAGING MENU" where all the tools are present to perform digital forensics copies with several standards like RAW, AFF and EWF.

ARTIFACTS ANALYSIS

Under the "Artifacts Analysis" menu are several sub-categories where it's possible to find many specifics tools.

MALWARE ANALYSIS section

Under the "Malware Analysis" section the tools are organized and filtered following the type of the possible technical analysis.



Kernel write blocker

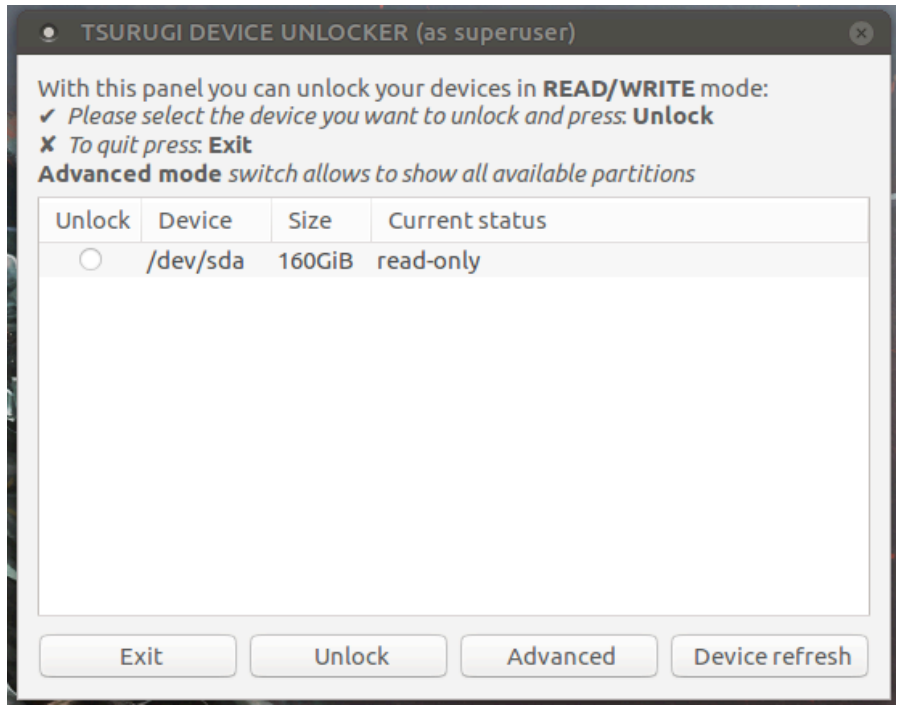
A Write Blocker system has been enforced at kernel level to avoid device write access and integrity alteration, so that every device connected to the system will be in Read Only mode by default. The forensics patched kernel is based on 5.4.2 version that implements many new drivers and features.

Graphical device unlocker

The modified kernel with a forensic patch, that by default locks in Read Only mode every device, has been built with a specific graphical device unlocker to easily unlock in Read/Write mode any connected devices where needed to do modifications.

An advanced mode allows the user to select and unlock also at partition level if needed. Of course, it's possible to perform the same actions using the CLI (Command Line Interface) with the command

"***blockdev***" or the modified script "***wrtblk-disable***" (*tuned to take into account old and new script syntax*).



Default configuration protection

To prevent settings changes, potentially due to future updates, each time a user session starts, all values about "*device automount*", "*device autorun*" and "*system hibernation*" options are put with default customized values.

Automatic set HI-DPI

For high screen resolution with more than 2560 pixels there is a hidden feature that, to avoid having icons that are too small, automatically zooms the whole system to fit the screen resolution. There is an icon on the desktop and inside the graphics menu that allows you to switch at any moment to the original screen resolution.

Customized boot option

In live mode, during the boot phase, it's possible to decide to start Tsurugi Linux in graphical or CLI mode, both directly on system or totally loaded in RAM memory. To avoid any possible graphical hardware problems it's also possible to disable specific graphic drivers in case of visualization problems or potential crashes directly in the customized grub menu.

OSINT profile switcher

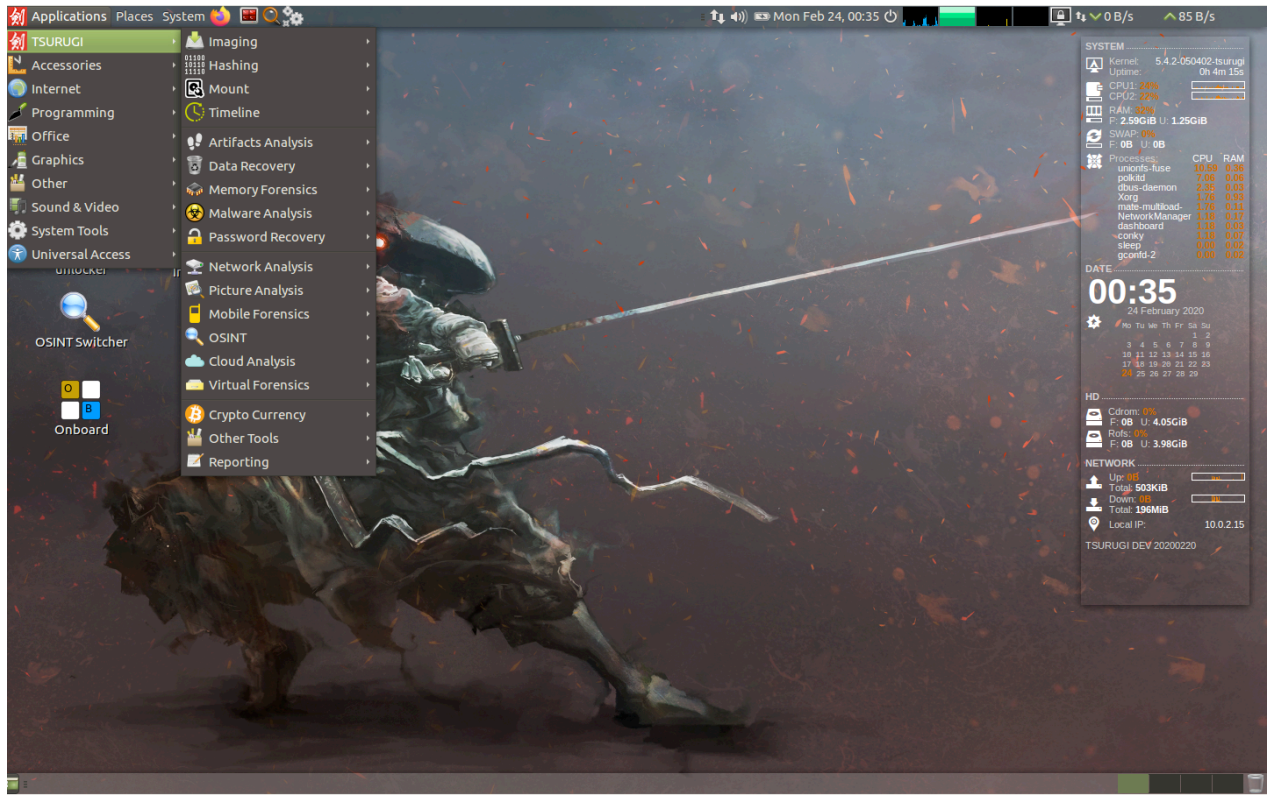
The OSINT Profile Switcher (***profile_switcher_tsurugi***) is a feature that allows users to quickly switch from DFIR to OSINT profile and put in evidence these last tools just by clicking on a desktop icon.

The difference is that the Tsurugi menu becomes lighter because only a few categories are useful for OSINT activities. To easily differentiate the two profiles, the default wallpaper also changes.

A hidden feature allows users to reset all menus and wallpapers (DFIR and OSINT profiles) to default values (command line: **profile_switcher_tsurugi default**).

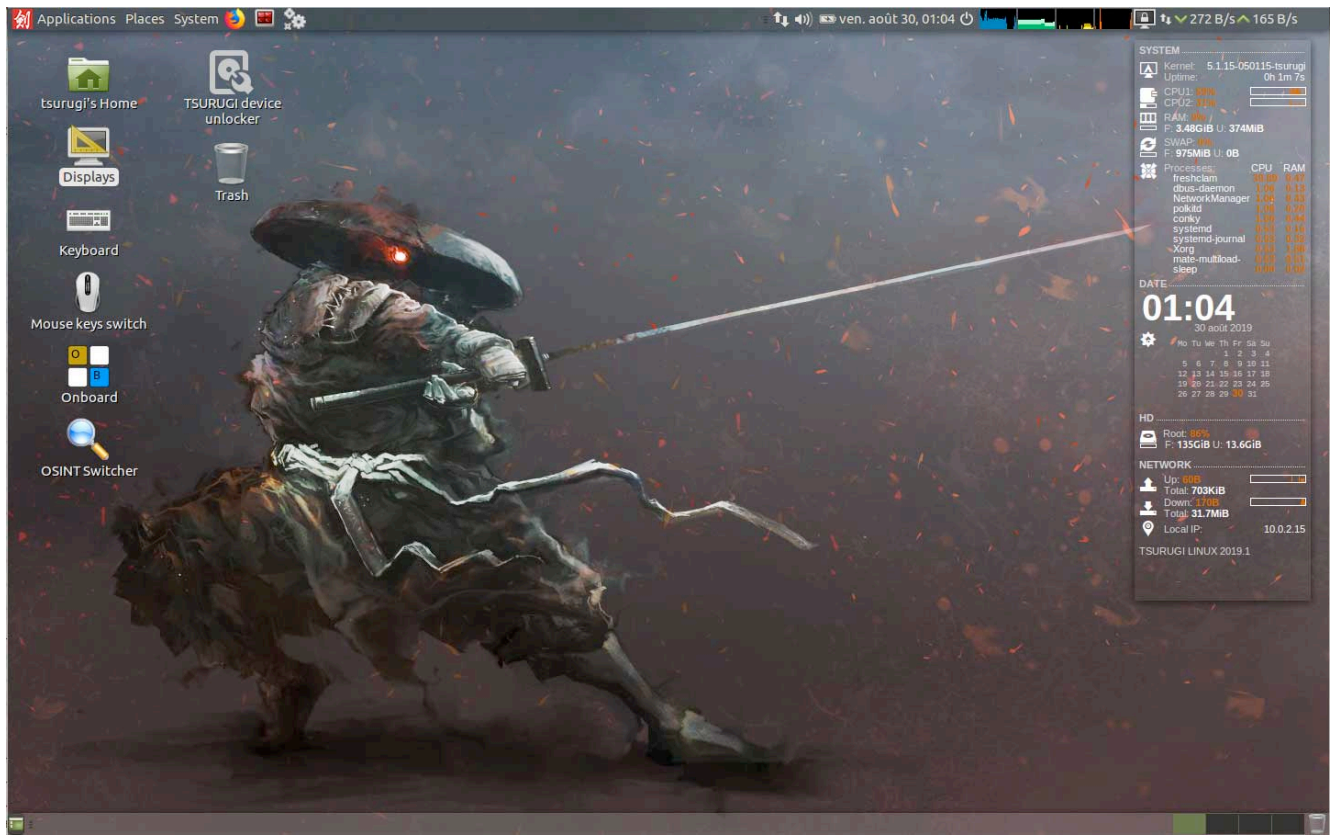
DFIR MENU

Shown below is the default DFIR appearance where the OSINT tools are still present but are drowned out by many other menu categories and so less practical to use. For this reason, a dedicated OSINT profile has been added and a new feature is available to easily switch.



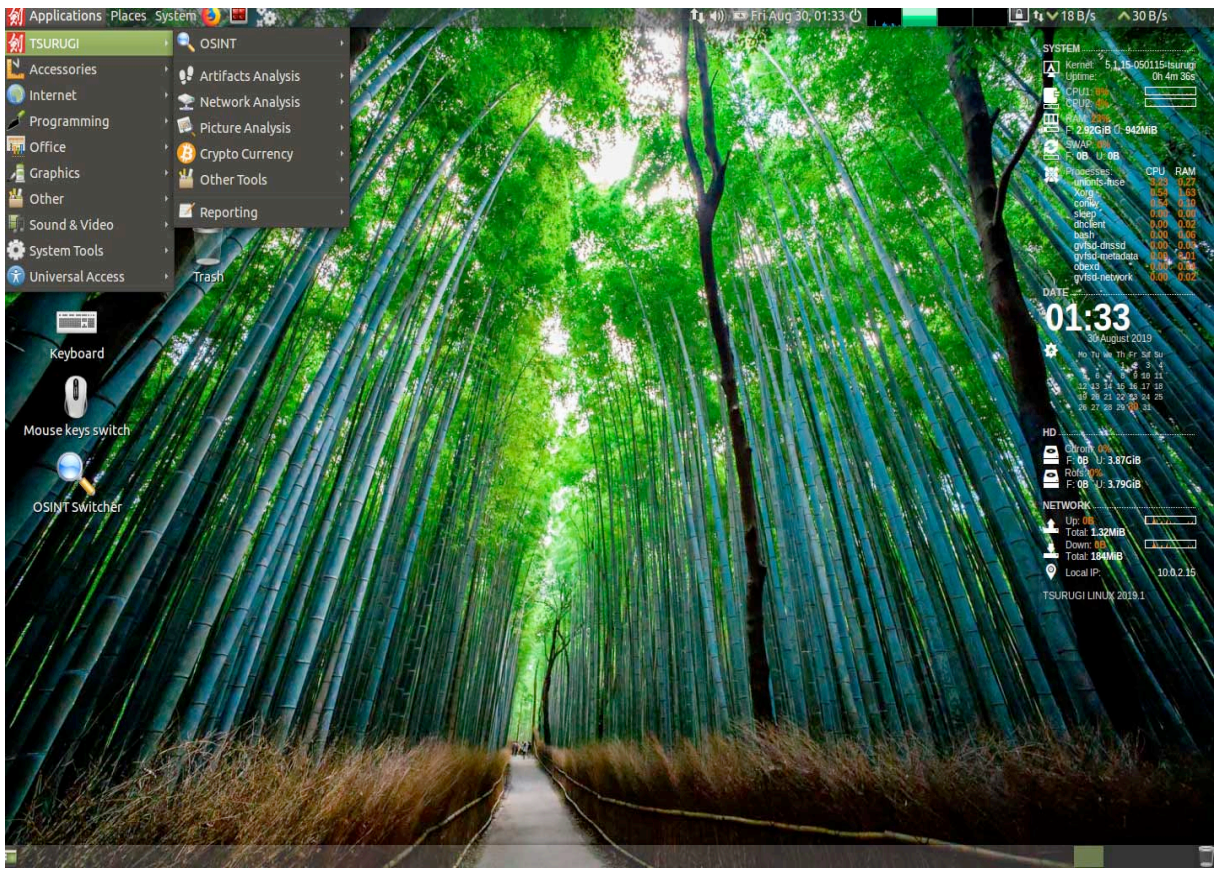
OSINT profile switcher button

On the desktop (*and also inside Tsurugi Linux menus*) there is the "***OSINT profile switcher***" button that allows users to modify the default appearance to switch to the OSINT profile environment.



OSINT profile

After a simple click, the Desktop appearance changes and the Tsurugi menus show only useful OSINT tools categories.



OSINT tools

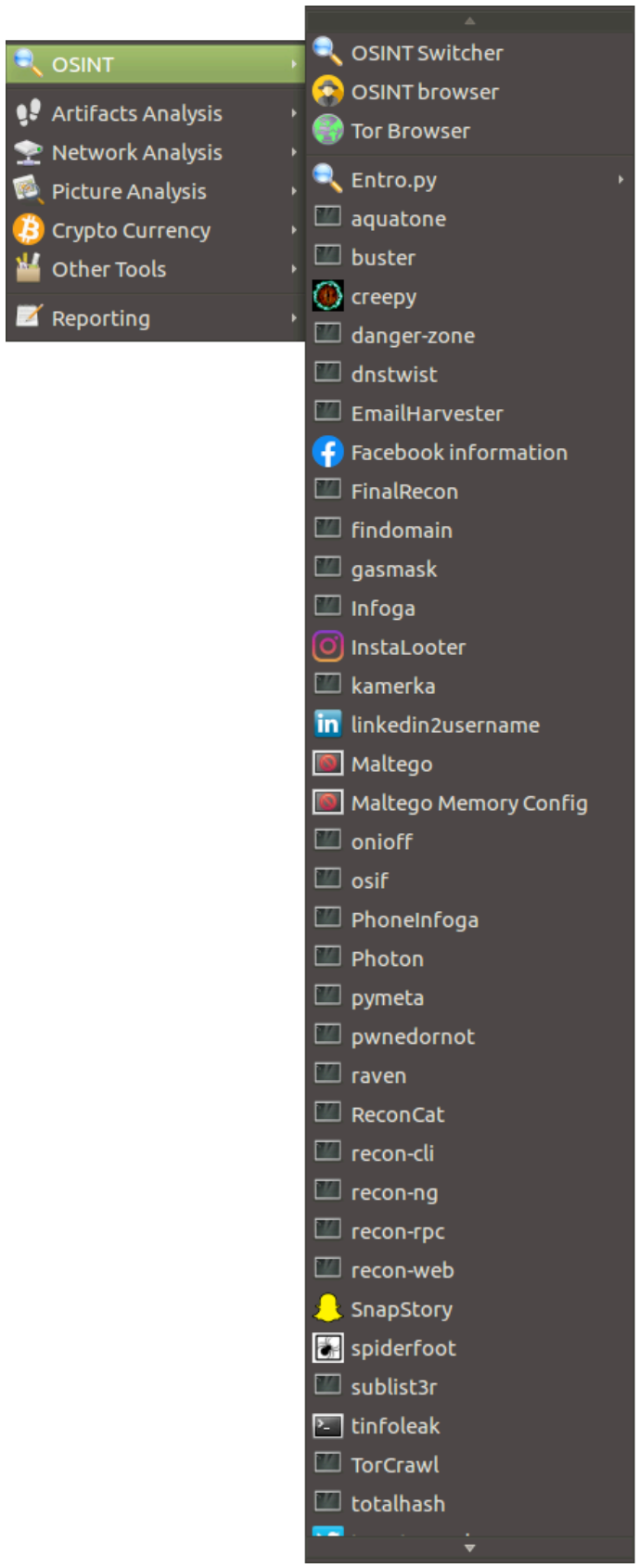
The main goal of Tsurugi Linux is DFIR analysis but many OSINT tools have also been installed to perform investigations on social networks, IP addresses, domains and many other fields.

In the menu it's possible find a customized OSINT browser with a specific profile with many useful plugins and bookmarks sorted under several main categories like:

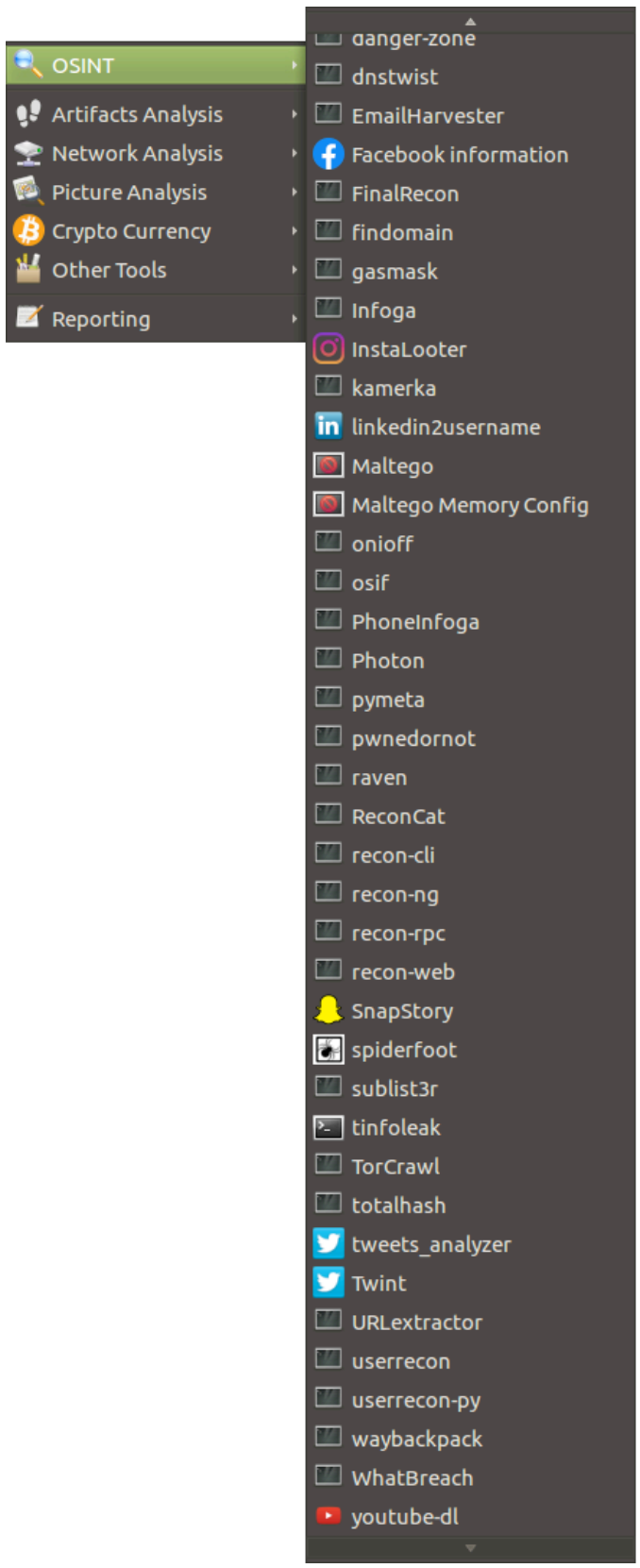
- Transports (SEA, AIR, RAILROADS…)
- Socials
- Metadata
- Date-time
- Website analysis

- Geo based researches
- Radio
- Search engines (with specific IOT categories)
- Commercial registries
- and many more…

Below is a list of OSINT tools (PART 1):



OSINT TOOLS (PART 2):



Graphical dashboard

A graphical dashboard is available on the desktop with real time information (if needed, it's possible to reset it by “Dashboard reset” button or “dashboard” command on Command Line Interface).

Mouse keys switch

A Mouse keys switch function has been added to easily move the mouse pointer with only the keypad if needed. The on/off button is available on the Desktop.

Automatic SSH keys generation

To avoid sharing the same SSH secret keys in all ISO files, an automatic process during the boot of the live session builds secure SSH keys.

RAM saturation workaround

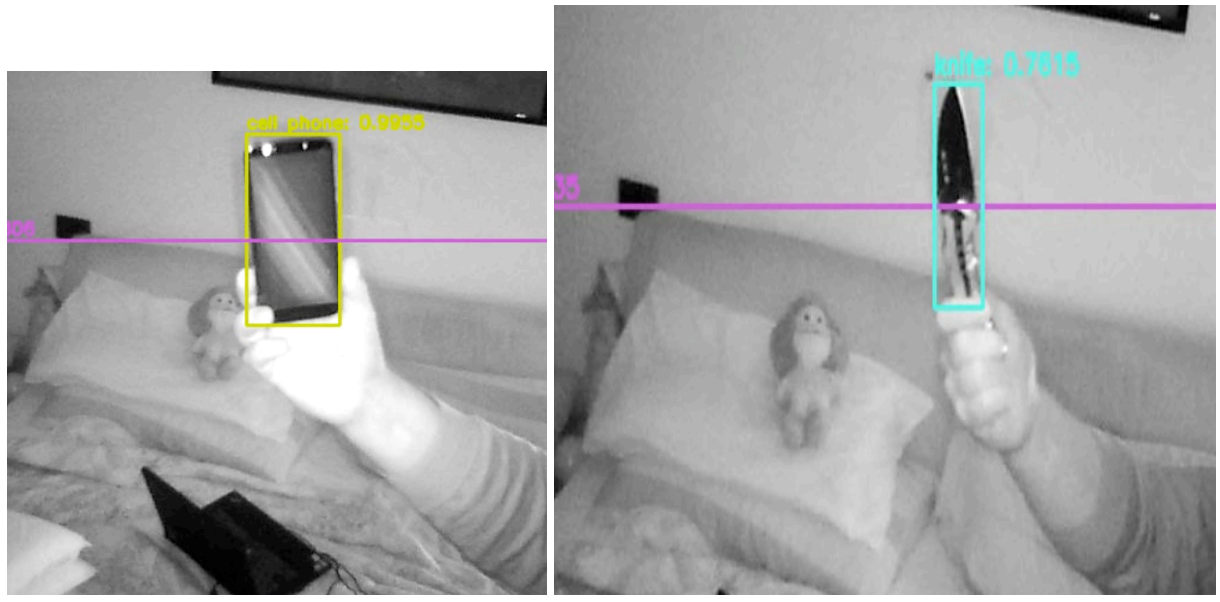
In some particular cases, it's possible to find faulty or incompatible hardware that generates a huge quantity of error logs and so that, in live mode, saturates RAM. For this specific situation, a custom logrotate configuration has been added, available under "RAM saturation workaround" button, that allows users to work on this system despite this hardware problem.

Computer Vision

Since 2019, a special section has also been added dedicated to Computer Vision investigations with many custom tools where it's possible to perform automatic face recognition tasks, compare pictures, recognize a person or an object inside photos or videos (live or registered).

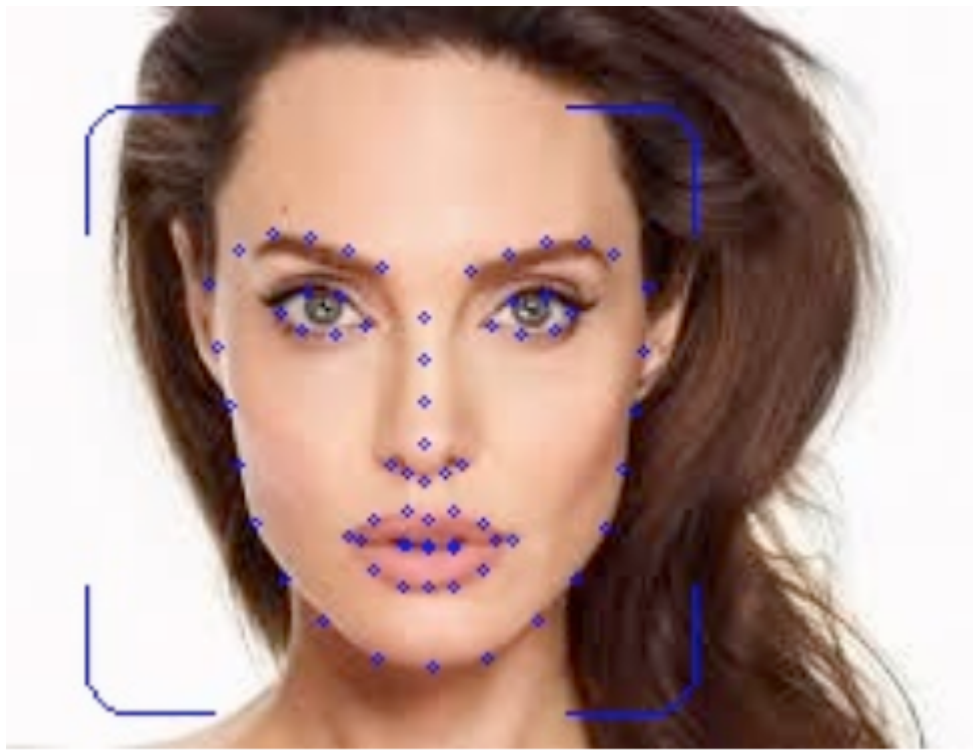
Object recognition

With object recognition tools implemented in Tsurugi Linux, it's possible to automatically detect and find specific objects (persons, cars, chairs…) in pictures, video or live streaming.



Face landmarks

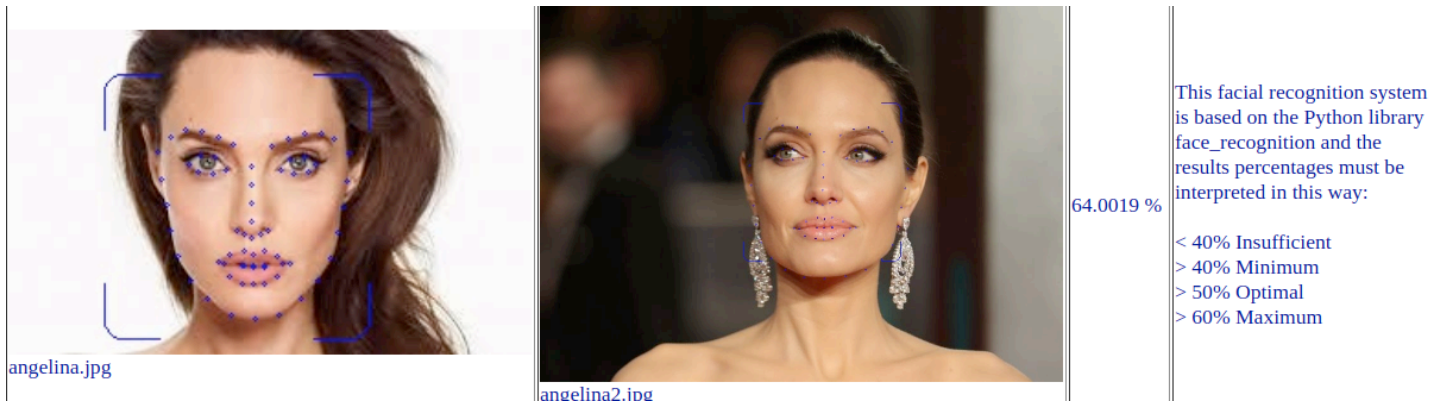
With face recognition tools it's possible to automatically put "face landmarks" on pictures to calculate face measurement and so be able to find a specific person inside pictures, video or live streaming.



Face recognition match

Based on face landmarks and several other algorithms, it's possible to search for a specific person inside many pictures and show the exact match with the similarity percentage.

Below is a part of the HTML report automatically generated for all the matches.



TSURUGI ACQUIRE

Tsurugi Acquire is a lightweight and streamlined version of Tsurugi Linux [LAB], aimed at providing the basic tools needed to boot a PC and acquire mass storage devices. For this reason, the installer has been deleted and **it runs only in live mode**.

A small subset of tools is installed to make the whole ISO smaller and its main purposes are to easily reside in RAM, be fast at boot and support as many architectures as possible.

Based on Ubuntu 16.04 LTS with a patched kernel 5.4.2 as well as Tsurugi Linux main distro, it has instead a 32 bit kernel to grant more compatibility and easily run on the oldest devices.

Below is the main desktop of Tsurugi Acquire with the smallest menu with only specific acquisition tools.

BENTO Toolkit

Bento is a portable USB DFIR toolkit designed for live forensics and incident response investigations.

Made to support CSI during search and seizure activities Bento DFIR toolkit provides first responders with a complete and easy way to face the most common activities like identification, information gathering, acquisition, seizure and preservation of digital evidence.

Bento isn't intended for forensic analysis other than the strictly necessary investigations in live mode and for triaging.

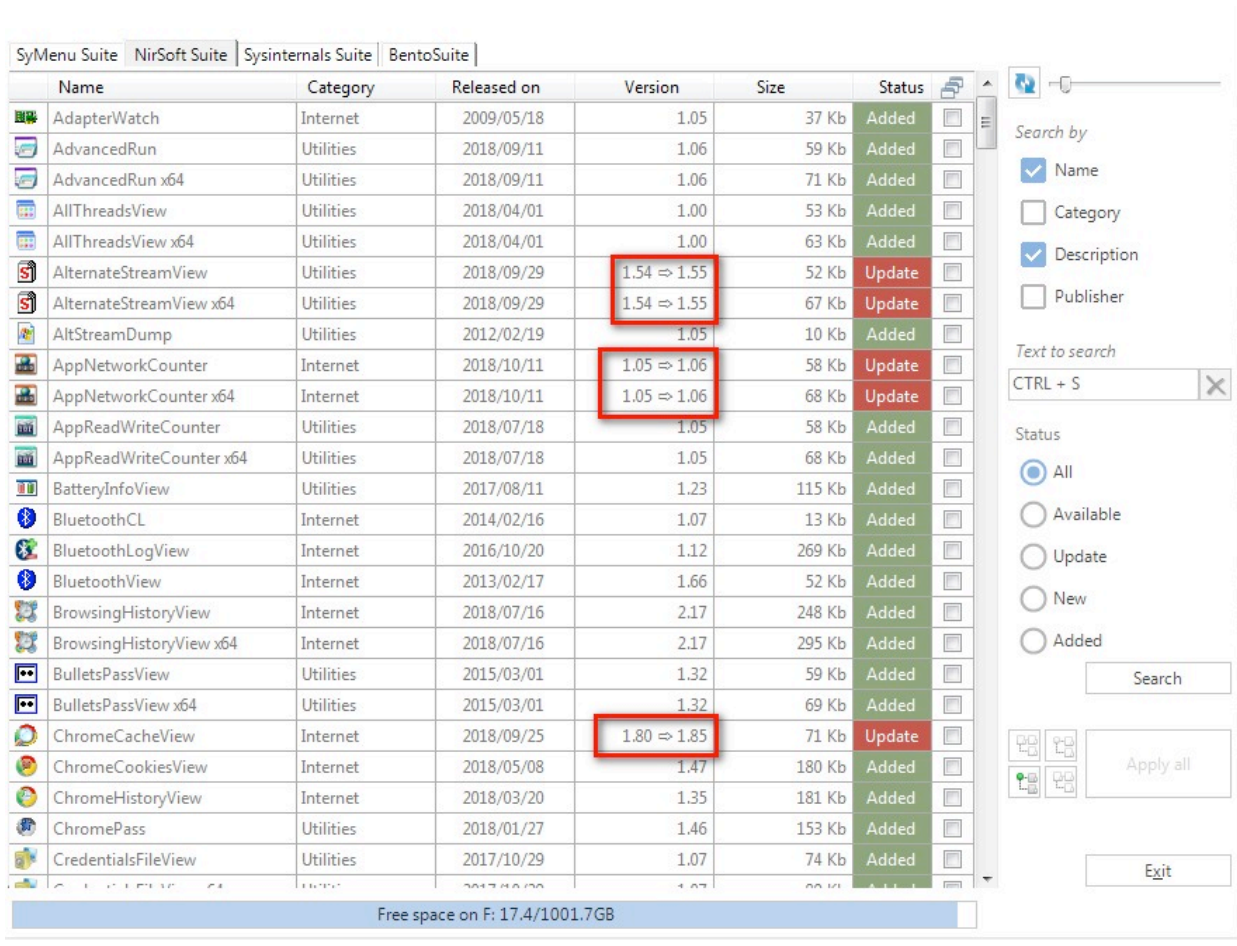
Bento works mainly on Windows operating systems but there are a lot of tools also for Linux and macOS.

Automatic updates

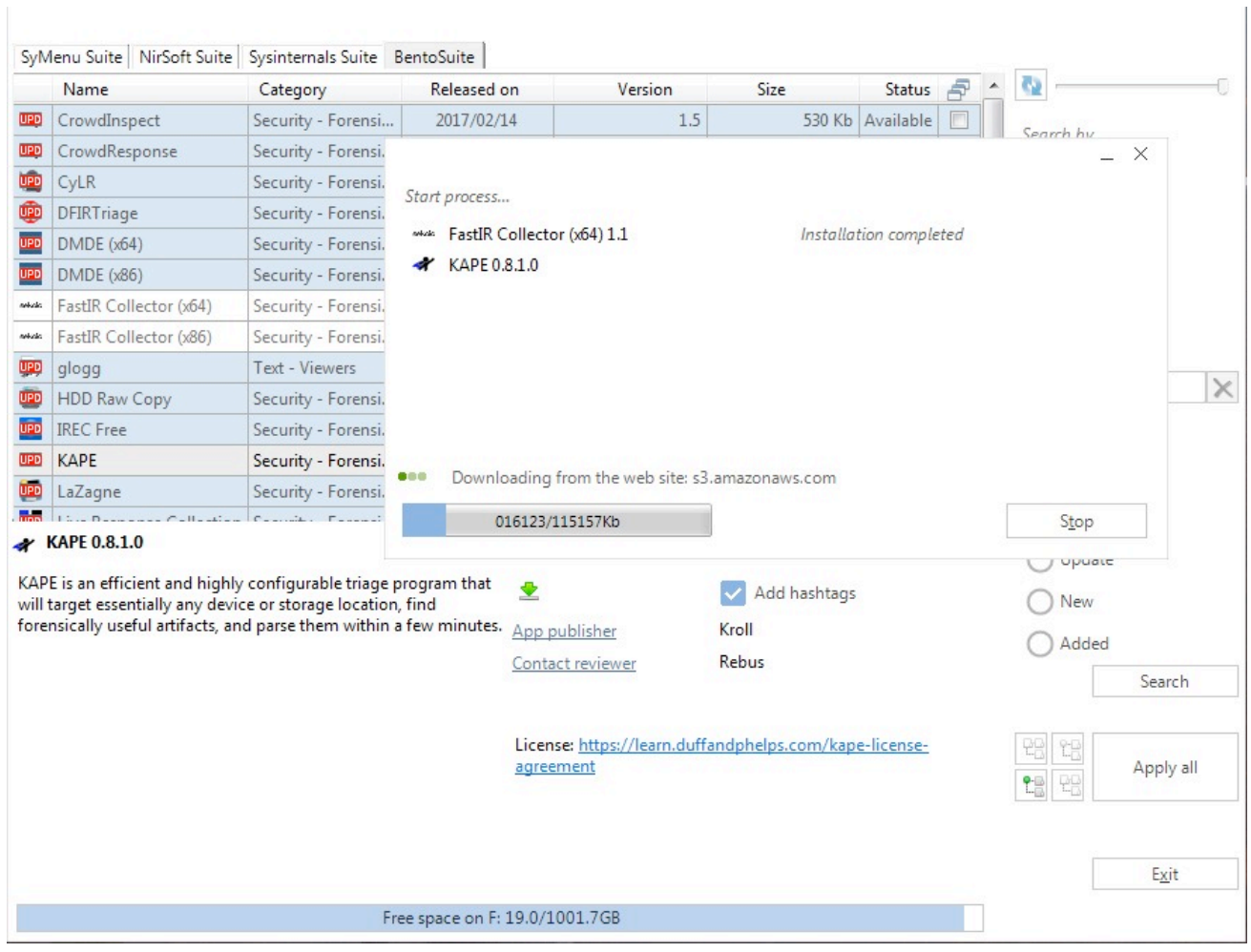
It’s possible to easily retrieve the latest tool updates just by launching the main interface.

All the available updates are shown below so it’s easy to validate and start the tools update.

New tools installation:



It's also possible to add new tools in Bento just using the specific menu:



Tsurugi Linux additional information and contacts

Tsurugi Linux is an open source project that is and will be totally free and independent without involving any commercial brand because our main goal is to share knowledge and "***give back to the community***".

Here is information about the project and how to communicate with the team:

Official website: https://tsurugi-linux.org

Twitter account: @tsurugi_linux

Email: info@tsurugi-linux.org

Download page: https://tsurugi-linux.org/downloads.php

About the Author

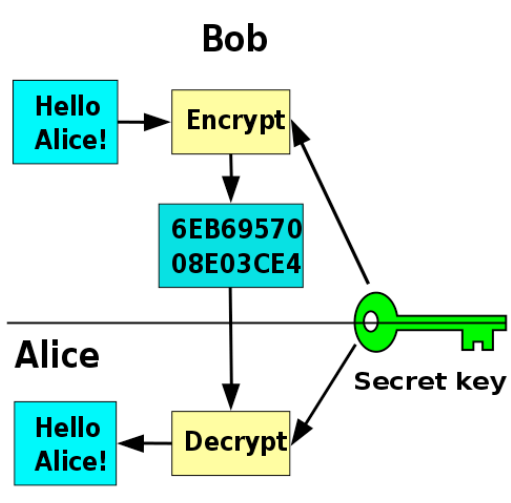
Giovanni Rattaro is a senior cyber security expert and an international public speaker based in Paris, old italian Backtrack Linux ambassador and ex DEFT Linux developer, now is the Tsurugi Linux core developer and project team leader (https://tsurugi-linux.org). DFIR instructor in his free time and passionate of many other topics like cyber-threat intelligence investigations, OSINT activities and interpersonal communication with a particular focus on "non verbal" (body language).

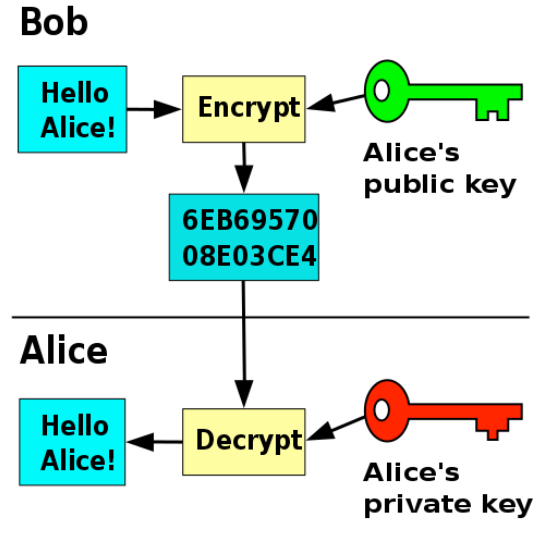
CryKeX – Linux Memory Cryptographic Keys Extractor

by Maksym Zaitsev

Nowadays, cryptography is almost everywhere, but not so long ago it was considered a weapon in some countries. Historically, there were some limitations for the key length and encryption algorithms, but now, due to the Kerckhoffs's principle, you can use almost any type of cipher you would like, keeping only the key in secret. Those keys, however, should be well protected, which, unfortunately, isn't the case for most modern software and this article will show how to obtain those keys without much effort.

So, let's warm up a little bit with cipher operations. Basically, a symmetric cipher (AES, RC4, etc.) requires only one secret key to encrypt and decrypt the data, whereas an asymmetric cipher (RSA, ECC, etc.) requires two separate keys (public and private) for both actions:





Given that we know how both algorithms work, in order to decrypt the data, we just have to get the keys. Although there are still some possibilities for cryptanalysis, modern ciphers are pretty resistant to many known attacks and breaking them isn't a trivial task, as numerous studies show.

Hence, every time an encryption or decryption operation is performed, a key has to be used and thus, exposed. Even if you don't use a text file on a hard disk to store your keys, they will always be stored in the RAM (volatile memory). Of course, after your PC is turned off, those keys will be lost and couldn't be easily restored, though such an attack can be done in practice if certain conditions are met. However, if you have physical access to a PC, you can extract them in a different and easier way.

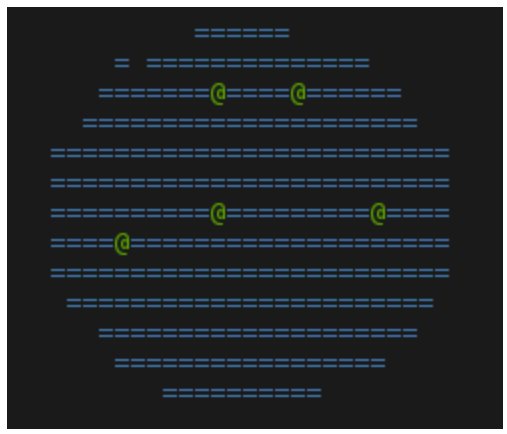
Before we proceed to the actual topic, let's refresh our memory a little bit on how software works.

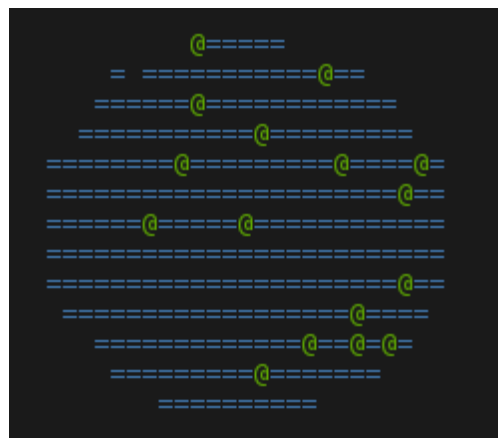
Every program launched will have its own virtual memory space, which will be divided in sections. The most important of them are the assembly code itself, system libraries needed for execution, the heap (used for objects, structures) and the stack (variables, pointers). Encryption keys are usually either dynamically allocated structures (heap) and/or just values stored in variables (stack). So, if we dump the memory of a process during the execution, we can try to find something that resembles a cryptographic key.

Assuming that we know the length (128 bits), we also know that AES encryption doesn't just take the user's key and encrypt the data with it, AES will do what's called key scheduling, basically making it look more random and secure. The "randomness" of data can be measured mathematically using entropy.

If the user supplies "mykey" as encryption key for AES, the real scheduled key will be 9adbe0b3033881f88ebd825bcf763b43.

The entropy of "mykey" is less then 2 random bits per byte (2/8), whereas the real key has the entropy of 4/8. We can make a graphical representation using the standard deviation proportional to the center of a circle:





OK, enough theory, let's get to practice by trying to debug an OpenSSL encryption process and see what it does exactly to the key and how we can extract it.

First let's launch the encryption process with "mykey" and "mytext" to encrypt in a file named testAES.enc with AES-128-ECB:

```
$ openssl aes-128-ecb -nosalt -out testAES.enc
```

```
$ openssl aes-128-ecb -nosalt -out testAES.enc
enter aes-128-ecb encryption password:
Verifying - enter aes-128-ecb encryption password:
mytext
```

We will enter our key twice and then some data, but just before finishing, we will debug the process using GDB with pwngdb extension:

```
$ gdb -p $(pidof openssl)
```

Then ask to make a break point in the end of the last encryption function and continue the execution by validating our input:

```
gdb> break *EVP_EncryptFinal_ex+126
gdb> continue
```

At this point, RBX register (as well as current stack location) contains a pointer to our structure and it has been already populated with the scheduled key at a 120 bytes offset:

```
gdb> x/2a *($rbx+120)
```

```
*RBX  0x235ec38 —▸ 0x7f9f4ba9fa00 ◂— 0x10000001a2
*RCX  0xf
*RDX  0x235ec20 ◂— 0x0
*RDI  0x9
*RSI  0x235ece0 ◂— 0x0
*R8   0x7f9f4b4554e0 (_IO_2_1_stdin_) ◂— 0xfbad2298
*R9   0x7f9f4bf02700 ◂— 0x7f9f4bf02700
*R10  0x7ffd9be5f840 ◂— 0x0
*R11  0x7f9f4b793dd0 (EVP_EncryptFinal_ex) ◂— push   r13
*R12  0x235ec20 ◂— 0x0
*R13  0x235ece0 ◂— 0x0
*R14  0x235ec38 —▸ 0x7f9f4ba9fa00 ◂— 0x10000001a2
*R15  0x235ece0 ◂— 0x0
*RBP  0x10
*RSP  0x7ffd9be5fa50 —▸ 0x235ec38 —▸ 0x7f9f4ba9fa00 ◂— 0x10000001a2
*RIP  0x7f9f4b793e4e (EVP_EncryptFinal_ex+126) ◂— mov    rax, qword ptr [rbx]
[ DISASM ]
 ► 0x7f9f4b793e4e <EVP_EncryptFinal_ex+126>    mov    rax, qword ptr [rbx]
   0x7f9f4b793e51 <EVP_EncryptFinal_ex+129>    mov    ecx, ebp
   0x7f9f4b793e53 <EVP_EncryptFinal_ex+131>    lea    rdx, [rbx + 0x38]
   0x7f9f4b793e57 <EVP_EncryptFinal_ex+135>    mov    rsi, r13
   0x7f9f4b793e5a <EVP_EncryptFinal_ex+138>    mov    rdi, rbx
   0x7f9f4b793e5d <EVP_EncryptFinal_ex+141>    call   qword ptr [rax + 0x20]

   0x7f9f4b793e60 <EVP_EncryptFinal_ex+144>    test   eax, eax
   0x7f9f4b793e62 <EVP_EncryptFinal_ex+146>    je     EVP_EncryptFinal_ex+79 <0x7f9f4
b793e1f>

   0x7f9f4b793e64 <EVP_EncryptFinal_ex+148>    mov    dword ptr [r12], ebp
   0x7f9f4b793e68 <EVP_EncryptFinal_ex+152>    add    rsp, 8
   0x7f9f4b793e6c <EVP_EncryptFinal_ex+156>    pop    rbx
[ STACK ]
00:0000| rsp  0x7ffd9be5fa50 —▸ 0x235ec38 —▸ 0x7f9f4ba9fa00 ◂— 0x10000001a2
01:0008|      0x7ffd9be5fa58 —▸ 0x235ec20 ◂— 0x0
02:0010|      0x7ffd9be5fa60 —▸ 0x235eba0 —▸ 0x7f9f4bab7480 ◂— 0x20a
03:0018|      0x7ffd9be5fa68 ◂— 0x0
... ↓
05:0028|      0x7ffd9be5fa78 —▸ 0x7f9f4b79d2d2 ◂— movsxd rdx, eax
06:0030|      0x7ffd9be5fa80 —▸ 0x235ba10 ◂— 0xa74786574796d /* u'mytext\n' */
... ↓
[ BACKTRACE ]
 ► f 0     7f9f4b793e4e EVP_EncryptFinal_ex+126
   f 1     7f9f4b79d2d2
   f 2           424141
   f 3           41a758
   f 4           41a413
   f 5     7f9f4b0d0b45 __libc_start_main+245
Breakpoint *EVP_EncryptFinal_ex+126
pwndbg> x/2a *($rbx+120)
0x235fd30:      0xf8813803b3e0db9a      0x433b76cf5b82bd8e
```

This will give us the true encryption key in Little-Endian (04030201 instead of 01020304).

Let's advance a bit just before the encryption itself:

```
gdb> step 32
```

```
[ REGISTERS ]
*RAX  0x9
 RBX  0x235ec38 —▸ 0x7f9f4ba9fa00 ◂— 0x10000001a2
*RCX  0x235fd30 ◂— 0xf8813803b3e0db9a
*RDX  0x10
*RDI  0x235ec70 ◂— 0x90a74786574796d ('mytext\n\t')
 RSI  0x235ece0 ◂— 0x0
*R8   0x1
*R9   0x10
*R10  0x9
*R11  0x235fd30 ◂— 0xf8813803b3e0db9a
 R12  0x235ec20 ◂— 0x0
 R13  0x235ece0 ◂— 0x0
 R14  0x235ec38 —▸ 0x7f9f4ba9fa00 ◂— 0x10000001a2
 R15  0x235ece0 ◂— 0x0
 RBP  0x10
*RSP  0x7ffd9be5fa38 —▸ 0x7f9f4b798fb5 ◂— mov    eax, 1
*RIP  0x7f9f4b7109f3 ◂— movups xmm1, xmmword ptr [rcx + 0x10]
[ DISASM ]
► 0x7f9f4b7109f3    movups xmm1, xmmword ptr [rcx + 0x10]
  0x7f9f4b7109f7    lea    rcx, [rcx + 0x20]
  0x7f9f4b7109fb    xorps  xmm2, xmm0
  0x7f9f4b7109fe    aesenc xmm2, xmm1
  0x7f9f4b710a03    dec    eax
  0x7f9f4b710a05    movups xmm1, xmmword ptr [rcx]
  0x7f9f4b710a08    lea    rcx, [rcx + 0x10]
  0x7f9f4b710a0c    jne    0x7f9f4b7109fe
   ↓
  0x7f9f4b7109fe    aesenc xmm2, xmm1
  0x7f9f4b710a03    dec    eax
  0x7f9f4b710a05    movups xmm1, xmmword ptr [rcx]
[ STACK ]
00:0000| rsp  0x7ffd9be5fa38 —▸ 0x7f9f4b798fb5 ◂— mov    eax, 1
01:0008|      0x7ffd9be5fa40 —▸ 0x235ec20 ◂— 0x0
02:0010|      0x7ffd9be5fa48 —▸ 0x7f9f4b793e60 (EVP_EncryptFinal_ex+144) ◂—
x, eax
03:0018|      0x7ffd9be5fa50 —▸ 0x235ec38 —▸ 0x7f9f4ba9fa00 ◂— 0x10000001a2
04:0020|      0x7ffd9be5fa58 —▸ 0x235ec20 ◂— 0x0
05:0028|      0x7ffd9be5fa60 —▸ 0x235eba0 —▸ 0x7f9f4bab7480 ◂— 0x20a
06:0030|      0x7ffd9be5fa68 ◂— 0x0
... ↓
[ BACKTRACE ]
► f 0     7f9f4b7109f3
  f 1     7f9f4b798fb5
  f 2     7f9f4b793e60 EVP_EncryptFinal_ex+144
  f 3     7f9f4b79d2d2
  f 4           424141
  f 5           41a758
  f 6           41a413
  f 7     7f9f4b0d0b45 __libc_start_main+245
```

Here, we are about to compute the first round of AES, the key is stored in XMM0 and the padded text to encrypt is stored in XMM2.

```
gdb> info registers xmm0

gdb> info registers xmm2
```

```
pwndbg> info registers xmm0
xmm0           {
  v4_float = {0x0, 0x0, 0x0, 0xbb},
  v2_double = {0x8000000000000000, 0x1b76cf5b82bd8e},
  v16_int8 = {0x9a, 0xdb, 0xe0, 0xb3, 0x3, 0x38, 0x81, 0xf8, 0x8e, 0xbd, 0x82, 0x5b,
0xcf, 0x76, 0x3b...},
  v8_int16 = {0xdb9a, 0xb3e0, 0x3803, 0xf881, 0xbd8e, 0x5b82, 0x76cf, 0x433b},
  v4_int32 = {0xb3e0db9a, 0xf8813803, 0x5b82bd8e, 0x433b76cf},
  v2_int64 = {0xf8813803b3e0db9a, 0x433b76cf5b82bd8e},
  uint128 = 0x433b76cf5b82bd8ef8813803b3e0db9a
}
pwndbg> info registers xmm2
xmm2           {
  v4_float = {0x0, 0x0, 0x0, 0x0},
  v2_double = {0x0, 0x0},
  v16_int8 = {0x6d, 0x79, 0x74, 0x65, 0x78, 0x74, 0xa, 0x9, 0x9, 0x9, 0x9, 0x9, 0x9,
0x9, 0x9...},
  v8_int16 = {0x796d, 0x6574, 0x7478, 0x90a, 0x909, 0x909, 0x909, 0x909},
  v4_int32 = {0x6574796d, 0x90a7478, 0x9090909, 0x9090909},
  v2_int64 = {0x90a74786574796d, 0x909090909090909},
  uint128 = 0x0909090909090909090a74786574796d
}
pwndbg>
```

You might notice that XMM registers are 128 bits, so I'm sorry to say, but the fact that your system is 64 bits is actually a lie. Modern CPUs contain 128, 256 and even 512 bits registers that were useful historically to store graphical objects and data, but later were also used for encryption and so the AES-NI assembly instructions were hardcoded into the processor to make AES even faster.

Of course, these results can be verified using some online tools:

```
$ hd mykey
00000000  6d 79 6b 65 79                                    |mykey|
00000005
$ hd mykey.hex
00000000  9a db e0 b3 03 38 81 f8  8e bd 82 5b cf 76 3b 43  |.....8.....[.v;C|
00000010
$ hd testAES.enc
00000000  16 d0 6a 8c be 25 5b 73  d6 27 7d 19 28 77 4e 8a  |..j..%[s.'}.(wN.|
00000010
```

Input type: Text

Input text: (hex)

```
16 d0 6a 8c be 25 5b 73  d6 27 7d 19 28 77 4e 8a
```

○ Plaintext ◉ Hex

Function: AES

Mode: ECB (electronic codebook)

Key: (hex)

```
9a db e0 b3 03 38 81 f8  8e bd 82 5b cf 76 3b 43
```

○ Plaintext ◉ Hex

> Encrypt! > Decrypt!

Decrypted text:

```
00000000  6d 79 74 65 78 74 0a 09 09 09 09 09 09 09 09 09  |m y t e x t . .
```

OK, but what about the memory? Well, in the case of OpenSSL, the keys are stored as they are in stack and heap. Despite the fact that the memory mapping for some structures would be anonymous and memory addresses could change as well, we can still predict some offsets:

```
gdb> info proc map
```

or

```
gdb> vmmap
```

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
          0x400000           0x484000 r-xp    84000 0      /usr/bin/openssl
          0x684000           0x685000 r--p     1000 84000  /usr/bin/openssl
          0x685000           0x68b000 rw-p     6000 85000  /usr/bin/openssl
          0x68b000           0x68c000 rw-p     1000 0
         0x1aa2000          0x1ae5000 rw-p    43000 0      [heap]
    0x7fdfe1986000     0x7fdfe1b27000 r-xp   1a1000 0      /lib/x86_64-linux-gnu/libc-2.19.so
    0x7fdfe1b27000     0x7fdfe1d27000 ---p   200000 1a1000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7fdfe1d27000     0x7fdfe1d2b000 r--p     4000 1a1000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7fdfe1d2b000     0x7fdfe1d2d000 rw-p     2000 1a5000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7fdfe1d2d000     0x7fdfe1d31000 rw-p     4000 0
    0x7fdfe1d31000     0x7fdfe1d34000 r-xp     3000 0      /lib/x86_64-linux-gnu/libdl-2.19.so
    0x7fdfe1d34000     0x7fdfe1f33000 ---p   1ff000 3000   /lib/x86_64-linux-gnu/libdl-2.19.so
    0x7fdfe1f33000     0x7fdfe1f34000 r--p     1000 2000   /lib/x86_64-linux-gnu/libdl-2.19.so
    0x7fdfe1f34000     0x7fdfe1f35000 rw-p     1000 3000   /lib/x86_64-linux-gnu/libdl-2.19.so
    0x7fdfe1f35000     0x7fdfe2168000 r-xp   233000 0      /usr/lib/x86_64-linux-gnu/libcrypto.so.1.0.0
    0x7fdfe2168000     0x7fdfe2368000 ---p   200000 233000 /usr/lib/x86_64-linux-gnu/libcrypto.so.1.0.0
    0x7fdfe2368000     0x7fdfe2386000 r--p    1e000 233000 /usr/lib/x86_64-linux-gnu/libcrypto.so.1.0.0
    0x7fdfe2386000     0x7fdfe2396000 rw-p    10000 251000 /usr/lib/x86_64-linux-gnu/libcrypto.so.1.0.0
    0x7fdfe2396000     0x7fdfe239a000 rw-p     4000 0
    0x7fdfe239a000     0x7fdfe23fa000 r-xp    60000 0      /usr/lib/x86_64-linux-gnu/libssl.so.1.0.0
    0x7fdfe23fa000     0x7fdfe25fa000 ---p   200000 60000  /usr/lib/x86_64-linux-gnu/libssl.so.1.0.0
    0x7fdfe25fa000     0x7fdfe25fe000 r--p     4000 60000  /usr/lib/x86_64-linux-gnu/libssl.so.1.0.0
    0x7fdfe25fe000     0x7fdfe2604000 rw-p     6000 64000  /usr/lib/x86_64-linux-gnu/libssl.so.1.0.0
    0x7fdfe2604000     0x7fdfe2625000 r-xp    21000 0      /lib/x86_64-linux-gnu/ld-2.19.so
    0x7fdfe27d8000     0x7fdfe27dc000 rw-p     4000 0
    0x7fdfe2821000     0x7fdfe2824000 rw-p     3000 0
    0x7fdfe2824000     0x7fdfe2825000 r--p     1000 20000  /lib/x86_64-linux-gnu/ld-2.19.so
    0x7fdfe2825000     0x7fdfe2826000 rw-p     1000 21000  /lib/x86_64-linux-gnu/ld-2.19.so
    0x7fdfe2826000     0x7fdfe2827000 rw-p     1000 0
    0x7ffccd1f9000     0x7ffccd21a000 rw-p    21000 0      [stack]
    0x7ffccd32a000     0x7ffccd32c000 r-xp     2000 0      [vdso]
    0x7ffccd32c000     0x7ffccd32e000 r--p     2000 0      [vvar]
0xffffffffff600000 0xffffffffff601000 r-xp     1000 0      [vsyscall]
```

```
gdb> x/2a 0x1aa2000+0x23d30
```

```
pwndbg> x/2a 0x1aa2000+0x23d30
0x1ac5d30:      0xf8813803b3e0db9a      0x433b76cf5b82bd8e
```

As we can see, the key is always stored in heap region after 0x23d30 bytes from its beginning (for my OpenSSL version). If the key is stored in a file, this offset will be a bit different (0x23b40). Moreover, before storing the key, there will be zeroes and 0x111 value:

```
pwndbg> x/4a 0x1aa2000+0x23d30-0x20
0x1ac5d10:      0x0     0x0
0x1ac5d20:      0x0     0x111
```

The stack offset varies, same for storage prefixes and suffixes, but you can always find it:

```
gdb> find 0x7ffe335eb000, +0x21000, 0xf8813803b3e0db9a
```

```
    0x7ffe335eb000     0x7ffe3360c000 rw-p    21000 0      [stack]
    0x7ffe33716000     0x7ffe33718000 r-xp     2000 0      [vdso]
    0x7ffe33718000     0x7ffe3371a000 r--p     2000 0      [vvar]
0xffffffffff600000 0xffffffffff601000 r-xp     1000 0      [vsyscall]
pwndbg> find 0x7ffe335eb000, +0x21000, 0xf8813803b3e0db9a
0x7ffe33608fe0
1 pattern found.
```

Anyway, since the key is stored in clear, it's not a big deal to recover it, plus the key has a specific header and entropy, thus it can indeed be extracted just by dumping the memory. Some projects, however, like OpenVPN, despite relying on OpenSSL, only store keys in heap and will even store multiple locations for the same key (0x23ba0 and 0x24170). OpenSSH, for instance, has its own crypto implementation (since 2014), and doesn't store keys in stack as well, but still doesn't change its offsets for heap and just like OpenVPN, stores two values of the same key (0x217a0 and 0x24260, but their distance and actual values may vary as well).

Fortunately, the keys and the data are securely erased from the memory after usage:

```
[ DISASM ]
► 0x7f2df8fb11d0 <CRYPTO_free>       mov    rax, qword ptr [rip + 0x3e9691]
  0x7f2df8fb11d7 <CRYPTO_free+7>     push   rbx
  0x7f2df8fb11d8 <CRYPTO_free+8>     mov    rbx, rdi
  0x7f2df8fb11db <CRYPTO_free+11>    test   rax, rax
  0x7f2df8fb11de <CRYPTO_free+14>    je     CRYPTO_free+20 <0x7f2df8fb11e4>
   ↓
  0x7f2df8fb11e4 <CRYPTO_free+20>    mov    rdi, rbx
  0x7f2df8fb11e7 <CRYPTO_free+23>    call   qword ptr [rip + 0x3ddfb3] <0x7f2df8a076
00>

  0x7f2df8fb11ed <CRYPTO_free+29>    mov    rax, qword ptr [rip + 0x3e9674]
  0x7f2df8fb11f4 <CRYPTO_free+36>    test   rax, rax
  0x7f2df8fb11f7 <CRYPTO_free+39>    je     CRYPTO_free+56 <0x7f2df8fb1208>

  0x7f2df8fb11f9 <CRYPTO_free+41>    pop    rbx
[ STACK ]
00:0000| rsp  0x7ffc7500c328 —▸ 0x423c20 ◂— mov    rdi, qword ptr [rsp + 0x10]
01:0008|      0x7ffc7500c330 ◂— 0x1
02:0010|      0x7ffc7500c338 —▸ 0x1fa0ba0 —▸ 0x7f2df9393480 ◂— 0x20a
03:0018|      0x7ffc7500c340 —▸ 0x1f9da10 ◂— 0xa74786574796d /* u'mytext\n' */
04:0020|      0x7ffc7500c348 —▸ 0x1fa0670 —▸ 0x7f2df9392800 ◂— 0x402
05:0028|      0x7ffc7500c350 —▸ 0x1fa0ba0 —▸ 0x7f2df9393480 ◂— 0x20a
06:0030|      0x7ffc7500c358 ◂— 0x0
07:0038|      0x7ffc7500c360 —▸ 0x1f9d800 ◂— 0x0
```

Nonetheless, relying on offsets and/or suffixes/prefixes based on software isn't that great, a better way to find keys is directly using statistics. More than 10 years ago, there already were such studies and some effective methods were found and implemented.

The basic idea isn't that complex, search for data that has key length, verify its entropy, search for potential candidates to be the input for the scheduling nearby, if found, then it's highly likely to be a true encryption key:

```
$ aeskeyfind core.5658
9adbe0b3033881f88ebd825bcf763b43
Keyfind progress: 100%
```

This is a great project that works well, nevertheless, I improved its functionality by adding a possibility to extract cryptographic keys from live processes and arbitrary binaries, hence the name, CryKeX. I just dump the memory of a specified process then search for keys, plus you can also launch a binary and my tool will automatically dump the memory afterwards.

Now, let's see it in action:

```
$ ../CryKeX.sh openssl


 ######  ########  ##    ## ##    ## ######## ##     ##
##    ## ##     ##  ##  ##  ##   ##  ##        ##   ##
##       ##     ##   ####   ## ##   ##         ## ##
##       ########     ##    #####    ######      ###
##       ##   ##      ##    ## ##   ##         ## ##
##    ## ##    ##     ##    ##   ##  ##        ##   ##
 ######  ##     ##    ##    ##    ## ######## ##     ##

 ad8888888888ba
dP'          `"8b,
8  ,aaa,       "Y888a     ,aaaa,     ,aaa,  ,aa,
8  8' `8          "8baaaad""""""baaaad""""""baad""8b
8  8   8             """""""      """""""      """    8b
8  8, ,8         ,aaaaaaaaaaaaaaaaaaaaaaaaddddd88P
8  `""""'       ,d8""
Yb,         ,ad8"
 "Y8888888888P"

    Linux Memory Cryptographic Keys Extractor


DUMPING MEMORY ...

SEARCHING KEYS ...
```

```
*** AES-128-ECB ***
9adbe0b3033881f88ebd825bcf763b43
Keyfind progress: 100%
```

Such can be applied to nearly everything that uses encryption (browsers, containers, etc.). Of course, it also applies for forensic memory dumps. In reality, this is quite an offensive tool, for instance, it can extract cryptographic keys from ransomware.

Searching for asymmetric keys (like RSA) is more challenging since, there are some parameters to take into account (exponent, modulus, etc.), but it's still possible and has been done by the same team.

However, there is no such thing as perfect, for instance, some browsers, like Firefox, use OpenSSLv3 with a bit different key storage mechanics, PGP/GPG and LUKS will even obfuscate key storage in memory in order to make its recovery harder, but not impossible.

So, if the key is stored in memory and can be accessed, then where should we store it? Well, there are some solutions, like store it in CPU registers or in a secure chipset or even in a dedicated hardware. None of them is perfect, of course, but ideally, physical access to the hardware should be restricted, but then, if you can ensure physical security of your PC, why would you need encryption in the first place? I mean, it's impossible to guarantee physical security since we are all objects ourselves in the physical layer with its laws that we obey and can't control, but we still can achieve 99.9(9)% of security IF done right. The problem is that things are not done right all the time, but I hope that you have learned something from this article and will do things maybe not perfect, but at least better than they are right now. Remember, crypto is strong, but can be used weakly.

Thanks for reading.

@cryptolok

About the Author

I'm known internationally for various cybersecurity research and hacks, mentioned in dozens of articles, released tools with thousands lines of code in different languages including assembly, spoke at conferences worldwide and published papers.

Toss a coin to your... Toolkit

by Denis O'Brien

Have you ever analysed a document and wondered what that binary blob means, or maybe wanted to deobfuscate embedded data without pulling out your hair, or perhaps just to determine how risky it will be to open that document on a computer? This article introduces you to an online service that does all of that and more.

Introduction

IRIS-H is an online tool that helps malware analysts examine the components of files stored in directory-based or strictly structured formats, such as OLE-CF, OOXML, RTF, LNK and PDF. The purpose of the tool is to extract file components' metadata and enrich it with a human readable description. IRIS-H is also fitted with rule-based code logic that attempts to identify the risk level associated with opening the analysed file on a computer system.

What's under the hood?

The project to build the tool started as a hobby to learn server/client side JavaScript, so the tool is written using NodeJS and Angular with the exception of some AWS Lambda functions that enable data pre/post-processing, for example, rendering document page preview or decrypting a document with a provided password. The tool is hosted with AWS Beanstalk and stores data in AWS S3 and DynamoDB services.

It's worth mentioning that the IRIS-H user interface is powered by Nebular UI developed by the Akveo team. It's a pretty easy way of building a web user interface using Angular 8 and TrueScript.

The IRIS-H tool is available to anyone to use, but its code is not open source. One of the reasons for that is its heavy integration with before mentioned AWS services.

How can it help me?

Assuming you are in a malware analysis/digital forensics space, IRIS-H can help you perform static analysis of files stored in Microsoft Office, Portable Document and Shell Link Binary formats. Anyone familiar with static analysis knows that it has advantages and shortcomings. IRIS-H wins on analysis speed, but the document processing speed comes with limited ability to extract network/runtime based indicators of compromise. The tool was never designed to be another malware analysis sandbox though.

The tool was created with the idea of having a means of correlating analysed files based on their content and components' metadata. Where the original idea is still in development, the results of moving toward it can already be seen in the analysis report produced by the tool. These are just some of the features offered by IRIS-H:

- File Structure View with ability to preview and download individual file components/streams
- File Threat Level assessment with the breakdown of findings the assessment was based on
- View of Core, Application and Custom document properties where available
- Preview of the images contained in the analysed file
- Preview of executable content, such as VBA or JavaScript
- VBA Pseudo-code preview where available
- Rendered document page preview
- Textual content extracted from the analysed document
- Detailed description of file's components

- Preview of certificate used to sign the document
- Document embedded content extraction and preview

All the features above produce the data that is stored in a JSON structure and fed into IRIS-H user interface where it's transformed into a human readable report. Examples of a report can be found here.

The tool is also capable of creating a second JSON structure that can be fed into a SIEM and used as a means of correlating analysed files. Due to current hosting infrastructure resources constraints, this feature is disabled though.

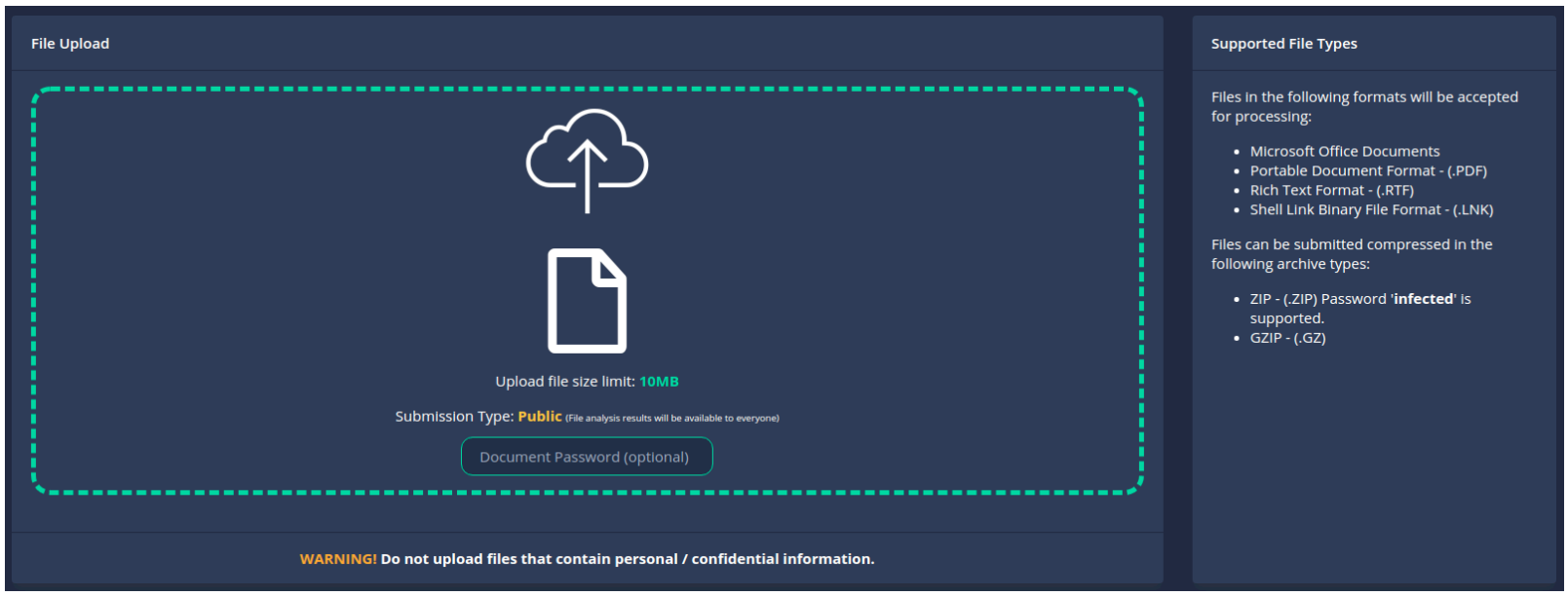
All-in-all, current IRIS-H functionality can be used to perform static analysis of files in the supported formats.

How to use the service

Using IRIS-H is as simple as browsing a website. Open the browser of your choice and navigate to https://iris-h.services.

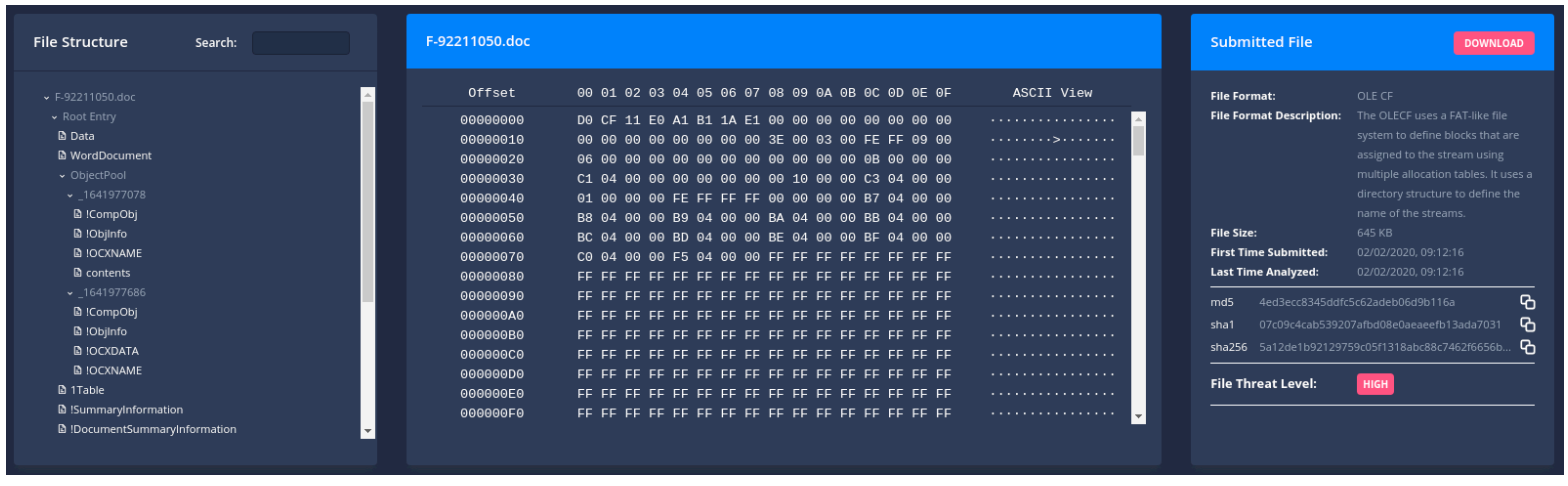
The landing page includes the overall statistics of service usage and a table listing the last 50 submissions with their corresponding analysis report links. The input fields at the top of each column enable table content filtering. Simply typing a keyword will filter the content to matching table entries.

The sidebar menu on the left has a number of options with one being 'File Upload'. Clicking on it takes the user to a file submission page where a file can be uploaded for processing.

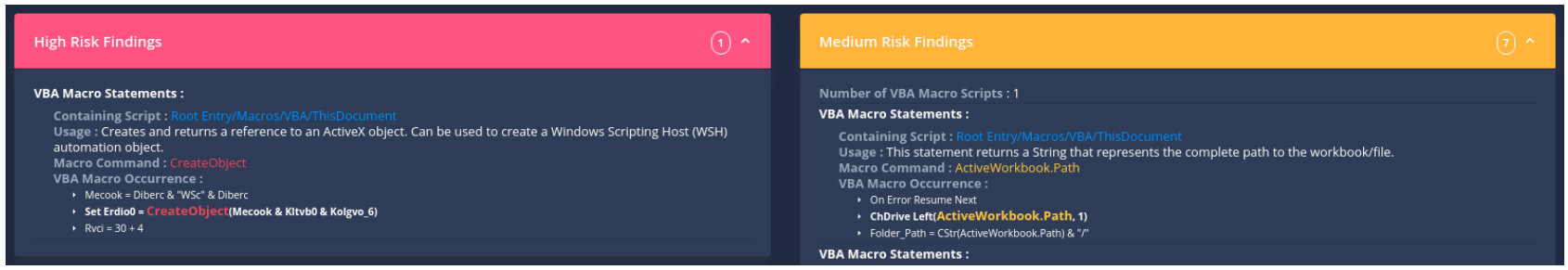


IRIS-H supports processing of password protected files. Password input fields on the submission page can be used to provide one before uploading a file. Important to note, there is a file upload size limit of 10MB enabled for all supported file types. Once processing is completed successfully, the browser is automatically redirected to the report page.

The report page is split into sections with each containing data and its corresponding description for the components discovered by IRIS-H in the processed file. Depending on the type of the file and its content, IRIS-H will generate views and UI panels to allow easier data preview and extraction. The uploaded file and its individual components can be viewed and downloaded through a hex viewer panel at the very top of the report.

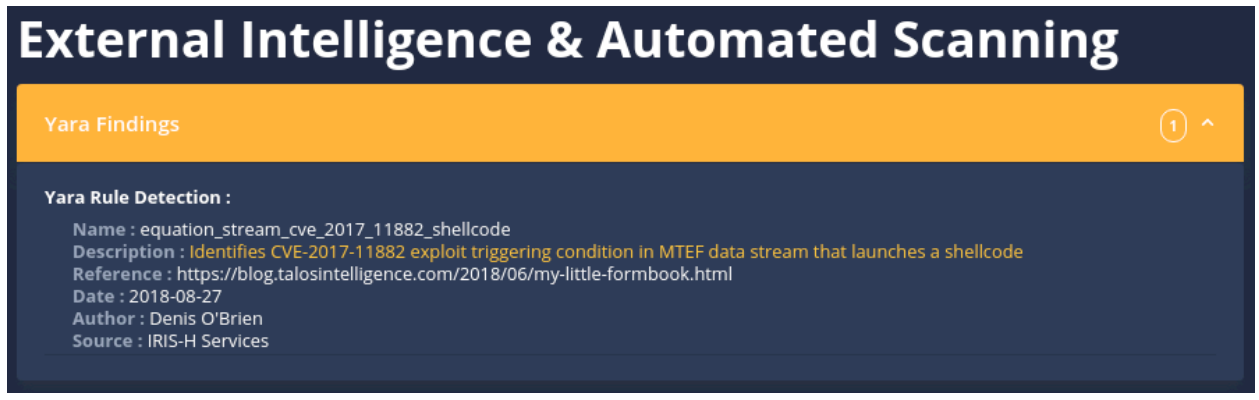


Anything IRIS-H discovers through its rules based analysis and metadata post-processing, will be grouped in the findings section of the report, just below the hex viewer panels.



Findings section panels enable a high level view of the elements that can be considered harmful to a computer system if used maliciously. The findings are color coded based on the potential risk involved. Standard color patterns of red, yellow, green are used to tag the findings. Where findings are related to the extracted code of detected executable content, such as VBA or JavaScript, the panels will also contain extracts from the code with dangerous/suspicious commands highlighted with corresponding color code.

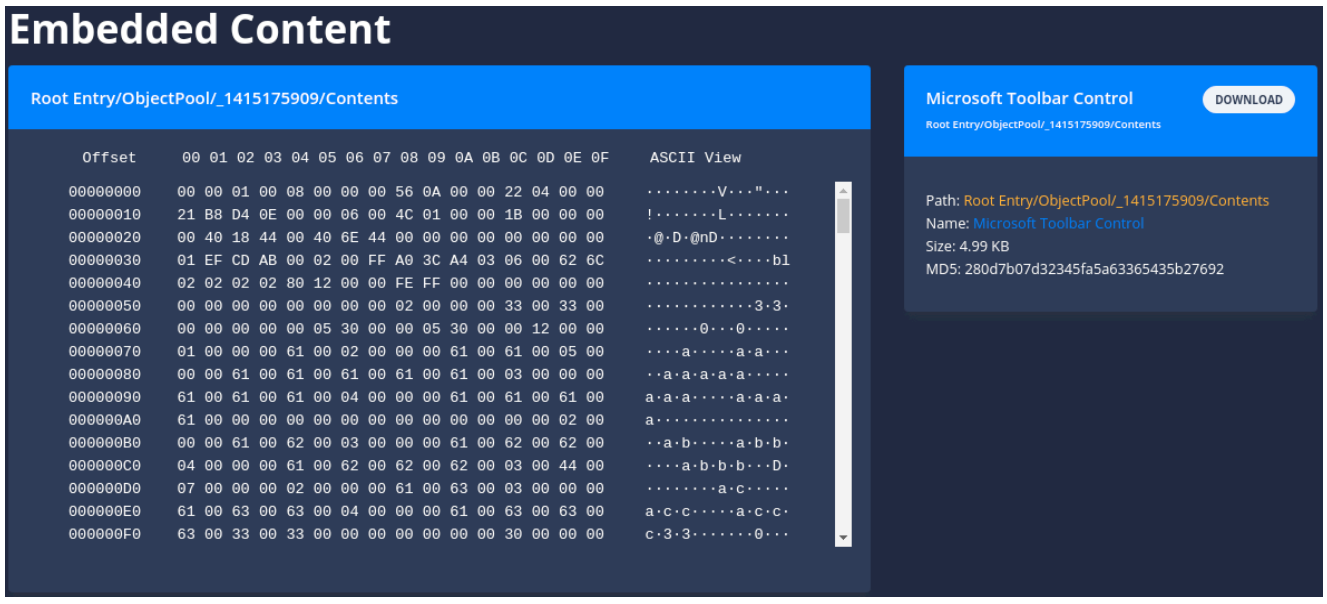
IRIS-H employs implementation of a YARA rules scanning engine for NodeJS to screen the submitted files and their individual components. It uses a community compiled collection of YARA rules maintained by Florian Roth.



All the decoded, uncompressed or otherwise transformed into original state data found in the file components is scanned for presence of hyperlinks. Any hyperlinks discovered will also be displayed in the findings panels color coded according to the risk level associated with discovered links. The risk assessment is based on the URI part analysis or the URL discovered. For example, links leading to a file with an executable extension (e.g. .exe) will be color coded as red.

One may notice the 'Auth' menu group in the sidebar. Personal accounts are supported by IRIS-H, but functionality to register an account is not currently enabled. At the moment, the only advantage of having a personal account is to keep the files' analysis reports private. Reports for the files submitted when a user is logged in are accessible only to that user. Even though account registrations are currently not available, you can reach out to the IRIS-H author to request an account, but do consider there's no full support for that feature yet and it has not been fully tested.

IRIS-H will create a separate section on the report page for any embedded content detected in the submitted file. For example, OLE objects embedded into an RTF file or Microsoft InkEdit Control into an Office Document. The panels in this section will allow for the embedded content preview and download.



The bottom section on the report page contains panels showing data extracted from the file's individual components. An example of such a component can be the 'WordDocument' stream of a Microsoft Office Word document saved in OLE-CF format. The panels will contain extracted intelligence for each stream processed with color coding applied to some elements to highlight them to the user analysing the file.

For people who like to analyse their files on the go, IRIS-H offers integration with MalScanBot. Files can be uploaded to IRIS-H through the MalScanBot Telegram channel using any mobile device.

Future Plans

The task of supporting an online service in your free time is challenging, but not impossible. The addition of new features to IRIS-H is slow and involves a higher risk of introducing a bug. This is why the tool is still in a beta development phase, meaning the stability or accuracy is not always guaranteed. Here are some future milestones though:

- add support for HWP (Hanword) files
- introduce public API for file submission and analysis report retrieval
- create AWS Lambda functions to help process different types of data obfuscation
- expand integration with other online services, such as polyswarm.io
- further UI improvements to enhance data navigation experience on mobile devices

- enable personal account support to allow for private files processing, meaning the analysis report will only be available to the submitter
- create stand alone docker container running IRIS-H tool to allow on-premises deployment

Making future plans would be nothing without input from IRIS-H users. You can reach out with any feedback, suggestions or bug reports directly to the author through Twitter handle @Malwageddon or GitHub IRIS-H project page.

Conclusion

Performing Malware Analysis or Digital Forensics can be challenging sometimes. Every specialist in those fields has a personal toolkit to help them navigate through vast spaces of digital data. IRIS-H is just one of those tools that can be used to simplify getting access to the data otherwise hidden deep in a blob of binary data.

References

https://github.com/akveo/nebular

https://github.com/chunyenHuang/hummusRecipe

https://github.com/nolze/msoffcrypto-tool

https://github.com/bontchev/pcodedmp

https://github.com/nospaceships/node-yara

https://aws.amazon.com/lambda/

https://aws.amazon.com/elasticbeanstalk/

https://polyswarm.io/

https://t.me/MalScanBot

About the author

My name is Denis O'Brien. I've been in Information Technology and Information Security fields for slightly over 25 years. I've been wearing a good number of hats during that time, ranging from entry level positions in IT Support to managerial roles. Currently leading a local Security Operations Center for a global Fortune 500 company.

MwMon - Malware Monitoring

by Vlad Ioan Topan

Malware behavior analysis has been done to death by this point - not just by professionals employed by AV companies, as the case used to be some 15 years ago, but even as an enjoyable Saturday afternoon by security hobbyists, and everybody in-between. From the rather terse, few and far between malware descriptions of the past, which usually lumped together samples into "families" and focused almost exclusively on file-infectors (the *original* viruses), nowadays a plethora of websites provide in-depth descriptions of malware behavior, almost always inferred from monitoring the OS API calls of a sample automatically inside a controlled (and usually virtualized) environment. Off the shelf complete environments have been created for hobbyists and "professionals" alike (the most popular among them being Cuckoo Sandbox).

The OS provides all the access to resources - both hardware (memory, processors, network and sound cards, etc.) and virtual (files, network sockets, the Windows Registry, etc.) - through documented APIs implemented in shared libraries included with the OS (DLLs on Windows, .sos on POSIX systems). Monitoring the way in which these APIs are called provides insight into detailed, specific events (e.g. *program X connected to TCP port Y on host Z*), which can then be analyzed heuristically to infer higher-level behavior (e.g. *ransomware X attempts to connect to a list of potential C2 servers, which it read from an embedded XML configuration file*). A wide variety of techniques exist that allow basic API monitoring, but there are few tools available for free and usable for malware analysis - *MwMon* was

born with this specific goal, to provide a basic tool that can monitor malware API usage. Before going into the specifics, let's review the API monitoring techniques and some public tools and libraries that implement them.

API Monitoring

Although some specific APIs can be monitored explicitly via, well, other APIs (usually internal and at best partially documented) provided by the OS for that purpose, most APIs are monitored by hooking the corresponding function calls.

Explicit monitoring is provided, for example, for DLL loading by using LdrRegisterDllNotification() and providing a callback (PLDR_DLL_NOTIFICATION_FUNCTION) that will receive the event type (load/unload) and information about the DLL (in particular, the full path, the load address and the SizeOfImage). An example of using this callback can be found in *MwMon* in src/um/libhook.c in register_hook_LdrLoadDll(), which registers the callback callback_LdrLoadDll() to install hooks in DLLs loaded after *MwMon*'s initialization.

Hooking allows monitoring any APIs, and can be done in a very large number of ways. For example, monitoring DLL loading could also be achieved by hooking LdrLoadDll() in ntdll.dll. Technically, in this context, hooking means interjecting the flow of an event before it starts or after it has completed, either to alter the course or to monitor the event and its outcome. In the case of Windows APIs, hooking before the call allows monitoring (and altering) the parameters and/or preventing the call altogether, while hooking after the call is used to monitor the result of the call and/or the output parameters.

Briefly, the following techniques can be used to monitor API calls:

- *inline hooks* (inserting jump opcodes at the entry of the API's code) - this is the classic (and most commonly used) technique
- *patching the import table* (IAT)
- *patching the export table* (EAT)
- *removing execution permissions on pages* (uncommon, somewhat complex to do correctly, but harder to detect)

- *proxy DLLs*
- abusing various OS mechanisms (see below)
- *kernel drivers* and *hypervisors*

The first technique was even used by Microsoft (on 32-bit Windows versions) to provide support for hot-patching (live patching of the OS), so a 5-byte position-independent stub is present at the start of all APIs on 32-bits (the sequence mov edi, edi; push ebp; mov ebp, esp, which translates to 8B FF 55 8B EC as machine code). On 64 bits, a disassembler engine is required to be able to replace a correct number of bytes (the x86 ISA uses variable-length instructions) and to be able to correct for non-relocatable instructions (e.g. relative jumps) in the "stolen" bytes. Most such disassemblers "cheat" by only being aware of a small subset of the (extremely vast) x86 ISA, usually just the opcodes, which are likely to be at the start of a Windows API. This technique is evidently easily (and often) detected by malware (APIs don't normally start with JMPs), but can also be concealed in various ways.

Patching the IAT means overwriting the address of an imported function in a program's (or a library's) import table with a pointer to the hooking code; patching the *EAT* works similarly, but involves patching the address of an exported symbol in its library's export table. Both techniques can be detected by comparing the in-memory values with the values found in the library's file on-disk (after applying appropriate relocations). Both are less common due to being less flexible than JMP-patching (the IAT and EAT values won't be used by the library itself, so only APIs called directly by the program will be monitored; also, separate hooks will be required for the *WCHAR* and *ASCII* variants of an API, despite the fact that the ASCII variant usually calls the WCHAR one internally).

Way back when dinosaurs roamed the Earth (during the prime time of Windows XP), Windows DLLs were not protected by WFP, making it easy to replace them with stubs that could log the parameters and then call the functions in the original DLL. The stubs can be generated automatically (scripted) for relevant functions and uninteresting functions can be opaquely forwarded to the original DLL (via explicit DLL forwarding).

The techniques reviewed by *aionescu* in his Hooking Nirvana 2015 presentation - TTD ("Nirvana"), AppVerifier, MinWin (API Sets), AppCompat shims and CFG (Control Flow Guard) hooks - are stealthier

API monitoring techniques, but with significantly more complex implementation requirements abusing the respective OS features.

Kernel drivers and *hypervisors* can be used to monitor application behavior at the lowest level (inside the OS and below it, respectively) via various memory instrumentation techniques (e.g. removing the execute permission from memory allocations that contain the DLL code and handling the generated faults or simply hiding the user-mode inline hooks, for example, by transparently removing read access to the memory and handling the faults by returning a "clean view" of the API entrypoint). While being almost impossible to detect, these are also the most complex to write (a hypervisor requires massive amounts of fine-tuned code). Although *MwMon* uses a kernel component strictly for monitoring process creation and injection purposes (see next section), the hooks are strictly user-mode.

As a side-note, malware that detects API hooks will usually still (try to) run, despite *generally attempting to avoid monitoring environments*, because API hooking is widely used by AV engines, which malware must be "compatible" with - it may however attempt to disable the hooks if possible.

Readily-available SDKs and libraries for API monitoring (such as *Microsoft's Detours* or the amazing Frida) can be used for malware monitoring, but are easy to discover by malware (due to their popularity) and are not necessarily designed for this purpose (an early attempt I made with *Frida* as a hooking engine kept crashing due to malware's anti-debugging tricks). Most of them also can't track new process creation to follow malware instances triggered for example by the *task scheduler*, which leads us into the next (brief) topic before putting it all together.

Process creation monitoring

Beside API tracing, malware monitoring also requires *keeping track of process creation*, to allow injecting the monitoring component (usually a DLL) into new processes. Various techniques exist for process creation monitoring as well, including:

- Appinit_Dlls - comes with DLL injection out of the box, the downside is it only works on processes that load user32.dll
- defining a Debugger in Image File Execution Options - only works if the executable name is known beforehand (and static)

- starting a malware process in a suspended state, hooking all process creation APIs and recursively starting those in a suspended state, injecting the hooking DLL, hooking process creation APIs, etc. - this fails when new instances are not spawned directly (but through mechanisms like the *task scheduler*, which is popular nowadays)
- using a kernel mode component (driver) that registers a process creation notification via PsSetCreateProcessNotifyRoutine() or PsSetLoadImageNotifyRoutine() - *MwMon* uses the latter (see DriverEntry in src/km/mwmondrv.c, the callback is ImageLoaded() that further calls MwmInjectMwmonDllInCurrentProcess()) to inject into all newly created processes

Depending on the process creation monitoring technique, a corresponding *DLL injection* will usually follow. The injected DLL will hook APIs using whichever method was chosen and employ some method of tracking new DLLs being loaded (either by using LdrRegisterDllNotification() as explained above or by hooking LdrLoadDll() or one of the higher-level DLL load APIs) to hook them as well.

MwMon

MwMon was designed as a very simple (in terms of code size) but effective tool to trace malware behavior. The code is still in alpha stage, many features are yet to be implemented, but the basic malware tracing functionality works; most convenience features (such as configuring the behavior of the driver dynamically) are still missing (as of Feb. 2020).

To try it out, first make sure you have an isolated VM (not connected to any network) and a piece of malware. *CAUTION*: if you have no experience running live malware samples, **start out with benign programs and read about safely experimenting with malware** - some samples might infect your network or escape the VM into the host!

Building *MwMon* only requires the *Microsoft Enterprise WDK* (an ISO image containing the build tools and user mode + kernel mode SDKs) and is documented on the project's page. Binary releases are published periodically (when enough code changes) here. Deploying the driver and DLLs is documented on the homepage as well - essentially the driver must be installed using the provided .bat file and the hook DLLs must be copied to the appropriate system32 locations. The VM must be in "Test Signing Mode" because the driver is not signed by Microsoft/WHQL.

Monitoring a malware sample using *MwMon* starts either by loading the driver (which will inject the hook DLL in all newly spawned processes) or by passing a malware sample as a parameter to mwmon32.exe or mwmon64.exe (depending on the malware image type being PE32 or PE32+). In the latter case, the malware will be spawned suspended and the DLL will be injected using inject_mwmon_dll() in src/um/libhook.c; spawned processes will not be traced.

The PsSetLoadImageNotifyRoutine() kernel-mode callback used by *MwMon* allows manipulating the process memory at a very early stage (before any executable image is loaded in the process space), but any meaningful work must be delayed until at least ntdll.dll is loaded to avoid unnecessary complications. A hook is set on LdrLoadDll() that allows ntdll.dll and kernel32.dll to load, then loads the hook DLL (mwmonhk[32|64].dll) and removes itself. The relevant part of a debug log of net.exe starting looks like this:

```
[mwmondrv] <net.exe:1592> Image \Device\HarddiskVolume2\Windows\System32\net.exe (64) loaded @ 0
[mwmondrv] <net.exe:1592> Image \SystemRoot\System32\ntdll.dll (64) loaded @ 0x00007FF957CE0000
[mwmondrv] <net.exe:1592> Image \Windows\System32\kernel32.dll (64) loaded @ 0x00007FF957160000
[mwmondrv] Injecting mwmonhk64.dll (ntdll@0x00007FF957CE0000) into net.exe:1592...
[mwmondrv] Injecting into process 1592 (net.exe)...
[mwmondrv] Prepared the 64-bit shellcode @ 0x0000025B19C50000 (110 bytes)
[mwmondrv] Patched LdrLoadDll @ 0x00007FF957CF5AD0 (page: 0x00007FF957CF5000 [0x2000])
[mwmondrv] <net.exe:1592> Image \Windows\System32\KernelBase.dll (64) loaded @ 0x00007FF95A57000
```

net.exe starting with debugging enabled

Once the hook DLL is loaded, it will iterate over all loaded DLLs and iterate the hook list for a matching DLL name; if one is found, an import and an export with that entry's function name is searched for in the DLL. If found, the corresponding *IAT*/*EAT* entry is altered to point to the hook code (and in the case of *IAT* entries, the original address is preserved in the hook structure). On 64 bits a *hook gate* is used due to the way in which export *RVA*s are processed (as unsigned numbers - sign extension is not performed - so the hook address must be greater than the *EAT*'s address to obtain a *VA* that points correctly to the hook gate when adding the *ImageBase*).

An alternative mechanism can be enabled (by setting CFG_HOOK_ALL_EXPORTS to 1 in src/umkm/mwmcfg.h) that installs generic hooks on (almost) all functions (some are excluded based on a whitelist to simplify the code or to avoid irrelevant spam). Four parameters are dumped for each function; parameters that look like valid pointers are checked heuristically for strings (ASCII or WCHAR) and dumped as such. A more efficient alternative to this is to enable CFG_HOOK_GETPROC_EXPORTS,

which will install generic hooks dynamically on all APIs resolved through GetProcAddress(), which are not explicitly hooked.

The actual hooks are implemented in src/um/hooks.c; more than 100 APIs are hooked, which are generally interesting in terms of malware behavior, but more can be added. The process to add a new hook is simple: place one or more C-like API function declarations from an SDK header (or from the MSDN help page) in the file tools/sample_api_def.txt and append (as a comment) the DLL name to each function, e.g.:

```
PCWSTR StrStrIW(
 PCWSTR pszFirst,
 PCWSTR pszSrch
); // shlwapi.dll
```

Run tools/gen_hook_code.py - functional hook code will be written to stdout (the console window by default). From that output, add:

- the hook function code to hooks.c
- the structure definition (which looks something like {hook_SomeApiFunction, "somedll.dll", "SomeApiFunction"},) to the HOOK Hooks[] list at the bottom of hooks.c
- the HOOKID_SomeApiFunction, enum field (at the same index as the Hook structure) to the anonymous enum at the start of inc/um/libhook.h.

A sample commit that adds logging for StrStrIW() in shlwapi.dll (and also adds that previously not-hooked DLL to the list of hooked DLLs) is here: 0fb1592d.

In custom hooks, binary buffers can be dumped with the save_binary_buffer() function (see examples throughout src/um/hooks.c). All the output (logs and binary buffers) goes to the OUT_PATH folder defined in inc/um/common.h (set to c:\mwmon by default).

Examples:

Sodinokibi (16a3dc86bb16ea5c03b9d7fb660a9ccb)

Running the Sodinokibi sample from this any.run analysis yields the following interesting pieces of information (in c:\mwmon\<sample-name>-<bits>-P<PID>-api.log):

- MultiByteToWideChar() is used to convert many interesting strings to WCHAR, e.g. {EXT}-readme.txt (the pattern for the ransom file)
- powershell.exe is called with a base64-encoded script to delete shadow copies (file backups):

```
<T1104> APICALL: CreateProcessW("NULL", "powershell -e RwBlAHQALQBXAG0AaQBPAGIAagBlA
```

```
>>> b64decode('RwBlAHQALQBXAG0AaQBPAGIAagBlAGMAdAAgAFcAaQBuADMAMgBfAFMAaABhAGQAb
wB3AGMAbwBwAHkAIAB8ACAARgBvAHIARQBhAGMAaAAtAE8AYgBqAGUAYwB0ACAAewAkAF8ALgBEAGUAb
ABlAHQAZQAoACkAOwB9AA==').decode('utf16')
'Get-WmiObject Win32_Shadowcopy | ForEach-Object {$_.Delete();}'
```

- an iterative search for files to encrypt starts from C:, with the ransom file written in each folder:

```
<T1104> APICALL: FindFirstFileW("\\?\C:\*", 0x0019E7BC) => 0x0264DFF8
<T1104> APICALL: CreateFileW("\\?\c:\program files\vnj01vxo-readme.txt",
0x40000000, 0x0, 0x00000000, 0x2, 0x0, 0x00000000) => 832
```

APT33 / Turnedup (3d3642bc6274255577661c4357217df4)

The sample from this Cape analysis, which appears to be from the *Turnedup* family, provides a nice example of process injection via hollowing:

```
<T2860> APICALL: CreateProcessA("C:\Program Files (x86)\Internet Explorer\iexplore.exe",
<T2860> APICALL: GetModuleHandleA("ntdll.dll") => 0x77320000
<T2860> APICALL: ZwUnmapViewOfSection(); args: 0x00000234, 0x00400000, 0xDB7BD2E1, 0x0000
<T2860> APICALL: VirtualAllocEx(); args: 0x00000234, 0x00400000, 0x00069000, 0x00003000;
<T2860> APICALL: WriteProcessMemory(0x00000234, 0x00400000, 0x00780000, 0x00000400, 0x001
<T2860> [-] Writing to [\??\c:\mwmon\stikynote.exe-32-P584-WriteProcessMemory\stikynote.e
<T2860> APICALL: WriteProcessMemory(0x00000234, 0x00401000, 0x00780400, 0x00045400, 0x001
<T2860> [-] Writing to [\??\c:\mwmon\stikynote.exe-32-P584-WriteProcessMemory\stikynote.e
<T2860> APICALL: WriteProcessMemory(0x00000234, 0x00447000, 0x007C5800, 0x0000C000, 0x001
<T2860> [-] Writing to [\??\c:\mwmon\stikynote.exe-32-P584-WriteProcessMemory\stikynote.e
<T2860> APICALL: WriteProcessMemory(0x00000234, 0x00453000, 0x007D1800, 0x00003200, 0x001
<T2860> [-] Writing to [\??\c:\mwmon\stikynote.exe-32-P584-WriteProcessMemory\stikynote.e
<T2860> APICALL: WriteProcessMemory(0x00000234, 0x0045C000, 0x007D4A00, 0x00008E00, 0x001
<T2860> [-] Writing to [\??\c:\mwmon\stikynote.exe-32-P584-WriteProcessMemory\stikynote.e
<T2860> APICALL: WriteProcessMemory(0x00000234, 0x00465000, 0x007DD800, 0x00003600, 0x001
<T2860> [-] Writing to [\??\c:\mwmon\stikynote.exe-32-P584-WriteProcessMemory\stikynote.e
<T2860> APICALL: GetThreadContext(); args: 0x000000F8, 0x001BF080, 0x00780000, 0x00000000
<T2860> APICALL: WriteProcessMemory(0x00000234, 0x00B6F008, 0x001BF078, 0x00000004, 0x001
<T2860> [-] Writing to [\??\c:\mwmon\stikynote.exe-32-P584-WriteProcessMemory\stikynote.e
<T2860> APICALL: GetProcAddress(0x77320000 <C:\Windows\SYSTEM32\ntdll.dll>, "ZwResumeThre
<T2860> APICALL: ZwResumeThread(); args: 0x000000F8, 0x00000000, 0x00780000, 0x00000000;
```

Turnedup - process injection via hollowing

The injected sections are saved to disk (as WriteProcessMemory() buffers) to c:\mwmon \stikynote.exe-32-P<PID>-WriteProcessMemory and can be reconstructed as an EXE (which in this particular case is less useful as it is an exact replica of stickynotes.exe, which itself is a copy of the original malware sample).

Trojan (527adb1483f5474c48c14d73b40d2fd3)

The unidentified trojan from this Cape analysis is a WinRAR self-extractor that unpacks files with randomly pre-generated names to a folder and then starts a .vbs script from there:

```
<T2876> APICALL: ShellExecuteExW(0x006FEED8: lpVerb="NULL", lpFile="C:
\i0066q0720\fxxucp.vbs", lpParameters="NULL", lpDirectory="NULL")
```

The .vbs (running under vbscript.exe, which is subsequently traced) provides a nice view into the AMSI scanning process as implemented by Windows Defender - first, AmsiScanString() is called and passed the script as an argument (which also provides a good hooking point to drop executed intermediary scripts to disk):

```
<T3892> APICALL: AmsiScanString(); args: 0x000002166323E1C0,
0x0000021664F630F0, 0x0000021663241498, 0x0000000000000000; retaddr:
0x0000000000000000
<T3892>  - arg.2: IHost.CreateObject("WScript.Shel
<T3892>  - arg.3: C:\i0066q0720\fxxucp.vbs
```

This triggers the loading of mpclient.dll, whose exports are dynamically resolved:

```
<T3892> APICALL: LoadLibraryExW("C:\Program Files\Windows Defender
\MPCLIENT.DLL", 0x0000000000000000, 0x8) => 0x00007FFFBBFA0000
<T3892> APICALL: GetProcAddress(0x00007FFFBBFA0000 <C:\Program Files\Windows
Defender\MPCLIENT.DLL>, "MpManagerOpen") => 0x00007FFFE8100380 (dyn-hook:
yes/0x00007FFFE8100380)
<T3892> APICALL: GetProcAddress(0x00007FFFBBFA0000 <C:\Program Files\Windows
Defender\MPCLIENT.DLL>, "MpHandleClose") => 0x00007FFFE81003C0 (dyn-hook:
yes/0x00007FFFE81003C0)
<T3892> APICALL: GetProcAddress(0x00007FFFBBFA0000 <C:\Program Files\Windows
Defender\MPCLIENT.DLL>, "MpFreeMemory") => 0x00007FFFE8100400 (dyn-hook: yes/
```

```
0x00007FFFE8100400)
<T3892> APICALL: GetProcAddress(0x00007FFFBBFA0000 <C:\Program Files\Windows
Defender\MPCLIENT.DLL>, "MpScanStart") => 0x00007FFFE8100440 (dyn-hook: yes/
0x00007FFFE8100440)
[...]
```

and eventually MpAmsiScan(...) is called:

```
<T3892> APICALL: MpAmsiScan(); args: 0x0000021663289660, 0x000000C3C8F4D720,
0x000000C3C8F4D830, 0x0000000000000001; retaddr: 0x0000000000000001
```

(which is hooked generically as CFG_HOOK_GETPROC_EXPORTS was set, so the parameters are not very informative).

Fun alternative uses

Having an MwMon-based API monitoring system in place can be used for other (fun) things as well - particularly with regard to analyzing "undocumented" application behavior (particularly useful with applications that treat the end user as the "product" and raise privacy concerns). Having hooks on send()/recv() allows monitoring applications that "call home", allowing you to see the data buffers passed back and forth under the SSL encryption. One such potentially fun experiment is hooking MpTelemetryInitialize() and related functions in *Windows Defender*'s mpclient.dll to see what gets uploaded into the "cloud". It is also possible to *block an application's access* to the Internet altogether or to the local filesystem (or parts of it) as a basic form of sandboxing while maintaining all other functionality intact. Do note that disabling kernel-mode signature enforcement to allow *MwMon*'s driver to load significantly weakens the security posture of the machine, so it shouldn't be done on production machines.

Bugs or crashes can be investigated as well in any application and without relying on the source code - think of this as an alternative, configurable Process Monitor that can log any APIs (not limited to the standard file, registry, network and process categories in the original) and can perform actions when certain conditions are met, e.g. extracting more information or breaking into a debugger.

Investigating undocumented Windows behavior is also possible, such as monitoring all the registry reads performed by a Control Panel applet to figure out where specific settings are stored.

A parser to adnotate call <register> instructions in an IDA disassembly could be created based on return addresses, making reverse engineering easier when dynamic imports are used.

Potential improvements

The dynamic configuration and control of both the UM and KM components (via an IOCTL or minifilter ports, for example) is necessary to allow behavior changes that don't require full rebuilds of the entire solution.

The next large feature likely to be added is dumping the main executable's memory image at process detach to allow static analysis (as most malware samples are packed and encrypted, and at that stage more often than not the code is unpacked and decrypted in memory). The PE header needs slight adjustments (i.e. adjusting on-disk section sizes to fit the in-memory layout), and the import tables must be recreated heuristically to allow meaningful analysis. Better heuristics to detect unpacked code can be implemented (e.g. the classic "memory executed after being written" pattern) to handle cases where the code is unpacked to heap buffers in chunks and deleted after execution.

Converting the single-source hook definition to a pluginable interface that would allow any DLL to register a hook would be a huge convenience boost for adding new APIs. Basic behavior altering capabilities could also be implemented as plugin DLLs - for example returning altered data for VM detection APIs (e.g. hiding VM-specific devices, files and registry keys in listings).

Automatic analyzers and report generators can be created based on the logs and dumped buffers to highlight potentially interesting calls in the vast mass of data. Higher-level behavior can be inferred (as explained in the introductory section).

No hiding of the monitoring components is attempted - randomizing filenames on build and other evasive techniques could be employed.

Beside the functional improvements, performance improvements are needed (e.g. binary search in string tables instead of sequential comparisons, etc.).

Final words

As-is, *MwMon* provides insight into specific API use by malware (to allow monitoring expected behavior in detail) and generic monitoring of unknown, dynamically-resolved APIs, to point towards what other APIs could be interesting to watch for. The code is sufficiently documented to allow hacking on it for any specific purposes and, given the very limited amount of time available to be invested in this project, contributions (in the form of pull requests) are most welcome!

Happy hunting!

About the Author

I am an Information Security enthusiast - I've been working in InfoSec for about 14 years on antivirus & network security, most of those years at Bitdefender, currently focusing on vulnerability research (and when time allows it on pentesting, malware analysis and machine learning)

A Python tool for Robust Detection on Advanced Digital Image Copy-Move Attack by using a Modification of Two Algorithms

by Rahmat Nazali

We will introduce two algorithms taken from a previous work: the first one is titled duplication detection method, while the second one is titled simply robust detection method. For the sake of simplicity, let's just say **first algorithm** and **second algorithm**. The first algorithm was effective to be used on a normal Copy-Move attack, meaning it ran fast but only detected a simple attack, and it will likely turn false positive when run on an advanced attack. While the second algorithm was effective to be used on an advanced Copy-Move attack, meaning the run time is much slower, it will be likely more reliable to detect an advanced copy-move attack. Our proposed algorithm combines those two algorithms, up to certain cases with a certain tolerance, and able to adapt towards its input condition. Therefore, the image preprocess stage is no longer needed, with a trade-off in a slightly longer run time than the first algorithm, but as robust as the second algorithm. For this proposed algorithm, we have created a simple tool that implements that exact logic, that you may find here https://github.com/rahmatnazali/image-copy-move-detection.

Introduction

Copy-move is one type of an attack to fraud a digital image where the attacker duplicates some areas of the image and pastes it in different places but still on the same image so that a particular section from the image can be hidden.

After forgery, advanced methods like noise addition and blurring are often done on the image to make it harder to be recognized. Therefore, it is required to do a preprocess on the image such as filtering for noise removal before it can be directly detected by a naive algorithm. Assuming we have a more reliable algorithm that works even on noise addition and blurring, we can skip the image preprocess and save the running time. In previous research, we proposed a new flexible algorithm robust enough to detect both normal and advanced Copy-Move using a modification and several minor additions from two algorithms.

We will introduce two algorithms taken from a previous work: the first one is titled duplication detection method [4], while the second one is titled simply robust detection method [3]. For the sake of simplicity, let's just say **first algorithm** and **second algorithm**.

The first algorithm was effective to be used on a normal Copy-Move attack, meaning it ran fast but only detected a simple attack, and it will likely turn false positive when run on an advanced attack. While the second algorithm was effective to be used on an advanced Copy-Move attack, meaning the run time is much slower, it will be likely more reliable to detect an advanced copy-move attack.

Our proposed algorithm combines those two algorithms, up to certain cases with a certain tolerance, and able to adapt towards its input condition. Therefore, the image preprocess stage is no longer needed, with a trade-off in a slightly longer run time than the first algorithm, but as robust as the second algorithm.

For this proposed algorithm, we have created a simple tool that implements that exact logic, that you may find here https://github.com/rahmatnazali/image-copy-move-detection.

Previous Related Work

There are many cases that involve Copy-Move attacks that happened in the newspaper [1] as well as an official report [2]. A paper [3] suggests that people do not stop at doing a simple Copy-Move attack. Some of these attacks are followed by a process called post region duplication process [3] to further complicate the detection of the attacks by doing a certain pixel modification process after the main attack was done. These processes include blurring, sharpening [3], JPEG compression, and noise addition [4].

There are also several research studies that have a different approach for detecting a Copy-Move attack. Some of these approaches are color filter interpolation [5], re-sampling [6], computing its whole pixel [7] or by dividing it into several chunks of data [8]. Every method above has its own advantages, but they all have the same weakness, they cannot detect an image attacked by Copy-Move that had post region duplication process done to it.

Post region duplication process will change the certain region's pixel value or even the whole image's pixels, so using the algorithm that only compares individual pixel value is far from enough to detect duplicated regions. A paper provides methods that can obtain a certain characteristic of a chunk of an image region that will not change significantly when the post region duplication process was applied [3].

Design

The design of our proposed algorithm involves two methods described in Previous Related Work, namely duplication detection method [4] and robust detection method [3]. Both of these methods have a similarity in the concept where they both divide the ***MxN*** sized input image into ***LxL*** overlapping blocks, hence named overlapping technique.

After extracting a set of the overlapping block from the image, we then do a feature extraction to obtain the features of each image block. Both methods (i.e. duplication detection algorithm and robust detection algorithm) have a different way of obtaining the features of each image block. The duplication detection method takes the features from a decomposition process called Principal Component Analysis (PCA), while the robust detection method takes the features from its own pixel

value comparison rules. Those sets of features are then sorted based on the value of the features, so that the same image block will likely have identical feature values.

The first method: duplication detection algorithm proposed the use of principal component analysis to compute the feature of the image block that was already described in the first algorithm paper. This method is quite fast, but when the input image was noised or blurred then this method could not be relied on. This method already has a predefined assumption that the duplicated region could be not only one pair. This method uses the offset system to represent the distance of a block image from other corresponding block images. Offset used in this research is a tuple representing data, having two elements ***x*** and ***y*** that represent two-dimensional coordinates. This method applies the mathematical absolute method to obtain the absolute value when the offset calculation occurs so that a block with a negative distance can be treated to have the same offset. This absolute method is considered to give this method a tolerance, but this tolerance will make a pair of a block that accidentally have the same offset value too close together, while it is also possible that those pairs are a false positive.

The second method: robust detection algorithm proposes the use of pixel value comparison to calculate the features as described in the second algorithm paper. This method is much slower because calculating the comparison of pixel value involves redundant memory access to obtain the certain pixel value. The first three features represent the mean of the pixel value in a specific color channel, which means there are three iterations times the sum of the pixel (i.e. image size) to compute the total pixel value from red, green, and blue from RGB color space, while the next four features will need four iterations times the sum of the pixel to be obtained.

This method utilizes a histogram to measure the offset frequency, then takes the offset with the highest frequency. This flow makes the method only be able to detect exactly one region of duplication since only one offset is chosen from a set of offsets.

On the other side, this method is proved from the previous research to be robust in handling low-quality input image even without a preprocess method (i.e. noise reduction, smoothing). Some of the examples of a low-quality input image is an image that contains a considerable amount of noise or has been through a blurring process. This feature can be achieved because the characteristic features used are considered robust towards the image manipulation process. For instance, a blurring process is a

method to average certain pixels in a region and so it will not change the total pixels in that region. Since the total pixel in the region remains unchanged, the value will be identical or at least close to the value of the characteristic features, because the characteristic features themselves are obtained from the comparison of the pixel value. This condition makes the method to be considered robust.

We then try to fix the weaknesses from each of the methods. From the first method, we remove the absolute operation from the offset calculation formula. This makes the offset range from a negative number to positive number (formerly only zero to infinity, because of the absolute process). This type of offset is more accurate since this type will distinguish offsets that have the same value but different kind (i.e. positive number or negative number). On the other side, the former second method will choose exactly one offset with the highest frequency. We then change it so that it will have a frequency threshold variable that limits the offsets to be chosen. This means that we can take more than one offset, provided all of the taken offset has an offset frequency larger than the threshold.

Using two methods that have already been improved, we created a new algorithm that uses these algorithms altogether and creates a kind of tolerance technique so that a certain method will not dominate another method. This tolerance technique involves in round operation towards the characteristic features so that nearly all of the features could take a part when the features sorting comes.

The design of the proposed algorithm starts with computing characteristic features from each of the two methods and store it in a single container at once. Thus, we have a container that contains three pieces of data, namely the block coordinate, characteristic features from the first method (i.e. the principal components), and characteristic features from the second method (i.e. the pixel value comparison explained previously), shown in Figure 1.

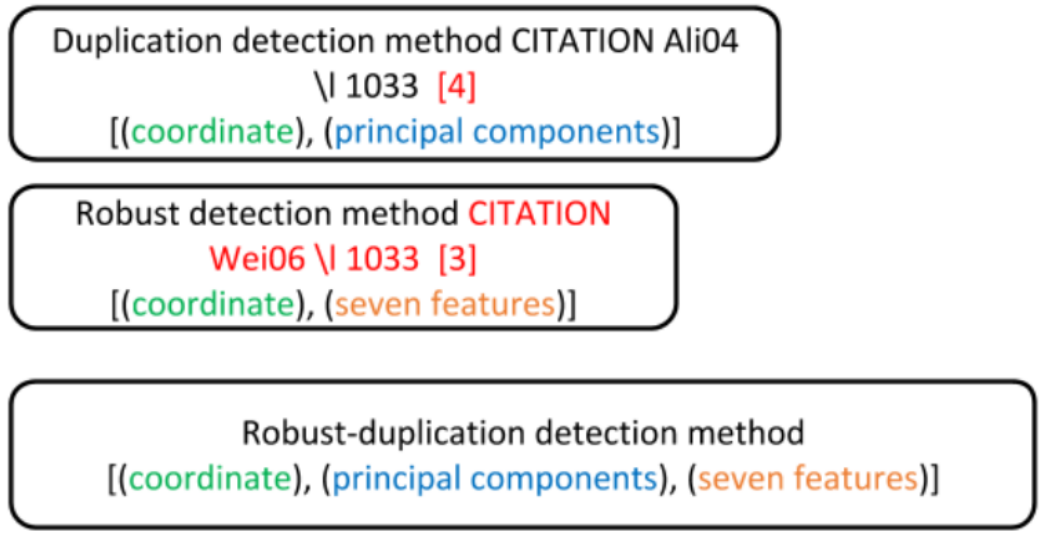


Figure 1 Proposed design

Each of these characteristic features then goes through a rounding process towards a certain precision. At a glance, this round operation will decrease the accuracy from each of the methods, but a tolerance will be obtained because the loss of domination from every method due to their features that have a decreased precision. The tolerance effect will be very useful when features sorting takes place.

This container is then sorted using a lexicographical sorting method based on the value of the characteristic features from each of the methods. This sorting process will draw close the image blocks that have similar characteristic features, meaning that those blocks are suspected to be identical or at least similar.

Knowing that identical or similar image blocks will always be close to each other, we create pair sets of an image block from the sorting result that is close to another within a certain neighboring distance threshold.

A filtering process will then be done to the sets of the image block to remove a pair that did not meet a certain threshold condition. This filtering process will leave only paired sets of an image block that are more likely suspected to be identical or at least similar in form. Finally, an image answer will be created based on each coordinate of the remaining pair sets of an image block.

Proposed Method & Implementation

The robust-duplication detection method starts with dividing ***M***x***N*** sized image to become several ***L***x***L*** sized image blocks, where ***L*** is relatively low compared to ***M*** and ***N***. These created image blocks have an overlapping feature, meaning that a block will always overlap several image blocks that are neighboring it. We will then call the amount of the created overlapping blocks as ***Nb***, and can be calculated using Formula 1 [4] below.

$$N_b = \left(\frac{M - L + 1}{i}\right) \times \left(\frac{N - L + 1}{j}\right) \qquad (1)$$

After obtaining the overlapping blocks, a feature extraction process is done to further obtain several features of every image block. This step will be branched into two sub parts: the first feature extraction will compute the principal components of the image blocks as shown in the first algorithm paper, while the second step will compute the pixel value comparison features according to the robust detection method as shown in the second algorithm paper.

The first step contains two processes. In the first process, a principal component analysis process is done. All of these image blocks that contain ***b*** pixel are then processed as follows:

An input array ***x*** is prepared. This array contains ***Nb*** elements of one dimensional vector-based pixel value of the image blocks. For a grayscale image, each of the elements will have length exactly the same as the amount of the pixels. As for the colored image, the element will be three times the amount of pixels (namely red, green, and blue, assuming we are using RGB color space).

After we have created the input array, we then start to compute the principal components as explained in the first algorithm paper. The result of this process will be stored in ***s*** matrix along with their corresponding coordinate.

The second process is to compute the seven features as described in the second algorithm paper and store the result to the corresponding element in ***s*** matrix. We will then obtain a matrix formatted as seen in Figure 2 that contains the image block coordinate, its principal components, and its seven features.

```
[(0, 0), [192, 199, 212], [0.994528077103 4, … and so on …]]
 [(0, 1), [192, 199, 212], [0.994425990580 1, … and so on …]]
 [(0, 2), [192, 199, 212], [0.994248572039 8, … and so on …]]
 [(0, 3), [192, 199, 212], [0.994138876197 5, … and so on …]]
 [(0, 4), [192, 199, 211], [0.993919291937 1, … and so on …]]
 [(0, 5), [192, 199, 211], [0.993862933369 3, … and so on …]]
 [(0, 6), [192, 199, 211], [0.993660371175 4, … and so on …]]
 [(0, 7), [192, 199, 211], [0.993473993173 3, … and so on …]]
 [(0, 8), [192, 199, 211], [0.993222267582 6, … and so on …]]
```

Figure 2 An example of pieces of the feature computation result from 10 overlapping blocks using robust-duplication detection method

Step three, this ***s*** matrix is then sorted with lexicographical sorting method by its first and second characteristics (i.e. the principal components and the seven features). After the ***s*** matrix is sorted, the identical or similar image blocks will be close together.

Step 4, we now create an offset matrix container, namely ***t*** matrix. We then generate a combination of two coordinate ***(xi, yi)*** and ***(xj, yj)*** taken from ***s*** matrix with a neighbor threshold of ***|i-j| < Nn*** , with ***i*** and ***j*** are indexes that correspond to the ***s*** matrix, and ***Nn*** is the predefined neighboring threshold.

Step five, compute every ***t*** matrix element's offset using self-made formula, Formula 2 [4], and store it in a result container, say ***u*** matrix.

(2)

$$offset = (x_i - x_j, y_i - y_j)$$

Step six, we discard all sets of coordinates from ***u*** matrix that has an offset frequency lower than ***Nf*** (offset frequency threshold), followed by step seven to discard all sets of coordinates from ***u*** matrix that has offset value lower than ***Nd***, the offset magnitude threshold. The offset value can be computed using Formula 3 [4] below.

(3)

$$N_d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

The last step, we make a ground truth image by first creating a zero image (i.e. an image with all pixels colored black) and set the pixel's color to white to a region obtained from each of the sets of coordinates left in ***u*** matrix. We can then know that the suspected regions attacked by Copy-Move are the white colored region.

Experiment and Result

The dataset we use in this experiment is based on public dataset [11], but the majority of them did not meet our limitation. Many of them are categorized in multi paste (i.e. the duplicated regions are overlapped to each other). Further, we also realize that no public dataset met the case where we want to examine how our algorithm identifies an advanced attack because none of them involved a post region duplication process. As a solution, we decided to create our own dataset that consists of 30 images, with 10 images as authentic images, another 10 images as a normal Copy-Move attack images, and the last 10 images are advanced Copy-Move attack images that were created by blurring certain edges of the duplicated region. Some of the used public datasets are shown in Figure 3, while some of the datasets are shown in Figure 4.

For the experiment, we use a predefined threshold value of ***Nn*** to be 2, ***Nd*** to be 50, and ***Nf*** to be 750. We have three experiment scenarios. Scenario 1 is used to examine how the three algorithms react to an image that is actually authentic and whether any of them have false positives using 10 images from their own dataset. Scenario 2 is used to examine how the three algorithms react to a normal Copy-Move attack (i.e. no involvement of post region duplication process). Scenario 3 is used to examine how the three algorithms react to an advanced Copy-Move attack (i.e. involves post region duplication process). All of the results of these scenarios are shown based on Table 1. Furthermore, all of the explanations are also based on Table 1.

Scenario		Method		
		A	B	C
Authentic image	Accuracy	100%	100%	100%
	MSE	0	0	0
	Similarity	100%	100%	100%
Normal Copy-Move	Accuracy	83%	90%	100%
	MSE	542.40	1023.98	580.01
	Similarity	99.60%	99.37%	99.59%
Advanced Copy-Move	Accuracy	75%	80%	100%
	MSE	7504.83	2538.93	2002.91
	Similarity	96.03%	98.60%	98.86%
Normal Copy-Move from public dataset	Accuracy	56%	80%	80%
	MSE	6881.08	3772.86	3772.14
	Similarity	96.47%	98.07%	98.07%

Table 1 The experiment results from each scenario by (A) duplication detection method, (B) robust detection method, and (C) robust-duplication detection method

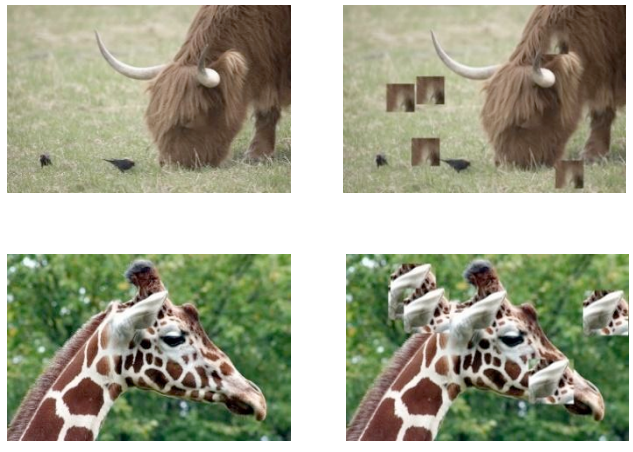


Figure 3 Examples of an original (left side) and an attacked (right side) public dataset image

A

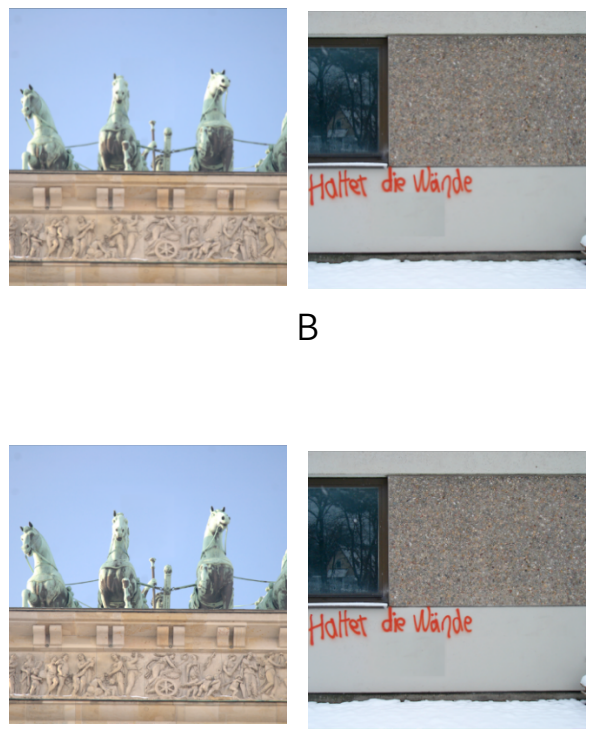
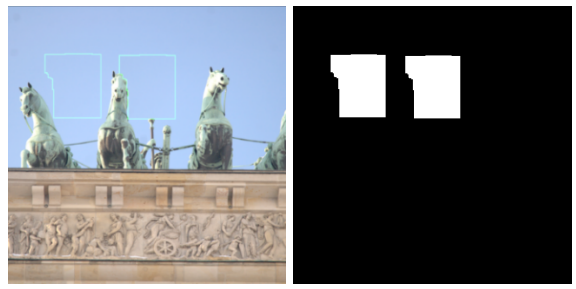


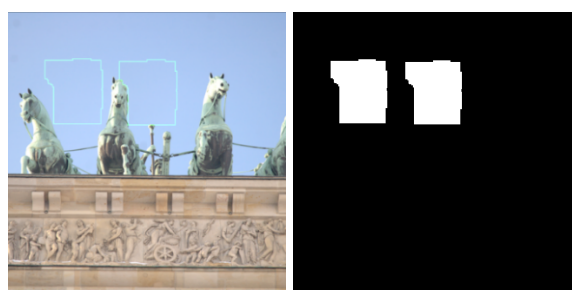
Figure 4 (A) one of the authentic public dataset (i.e. not attacked by Copy-Move), (B) own dataset created from a public dataset with normal Copy-Move attack, and (C) own dataset created from a public dataset with advanced Copy-Move attack

The average MSE of scenario 1 is zero for all three methods. This value means that all the methods can identify if the input image contains no Copy-Move region (i.e. no false positive condition is raised).

Based on the average MSE of scenario 2, the best algorithm to identify a normal Copy-Move attack is duplication detection algorithm, followed by our proposed algorithm, and surpassed by robust detection algorithm that has the lowest average MSE. These results are proven because we know that for a normal Copy-Move attacks, the duplication detection algorithm is better than the robust detection because the utilization of principal components is very accurate to be used to represent the image block's pixel values and far better than the pixel value comparison features. Our proposed algorithm could adapt to the condition and manage to achieve better average MSE than the robust detection algorithm. The comparison between some algorithm's results is shown in Figure 5.



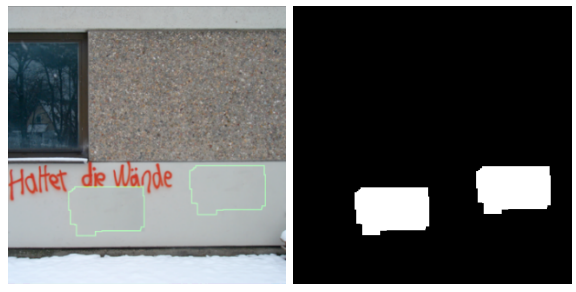
A



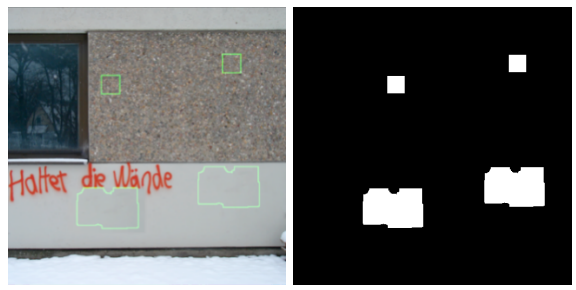
B

Figure 5 (A) Our proposed algorithm compared to (B) robust detection algorithm for a normal copy move attack.

Based on the average MSE of scenario 3, the best algorithm to identify an advanced Copy-Move attack is our proposed method, robust-duplication detection, followed by robust detection algorithm and then duplication detection algorithm. These results are also proven because we know that for an advanced attack (e.g. involves a blurring operation) is better identified with robust detection algorithm because of the pixel comparison features that are robust against the change of pixel value, and that duplication detection algorithm will not work efficiently because when a pixel value changes, albeit a bit, the principal components of the image block will also change and hereby affects the sorting procedure. Our proposed algorithm can manage to adapt and the sorting procedure could tolerate the broken principal components and use the pixel value comparison features instead. The comparison between some algorithm's results is shown in Figure 6.



A

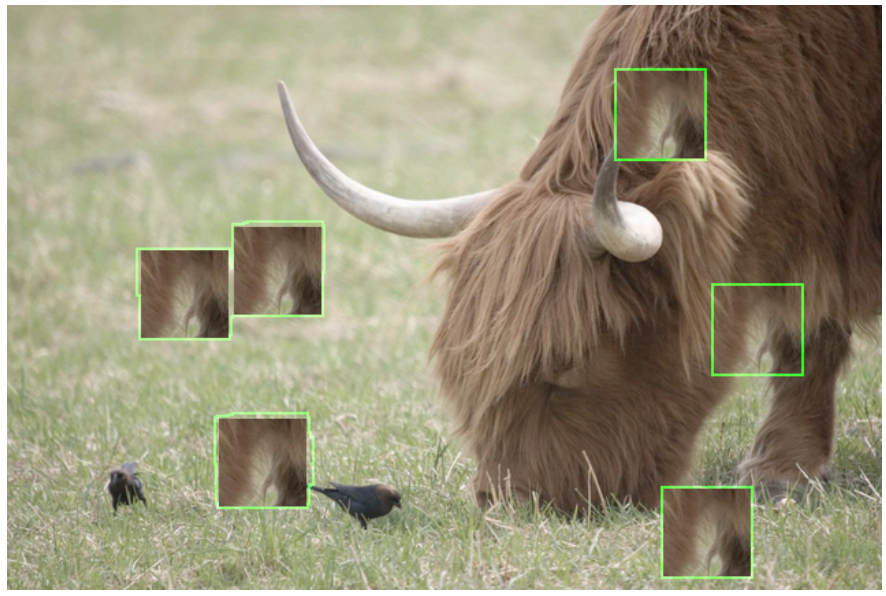


B

Figure 6 (A) Our proposed algorithm compared to (B) duplication detection algorithm for an advanced Copy-Move attack

From scenario 1 and scenario 2, we can know that our proposed method can adapt the variety of input image without determining a false positive towards the authentic image. As for scenario 3, we intend to use public dataset to represent the general experiment result, but with a limitation that we only use public dataset images with the duplicated region that did not get overlapped with each other. We also know that all of the selected public dataset images are considered to be a normal Copy-Move attack (i.e. without the involvement of post region duplication process) and so this scenario is actually identical to scenario 2.

According to the average MSE of the result images, our proposed algorithm is considered the best way to detect the chosen public dataset images, followed by robust detection algorithm, and lastly duplication detection. This proves that for a general case like public dataset, our proposed algorithm is capable to be used. Some results of this scenario using our proposed algorithm are shown in Figure 7.



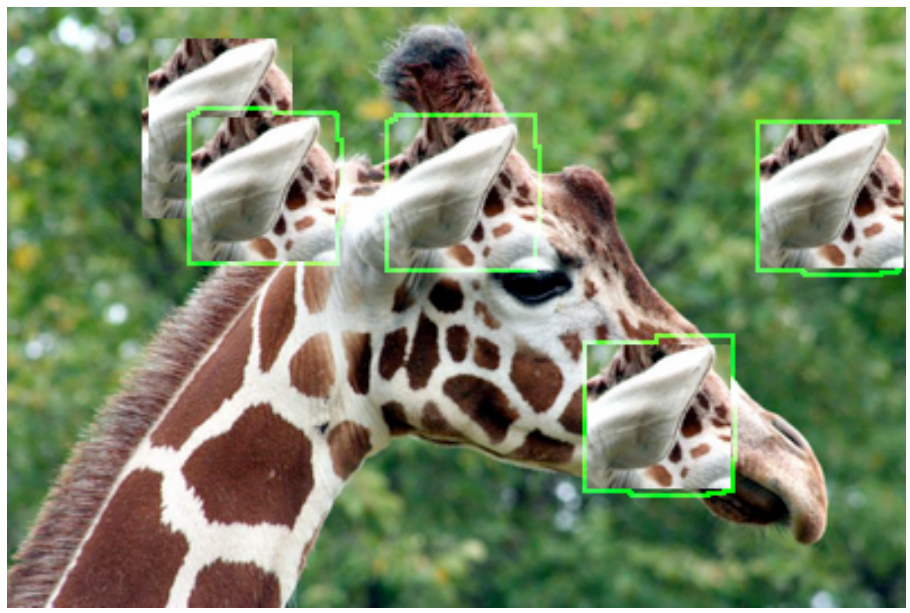


Figure 7 Examples of result image using robust-detection algorithm from public dataset

Running the tool

You can follow the step by step tutorial on the repository and run the tool with your own dataset. We included some example runner scripts inside the *example* folder. A brief capture of the console running the tool is like below:

```
/home/rahmat/tmp/image-copy-move-detection/venv/bin/python /home/rahmat/tmp/image-copy-move-detection/examples/example_01.py
dataset_example_blur.png
Step 1 of 4: Object and variable initialization, 231361 True
Step 2 of 4: Computing characteristic features
100%|██████████| 481/481 [08:58<00:00,  1.12s/it]
Step 3 of 4:Pairing image blocks
100%|██████████| 231361/231361 [00:01<00:00, 221291.59it/s]
Step 4 of 4: Image reconstruction
Found pair of possible fraud attack:
(-137, -2) 1624
Lossy conversion from float64 to uint8. Range [0.0, 255.0]. Convert image to uint8 prior to saving to suppress this warning.
Computing time : 538.7985460758209 second
Sorting time   : 0.8897430896759033 second
Analyzing time : 1.1465582847595215 second
Image creation : 0.6998012065887451 second
Total time     : 0:09:01 second

Done.

Process finished with exit code 0
```

Figure 8 Capture of running console

The running time is greatly affected by how large the image is, how small the overlapping block we picked, and of course, the specification of the device that you are using. In the later stage of the tool, it will print the pair(s) of coordinates found, if any, that strongly indicates a copy-move attack. From the Figure 8, it shows a pair: **(-137, -2) 1624**, meaning that 1624 overlapping blocks are detected to be identically mapped towards -137 point of x axis (to the left) and -2 point of y axis (to the bottom).

Conclusion

The implementation of our combined algorithm, roughly named *robust-duplication detection algorithm*, can be used as one of the digital image Copy-Move attack detection methods, effective for both a normal attack and an advanced attack that involves post region duplication process without requiring any preprocess stage.

We picked a blurring process as a simple representation of post region duplication process, and examined how the algorithm reacts to several types of input image and find that our method could adapt along the type of the input image. When a normal attack (i.e. pure Copy-Move) is done then the principal components feature will become dominant for the attack identification rather than the pixel comparison feature. Then on a case when an advanced attack (i.e. Copy-Move that involves post region duplication process) is done, the pixel comparison feature will become dominant rather than the

principal component feature. The automatic adaptation of these two features could be achieved because we implemented a lexicographical sort to a rounded collection of feature pairs. The rounding process will make the two features lose some of their precision to a certain level, and the lexicographical sorting will make it much looser. All of these advantages are limited to the fact that this method still could not detect Copy-Move attack regions that overlap each other, and are known limitations.

References

[1] K. K. Light. Fonda and P. Fakery, *The Washington Post, Saturday,* p. A21, 28 February 2004.

[2] "Sorry.. we were hoaxed: Iraqi PoW abuse pictures handed to us WERE fake," *Daily Mirror Newspaper,* 15 May 2004.

[3] W. Luo and J. Huang, "Robust Detection of Region-Duplication Forgery in Digital Image," *IEEE The 18th International Conference on Pattern Recognition (ICPR'06),* 2006.

[4] A. C. Popescu and H. Farid, "Exposing Digital Forgeries by Detecting Duplicated Image Regions," *Dartmouth College Technical Report,* pp. 1-11, 2004.

[5] A. Popescu and H. Farid, "Exposing Digital Forgeries in Color Filter Array Interpolated Images," *IEEE Trans. on signal processing,* pp. 3948-3959, 2005.

[6] A. Popescu and H. Farid, "Exposing Digital Forgeries by Detecting Traces of Resampling," *IEEE Trans. on signal processing,* pp. 758-767, 2005.

[7] J. Fridrich, D. Soukal and J. Lukas, "Detection of Copy-Move Forgery in digitals Images," *Proc. of Digital Forensic Research Workshop,* 2003.

[8] T.-T. Ng and S.-F. Chang, in *A Model for Image Splicing*, ICIP, pp. 1169-1172 Vol.2.

[9] H. Omrani and R. G. Beiragh, "Performance Assessment of Iranian Electricity Distribution Companies by an Integrated Cooperative Game Data Envelopment Analysis Principal Component Analysis Approach," *International Journal of Electrical Power and Energy Systems,* vol. 64, pp. 617-625, 2015.

[10] D. Vogan, "Lexicographic Order," Massachusetts Institute of Technology, [Online]. Available: http://www-math.mit.edu/~dav/lex2.pdf. [Accessed 29 5 2016].

[11] C. Riess and J. Jordan, "Image Manipulation Dataset," University of Erlangen-Nuremberg, [Online]. Available: https://www5.cs.fau.de/research/data/image-manipulation/. [Accessed 30 5 2016].

About The Author

A hardcore Pythonist. Currently a full time software developer on a governmental scale with a passion in security, especially in digital forensic. Usually reading, writing, and solving forensic puzzles in his spare time. Feel free to reach via his github account at *rahmatnazali*.

ARTHIR - ATT&CK Remote Threat Hunting Incident Response tool

by Michael Gough

ATT&CK™ **R**emote **T**hreat **H**unting **I**ncident **R**esponse (ARTHIR) is an update and fork of the older KANSA (2) incident response framework utilizing PowerShell. KANSA was originally developed by Dave Hull in 2014 and released on GitHub in 2015 but he stopped development in 2016 after going to work for a company that makes a competing product. There are a couple articles on KANSA referenced on the KANSA Github page for more background. There is also a video from the 2015 SECKC security conference of Dave discussing KANSA's design and purpose available on YouTube (5). There have been some recent updates to KANSA to add some changes to ingest output into a logging solution, but for the most part, there has been very little work on KANSA modules since 2016.

ARTHIR follows, and did not change the original KANSA modular design, since it works, as the modular design makes it easier to add additional content without changing the master script engine. ARTHIR works differently than KANSA in that a tool, utility, or script pushed and run can have the output the tool, utility, or script natively creates retrieved with ARTHIR modules back to the launching host. KANSA only pulled PowerShell console output back to the launching host, which greatly limited its capabilities. ARTHIR can run scripts as KANSA does, even run the existing KANSA scripts, but also utilities, tools, and scripts in the scripting language of choice, making ARTHIR far more flexible for incident response, forensics, threat hunting, and auditing.

The reason ARTHIR was created was to have a way to push, run, schedule a task, pull back reports that LOG-MD creates, and delete anything that was pushed, run, or created if desired. LOG-MD (3) is a free tool used for Incident Response that produces CSV and TXT output, thus the need to have tools create output in the native format of the tool. Results and output from ARTHIR modules can be pulled back to the launching host, share, or server to be analyzed, or sent to a log management or SIEM solution.

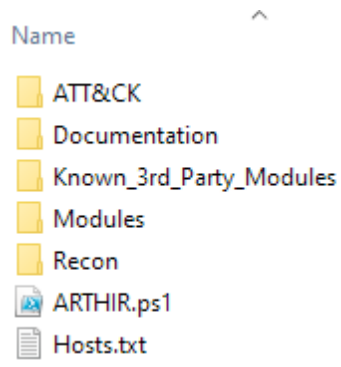
ARTHIR was created to solve a challenge many of us in the Windows realm have, the need for a tool that security and forensic analysts own, manage, and execute, and by the way, it's free. ARTHIR requires configuration of the Windows Remote Management (WinRM) service, preferably using Group Policy. WinRM is already built into every Windows system, so nothing new to install, answering the "not another agent debate". WinRM is a built-in and free option that can be used by companies and consultants to perform security investigations, incident response, forensics, hunting activities, and assessments by providing a PowerShell command line. Adding ARTHIR modules allows WinRM to do much more.

ARTHIR can be found at:

- www.ARTHIR.com
- https://github.com/MalwareArchaeology/ARTHIR

The structure of ARTHIR:

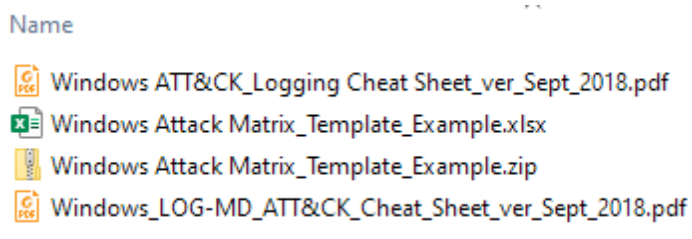
ARTHIR has several directories where items are stored and organized as shown by the following image:



HOSTS.txt contains the list of targets to run ARTHIR modules against, more on this file later. ARTHIR.PS1 is the main PowerShell script that does not get edited as it is the engine for ARTHIR.

ATT&CK:

The **A** in **A**RTHIR stands for **ATT&CK**, so it is only fitting to include some MITRE ATT&CK (4) reference information. The ***ATT&CK*** directory contains some reference documents to help users map ARTHIR modules to MITRE ATTACK Tactics and Technique IDs. The ATT&CK folder contains a cheat sheet that maps Windows Event IDs to MITRE ATT&CK Technique IDs, a blank spreadsheet to map items to MITRE ATT&CK, and a cheat sheet mapping LOG-MD capabilities to MITRE ATT&CK.



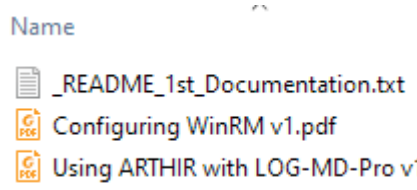
MITRE ATT&CK:

From MITRE's ATT&CK website, MITRE ATT&CK™ is described as "*a globally-accessible knowledge base of adversary tactics and techniques based on real-world observations. The ATT&CK knowledge base is used as a foundation for the development of specific threat models and methodologies in the private sector, in government, and in the cybersecurity product and service community*". Think of MITRE ATT&CK as a collection of Tactics and Techniques of what adversaries actually do to organizations during an attack that have been assigned a Name and ID to each technique. This breakdown of tactics, techniques, and assigning of IDs makes it easier to map an organization's capability as a part of the overall security program. Items such as scripts, tools, hunting, security solutions, queries, alerts and searches can be mapped to one or more Technique IDs in order to identify gaps, and understand what can and cannot be detected or blocked as far as a malicious attack. MITRE ATT&CK also maps the Tactics and Techniques to known threat actors in the event an organization knows that a specific threat actor attacks their specific industry. Red Teams (attackers) can use MITRE ATT&CK to map their testing and report the types of things that were tested using the tactics and technique IDs. If you contract for Red Team and Pentesting services and the vendor does not map their testing to MITRE ATT&CK, seriously consider another vendor. Being able to see what specific tactics and techniques a vendor uses in their attack testing can be an incredible learning opportunity for

organizations. Using the tested ATT&CK technique IDs that were not detected or blocked, provides an organization the gaps that should be addressed. Blue Teams (defenders) can use MITRE ATT&CK to map their detection and prevention capabilities, identify budget opportunities where there is little or no coverage, and address any gaps that are discovered, or found during an attack test. Red Teams working with Blue Teams, often called "Purple Teaming", and mapping efforts to MITRE ATT&CK allows a common language to be used for both teams. Using ATT&CK Tactics and Technique IDs between attacking and defending groups can improve communication to better understand an attack, what was detected or prevented, and then use the outcome to identify any gaps that need to be addressed.

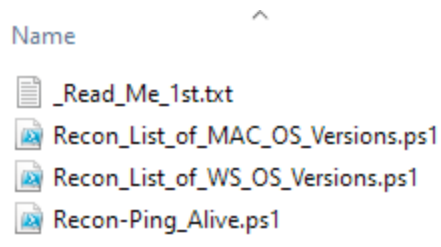
Documentation:

The ***Documentation*** folder contains some help guidance on using ARTHIR on its own, configuration of WinRM, and using ARTHIR with LOG-MD.



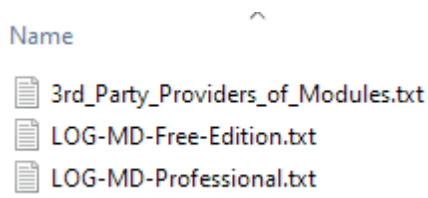
Reconnaissance Scripts:

The ***RECON*** folder contains some PowerShell scripts to perform reconnaissance, or information gathering before using ARTHIR. These scripts assist users in finding Windows systems (what ARTHIR runs on), exclude any MAC OSX systems, and then a ping script to see if the system is alive before running modules against the hosts, saving time. The ***_Read_Me_1st.txt*** file lists what documents are in the folder. These scripts are not executed by ARTHIR, rather they are run in an admin PowerShell window, and are used to obtain information, to better identify running Windows targets.



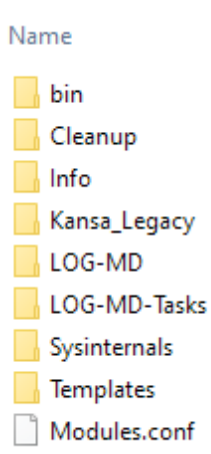
Third Party Modules:

The ***Known_3rd_Party_Modules*** folder contains third party tools that have modules created specifically for ARTHIR, in this case LOG-MD. If you want to write a bunch of ARTHIR modules for a utility or tool, this is where you would let folks know the modules are available and how to find them.



Modules:

The ***MODULES*** folder contains all the available modules for ARTHIR and are organized as seen in the following image by the type of module.



- BIN – This folder contains any tools/utilities/scripts/archives that will be pushed when using the "-Pushbin" flag.
- CLEANUP – This folder contains cleanup modules to delete items that are pushed, folders created, and/or any output that is created once the modules complete.
- INFO – This folder contains modules that gather information from a target host.
- KANSA_LEGACY – This folder contains the legacy Kansa modules that have been updated or converted to take advantage of all the ARTHIR features, and ones that have not yet been converted.

- LOG-MD – This folder contains several modules for use with LOG-MD Free Edition. If a user is a LOG-MD-Pro or LOG-MD-Premium user, then all the modules are included when purchasing LOG-MD. These modules are written for LOG-MD, but can easily be converted to any tool/utility/script that a user wants to run.

- LOG-MD-TASKS – This folder contains several Scheduled Task options, Hourly, Daily, and a cleanup module. These modules are written for LOG-MD, but can be easily converted to any tool/utility/script that a user wants to schedule.

- SYSINTERNALS – This folder contains several modules to run Sysinternals utilities such as Handles and Sigcheck.

- TEMPLATES – This folder contains templates to create modules for running a binary, script, scheduling a task, or pushing a zip archive.

MODULES.TXT:

This file is where users will select which modules to run. ARTHIR allows for running one module called out on the command line, or a series of modules in order of top to bottom as listed in Modules.txt to perform several tasks with each ARTHIR run. In order to run a module, just remove the comment "#" and delete the leading spaces as shown in the following image.

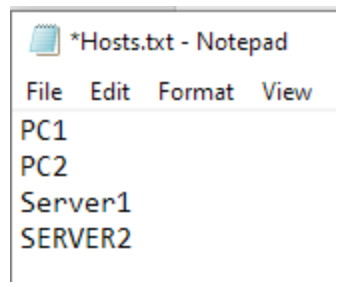
```
# Use this file to configure the modules you want to run and the order you want to run them in.
# They are organized from how long they take to run short to long.
# Long-running jobs should go at the end.
# Test each module you want to run and place the longest running ones at the end so you can
# review the results from modules that complete faster.
#
# The start and end time is listed in each job so you can plan accordingly.
#
#############################################################################################
#
#  Info gathering modules (Uses C:\Program Files\ARTHIR folder)
#
Info\Get-OS_Version_Details.ps1
#  Info\Get-PS_Version_Logging_Details.ps1
#
#  Cleanup\Get-Delete_ARTHIR_Folders.ps1
#
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#
#  Module to query LOG-MD API settings - Get VirusTotal key info
#
#  LOG-MD\Get-LOG-MD-API-Settings.ps1
#  LOG-MD\Get-LOG-MD-Create_VT_API-Settings.ps1
#
#############################################################################################
#
# LOG-MD TASKS
#
#  Modules to create regular Scheduled Tasks to run LOG-MD as a detection solution
#
## Daily Logs
#  LOG-MD-Tasks\Get-Log-MD-Pro_Task_Logs_Daily.ps1
## Daily PowerShell Logs
#  LOG-MD-Tasks\Get-Log-MD-Pro_Task_PS_Logs_Daily.ps1
## AutoRuns
#  LOG-MD-Tasks\Get-Log-MD-Pro_Task_AutoRuns_Check_VT_Hourly.ps1
#  LOG-MD-Tasks\Get-Log-MD-Pro_Task_AutoRuns_Hourly.ps1
## Large Registry Keys
#  LOG-MD-Tasks\Get-Log-MD-Pro_Task_Large_Keys_Daily.ps1
## Running Processes
#  LOG-MD-Tasks\Get-Log-MD-Pro_Task_Running_Processes_Check_VT_Hourly.ps1
```

Keep in mind the logical order modules are needed to run. Cleanup modules must go last after the reports are pulled back to the launching host. More examples of cleanup modules, Sysinternals, and Legacy Kansa modules that have been converted to ARTHIR are shown in the following image.

```
# -------------------------------------------------------------------------------------
##  Enable this if you want to remove all the LOG-MD files, deletes the directory
##  specified in the module
#
#  LOG-MD\Get-Log-MD-Pro_z_Cleanup_Reports.ps1
#  LOG-MD\Get-Log-MD-Pro_z_Cleanup_All.ps1
#
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#
# Sysinternals tools
#
#  Sysinternals\Get-Handle64.ps1
#  Sysinternals\Get-Sigcheck64.ps1
#
#############################################################################################
#
#  Legacy Kansa modules converted to ARTHIR
#
##  Configuration modules
#  Kansa_Legacy\Config\Get-Anti-MW-HealthStatus.ps1
#  Kansa_Legacy\Config\Get-Anti-MW-InfectionStatus.ps1
#  Kansa_Legacy\Config\Get-Hotfix_Patches.ps1
#  Kansa_Legacy\Config\Get-Local_Accounts.ps1
#  Kansa_Legacy\Config\Get-Local_Admin_Accounts.ps1
#
##  Log modules
#
#  Kansa_Legacy\Log\Get-AppCompatCache.ps1
#  Kansa_Legacy\Log\Get-CBS_Log.ps1
#
```

HOSTS.TXT:

This file in the root of the ARTHIR directory is where hosts are listed to run modules against. Users may also specify one host on the command line just like running a single module. Users can capitalize on the information collected with the RECON scripts to query Active Directory (AD) to get host names. Keep in mind hosts in AD may no longer exist, so a ping sweep is a good idea to determine running Windows hosts. Users can also use AD details to determine the last time a host checked in to AD, for example, the last 30, 60, or 90 days.



Running ARTHIR:

Now that the overall structure and files used in ARTHIR have been described, the execution of ARTHIR is fairly straightforward. Of course, there is some documentation with examples and details, but basically there are two conditions that users will run ARTHIR in. One, systems in a domain, and two, systems that are not in a domain, or are in a workgroup. The fundamental difference is in how WinRM is configured, but more importantly how ARTHIR will authenticate the user account specified for ARTHIR. A domain will use Kerberos by default, though users do have, and can use other options, and ***Negotiate*** is used for non-domain attached and workgroup systems that will use a local account to authenticate. Users will need a similar local administrative account and password on all non-domain attached devices to make running ARTHIR easier by defining one credential to scan all systems.

$Credential variable allows users to define a username and password and pass it to the remote system using secure PowerShell calls. Every ARTHIR execution after this variable is populated until the PowerShell session is closed, will use these credentials. If the $Credential is not used, then each time ARTHIR is executed, the user will be prompted for credentials. To populate $Credential with a username and password type the following and fill in the dialog box when prompted as seen in the following image:

ARTHIR runs in an administrative PowerShell window. Once a PowerShell session is opened, the following are examples of how to execute ARTHIR with the hosts.txt and modules.txt files, or specifying a single system and module, with and without a domain:

For Domain attached systems:

- .\ARTHIR.ps1 -TargetList hosts.txt .\Modules -Pushbin -Transcribe -Verbose -Credential $Credential

  ★Run all modules in Modules.txt on all hosts in Hosts.txt, Push a binary specified in one or more modules, add Transcription of what ran, use Verbose logging, and use the credential in $Credential using default Kerberos Active directory for the credential

```
PS D:\ARTHIR> .\ARTHIR.ps1 -TargetList hosts.txt .\Modules -Pushbin -Transcribe -Verbose -Credential $Credential
```

- .\ARTHIR.ps1 -Target <computername> -ModulePath ".\Modules\LOG-MD\Get-Log-MD-Pro_AutoRuns.ps1" -Pushbin -Credential <username>

  ★Run the Get-Log-MD-Pro_Autoruns.ps1 module on host <computername>, Push a binary specified in Get-Log-MD-Pro_Autoruns.ps1 module, and use a credential specify at runtime using default Kerberos Active directory for the credential

```
PS D:\ARTHIR> .\ARTHIR.ps1 -Target Bobs-PC -ModulePath ".\Modules\LOG-MD\Get-Log-MD-Pro_AutoRuns.ps1" -Pushbin -Credential Bob
```

For non-domain attached (standalone or workgroup) systems:

- .\ARTHIR.ps1 -TargetList hosts.txt .\Modules -Pushbin -Authentication Negotiate -Transcribe -Verbose -Credential $Credential

  ★Run all modules in Modules.txt on all hosts in Hosts.txt, Push a binary specified in one or more modules, add Transcription of what ran, use Verbose logging and negotiate the credentials in $Credential using a local account

```
PS D:\ARTHIR> .\ARTHIR.ps1 -TargetList hosts.txt .\Modules -Pushbin -Authentication Negotiate -Transcribe -Verbose -Credential $Credential
```

- .\ARTHIR.ps1 -Target <computername> -ModulePath ".\Modules\LOG-MD\Get_Log-MD-Pro_1_Configs.ps1" -Pushbin -Authentication Negotiate -Credential <username>

  ★Run the Get-Log-MD-Pro_Autoruns.ps1 module on host <computername>, Push a binary specified in Get-Log-MD-Pro_Autoruns.ps1 module, and use credentials specified at runtime using a local account

```
PS D:\ARTHIR> .\ARTHIR.ps1 -Target Bobs-PC -ModulePath ".\Modules\LOG-MD\Get_Log-MD-Pro_1_Configs.ps1" -Pushbin -Authentication Negotiate -Credential Bob
```

Once ARTHIR is started with the configured modules, the console will show the status of the execution listing the module(s) that will be run, the host(s) they will be run against, as well as the status and output folders. The results are located as seen by the following image:

```
PS D:\ARTHIR> ./ARTHIR.ps1 -TargetList hosts.txt .\Modules -Pushbin -Authentication Negotiate -Transcribe -Verbose -Credential $Credential
VERBOSE: Start Time - 2/29/2020 2:48 PM
VERBOSE: Found .\Modules\Modules.conf.
VERBOSE: Running modules:
Get-OS_Version_Details
Get-PS_Version_Logging_Details
VERBOSE: $Targets are DEFENDER Bob.
VERBOSE: Waiting for Get-OS_Version_Details to complete.

Id     Name            PSJobTypeName   State         HasMoreData     Location             Command
--     ----            -------------   -----         -----------     --------             -------
1      Job1            RemoteJob       Completed     True            DEFENDER             <#...
D:\ARTHIR\Output_20200229144845\Get-OS_Version_Details\DEFENDER\DEFENDER-Report_System_Info.txt
VERBOSE: Waiting for Get-PS_Version_Logging_Details to complete.
4      Job4            RemoteJob       Completed     True            DEFENDER             <#...
D:\ARTHIR\Output_20200229144845\Get-PS_Version_Logging_Details\DEFENDER\DEFENDER-Report_PS_Dot_Net_Versions.txt
VERBOSE: End Time - 2/29/2020 2:48 PM
```

Mapping MITRE ATT&CK in ARTHIR Modules:

The **A** in **A**RTHIR stands for **ATT&CK** and ARTHIR modules take into account a place to record what ATT&CK Tactics and/or Technique IDs users might map a module to help in mapping hunts, incident response and forensic investigations to MITRE ATT&CK. Below is an example using the LOG-MD Autoruns module mapped to MITRE ATT&CK. For those modules that address many ATT&CK Technique IDs, as the following image shows, users can create a grouping ID as shown for ease of tracking multiple Technique IDs.

```
<#
.SYNOPSIS
Get-LOG-MD-Pro_AutoRuns.ps1 returns output from AutoRuns (ASEP)
compared to the MasterDigest for find malicious or unknown autoruns.

Use the following to record the modules applicability to the MITRE ATT&CK Framework

MITRE ATT&CK Technique IDs: T1015 (Accessibity Features), T1182 (AppCert Dlls), T1103 (AppInit Dlls)
                            T1131 (Auth Package), T1122 (COM Hijack), T1183 (Image File Exec Options injection)
                            T1177 (Lsass Driver), T1031 (Modify Existing Service), T1050 (New Service),
                            T1013 (Port Monitors), T1060 (Reg Run/Startup folder), T1053 (Schedule Task),
                            T1084 (WMI Event Subscription)
ATT&CK TechID Grouping ID:  TGxxxxx
```

Ease of Module Creation:

Along with the many samples to help create modules, ARTHIR modules are designed to use variables that are easily changed to adjust where ARTHIR modules are placed, the directory used for output that will be retrieved, naming of reports to make it easier to retrieve output, renaming of reports to add a systemname, create a log entry on the host ARTHIR ran, and the name of the report(s) that will be retrieved as shown in the following image.

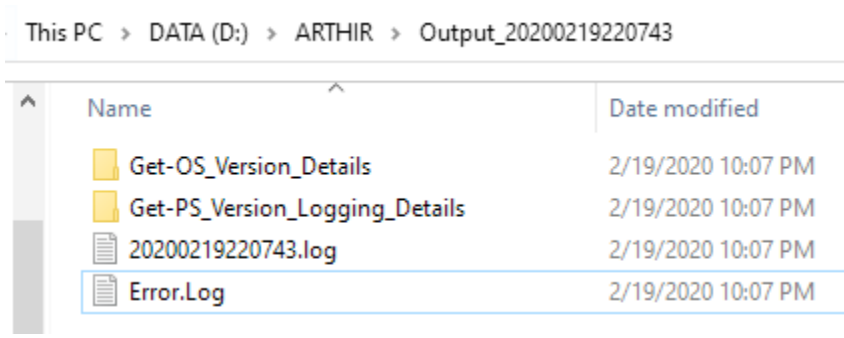
```
If you want to remove the binary and/or reports from remote systems after it has run
use the cleanup module(s) or specify with the $DeleteReports and $DeleteAll variables.

Adjust the variables to what you want to do with each item:
  $ARTHIR_Dir                   Set to a directory you want the tool to be stored
  $ARTHIR_OutputDir             Set to a directory you want the results of the modules to be stored for harvesting
  $ARTHIR_ReportName            What to name the report.  Match this to DOWNLOAD
  $RenameReports                Yes/No - Rename the reports to include the systemname or what you specify with $SysName variable
  $SysName                      What you want each report to be pre-pended with such as "computername"
  $WriteEventLogEntry           Create an event log entry that this module ran 'Yes' or 'No'
  $EventSource                  The name of the source the event will be written to the Application log (default is ARTHIR)
  $Event_ID                     What event ID to use in the log entry

  BINDEP                        The name of the binary/file you want to push to the remote systems
  DOWNLOAD                      The name of the report you will copy back to the host or launching system, wildcards are acceptable
```

The use of variables in the modules allows the actual PowerShell code to be left alone and to just change the variable information to adjust many popular items as shown in the following image:

```
.NOTES
The BINDEP and DOWNLOAD directives are needed by ARTHIR.ps1 to determine where to find
the binary to be used and how to handle output from this script.
Use the wildcard * to capture the systemname in the report.
 - Example:  C:\Program Files\LMD\Results\*Report_AutoRuns*

BINDEP .\Modules\bin\Log-MD-Pro.exe
DOWNLOAD C:\Program Files\LMD\Results\*Report_AutoRuns*
#>
$Tool_Name = "LOG-MD-Pro.exe"
$ARTHIR_Dir = "C:\Program Files\LMD"
$ARTHIR_OutputDir = "C:\Program Files\LMD\Results"
$ARTHIR_ReportName = "Report_AutoRuns.csv"
$RenameReports = "Yes"
$SysName = $env:computername
$WriteEventLogEntry = "Yes"
$EventSource = "ARTHIR"
$Event_ID = "1337"
```

Output:

ARTHIR will create a folder that is time stamped each time ARTHIR is run so output is not overwritten if run multiple times for similar or different modules. Under the time stamped directory will be a folder of each module that is successfully run with the results, and ***error.log*** for any errors or systems that were unreachable when the module ran. Details from the ***-Transcript*** and ***-Verbose*** flags will be recorded in the timestamp.log file as seen in the following image.



Use Cases for ARTHIR:

ARTHIR is limited only by the user's imagination. With the existing modules and sample modules included with ARTHIR, users can create or edit modules to meet their unique needs. The structure and variables used in the modules helps to make the modules more consistent and require less complex PowerShell coding for basic functions. ARTHIR is by no means a replacement for enterprise configuration, remote management, log management, or other security solutions, rather it augments what you have, or provides something for companies with limited budgets. ARTHIR is also useful for

consultants that find a client does not have a way to do remote Incident Response and Forensics efforts.

Incident Response:

ARTHIR can be used for Incident Response triage, and deep investigations. This allows users to integrate their favorite command line tools in a way to be executed on one, dozens, or more systems, pull back the results to a single system, and cleanup all traces of being there, except what will be logged in event logs as ARTHIR will make noise that will end up in the event logs. ARTHIR is heavily commented so that excluding ARTHIR activity is easier, especially if consuming information into log management or a SIEM.

Forensics:

ARTHIR can be used for Forensics investigations or to launch forensics command line tools and utilities. ARTHIR can push a memory dump utility like winpmem, pull back the memory dump, or push it to a server share, hashing it along the way.

Threat Hunting:

ARTHIR can be used to hunt for known artifacts or indicators, and validate certain conditions or artifacts do not exist that indicate malicious activity on other systems discovered during an investigation. Users can create hypotheses to hunt for, or use other hunting command line utilities, such as LOG-MD, yara, etc.

Compromise Assessments:

ARTHIR can be used to assess systems for any signs of existing compromise by consultants, merger and acquisition teams, or as a regular process to look for indications of compromise, or validate there are none.

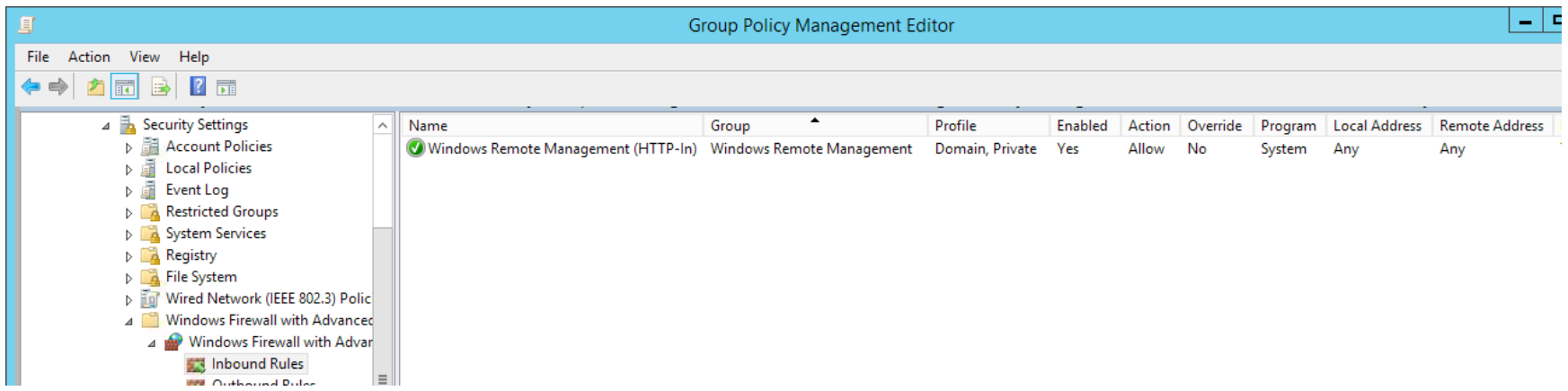
Audit Windows settings and Advanced Audit Logging:

ARTHIR can be used to assess crucial settings Incident Response, Forensics, and Threat Hunters want and need to do their job duties effectively. For example, LOG-MD has a free option to assess the Windows Advanced Audit Policy Configuration against the CIS Benchmarks, US GCB, Australian Cyber

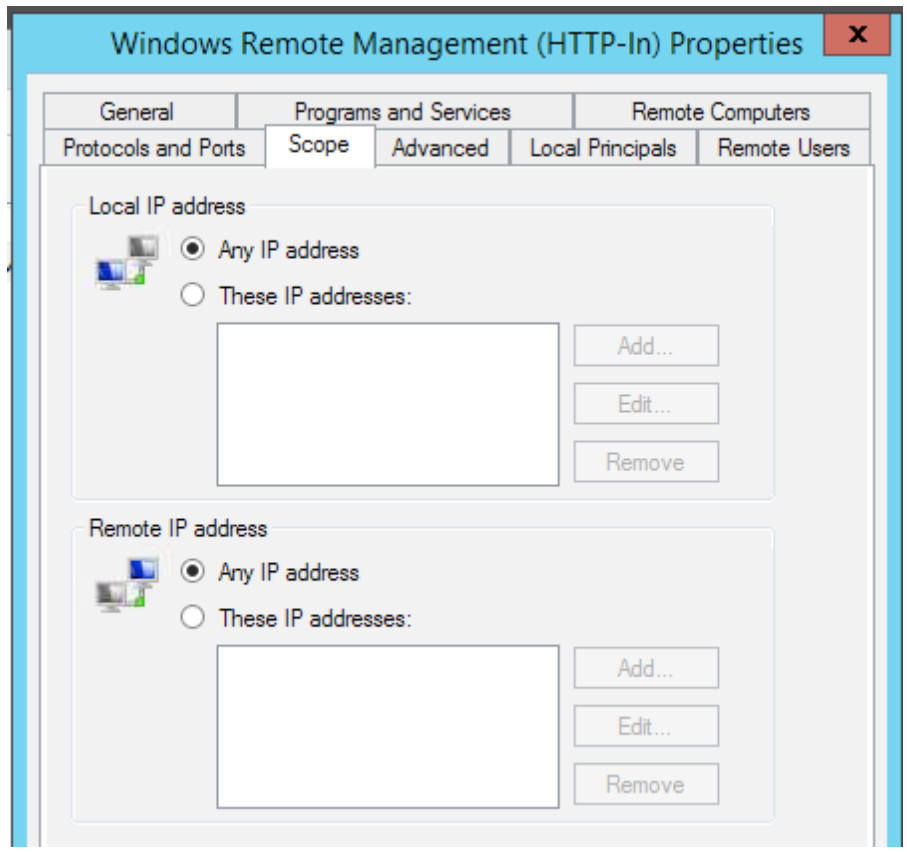
Standards, and the "Windows Logging Cheat Sheet". Improving local audit logging of Windows systems is critical to Incident Responders, Forensics, and Threat Hunters as they need to investigate the details of an incident as quickly, and in as much detail, as possible saving organizations time and money, and improving the investigative effectiveness in the process.

Securing WinRM and ARTHIR:

Whenever a remote control or remote management solution is installed into an environment, it should be secured so that only authorized individuals and systems can use the solution. Remote management solutions should log its execution so it can be monitored, and most importantly, the bad actors do not misuse it. In order to secure WinRM, so only those needing to execute WinRM can use it, the Windows Firewall would be used to create a list of approved clients that WinRM calls can originate. The following image shows an example of Group Policy settings for WinRM.



Once the WinRM policy rule has been created, the local IP Addresses (the systems executing ARTHIR) that are allowed to perform remote management can be added to the "Local IP address" list as seen in the following image. Of course, systems performing remote management should be locked down, monitored, and alerted to any suspicious activity due to the power to remote access all the systems in an organization.



Monitoring WinRM and ARTHIR Activity:

If WinRM is going to be widely used for things like ARTHIR, then monitoring WinRM is strongly recommended. If using a log management, SIEM, or a tool like LOG-MD that harvests event log data, the following logs should be consumed:

On the target:

- Application
  - ★1337 (or whatever ID is assigned in the ARTHIR script)
- Security
  - ★Event ID 4624 (success)
  - ★Event ID 4688 - wsmprovhost.exe (PowerShell Remoting)
  - ★Event ID 5156 for destination port 5985 (http) or 5986 (https)
- Applications and Services – Microsoft-Windows-Windows Remote Management/Operational
  - ★Event ID 142, 161 – Shows failed authentication due to WinRM not being configured

★Event ID 91 – States a WSMan Shell was established with PowerShell

- Applications and Services – Microsoft-Windows-PowerShell/Operational

★4103 – Script Name (ARTHIR)

★4104 – Task Category "Execute a Remote Command"

- Windows PowerShell (older version v2)

★800 – Look for target name

- On the system launching WinRM or ARTHIR look for the following:

- Security

★Event IDs 4624 (success) and 4625 (failed) type 3 (network) login events

★Event ID 4648 with a Process Name of PowerShell

★Event ID 5156 for destination port 5985 (http) or 5986 (https)

- Applications and Services – Microsoft-Windows-Windows Remote Management/Operational

★Event ID 6 - Creating WSMan Session. The connection string is: http://<systemname>:5985/wsman?PSVersion=5.1.18362.628

- Applications and Services – Microsoft-Windows-PowerShell/Operational

★4103 – Script Name (ARTHIR)

★4104 – Task Category "Execute a Remote Command"

- Windows PowerShell (older version v2)

★500, 501

★800 – Look for target name

I hope this article provides ideas on how powerful ARTHIR is and can be for Windows environments. If you create, modify, or update any ARTHIR modules, by all means, upload them to the public Github

repo and contribute to the project for others to use and benefit. Help all of us catch the bad actors. Happy Hunting!

1. https://MalwareArchaeology.com
2. https://github.com/davehull/KANSA
3. https://LOG-MD.com
4. https://attack.mitre.org/
5. Dave Hull SECKC 2016 talk on KANSA

About the Author

Michael is a Malware Archaeologist, Blue Team defender, Incident Responder, logoholic, and Principal Incident Response consultant for NCC Group. Michael developed several Windows logging cheat sheets to help the security industry understand Windows logging, where to start, and what to look for; they are available at Malware Archaeology (1). Michael is a primary contributor to the Open Source project ARTHIR, and co-developer of LOG-MD, a freemium tool that audits the audit log settings, harvests and reports on malicious Windows log data and many locations for malicious artifacts. Michael also ran BSides Texas for six years with Cons in Austin, San Antonio, Dallas and Houston, and leader of the BSides Austin Conference.

pwnedOrNot - OSINT Tool for Finding Passwords of Comprosied Email Accounts

By Lohitya Pushkar (thewhiteh4t)

Querying one or two email addresses is fine but what if we need to check official email addresses of a complete organisation; that is not possible manually. To save time and effort, I created **pwnedOrNot**, an automated **OSINT** tool for finding critical information.

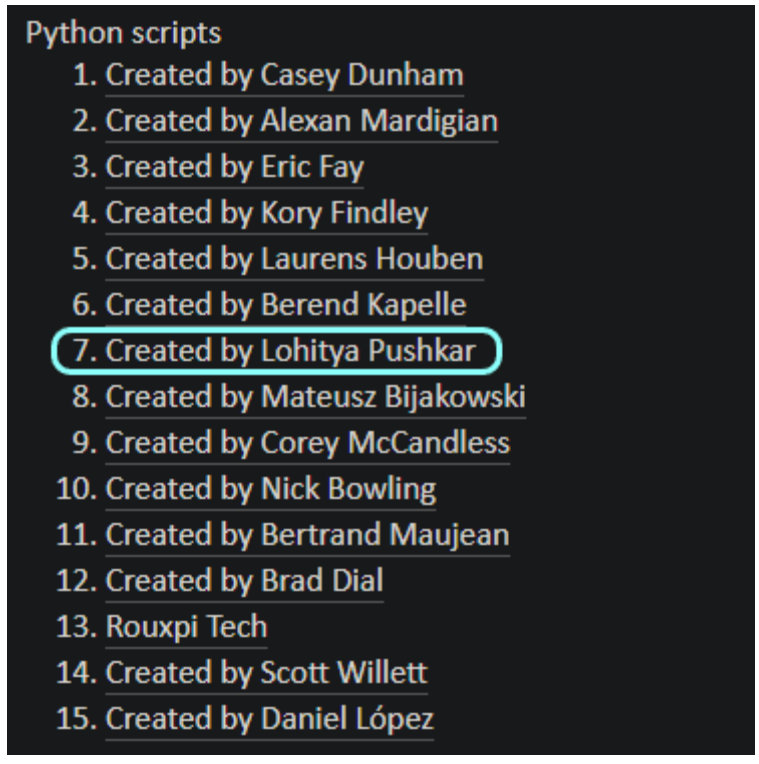
Introduction

One of the first things that comes to our mind when we talk about breaches and compromised email accounts is **Have I Been Pwned** by **Troy Hunt;** it is a free resource for anyone to quickly assess if they may have been put at risk due to an online account of theirs having been **compromised** or **"pwned"** in a data breach. Along with the website, HIBP also provides us with their **API,** which can be used to lookup accounts in a **fast** and **efficient** manner.

Breach Information

Querying one or two email addresses is fine but what if we need to check official email addresses of a complete organisation; that is not possible manually. To save time and effort, I created **pwnedOrNot**, an automated **OSINT** tool for finding critical information like:

- Name of Breach
- Domain Name
- Date of Breach
- Fabrication status
- Verification Status
- Retirement status
- Spam Status



HIBP API consumers

But that's just the information part of the tool, the main reason I created **pwnedOrNot** was to fetch information about the breach and then use that information to find public **password dumps** and locate the password for the requested email address. But if your internet safety practices were on point, there

is a possibility that your email is not **pwned**. But what if your credentials were compromised in a large scale organisation level breach such as:

- Yahoo - 2013,14 - 3 Billion, 500 Million Records
- First American - 2019 - 885 Million Records
- Facebook - 2019 - 540 Million Records
- Marriott - 2018 - 500 Million Records
- Friend Finder - 2016 - 412.2 Million Records

In breaches of this scale, there is a good chance that someone's, or an organisation's, credentials were compromised and initially these data dumps are available for sale on the dark web but eventually they show up on surface web under popular websites like **pastebin**.

But not every data dump lands into the surface web and so **pwnedOrNot** will at times show information about a breach but it will not show any passwords if a data dump is not available.

Practical Usage

pwnedOrNot can be beneficial for both **Blue Team** and **Red Team** assessments; in both cases a compromised email can be hazardous for an organisation of any scale. Attackers can do damage in multiple ways once they have access to a certain email account(s) such as:

- Along with access to the email account, attackers also gain access to the digital contact list attached to that email account; they can then scam other people in your contact list or send them malicious files to perform other nefarious activities.
- They can use the email account in mass email scams where a legitimate email account can be used to send a large number of scam emails or emails with malicious links and files.
- Many people find it hard to remember passwords and so they use the same password for multiple accounts and devices; attackers can try and infiltrate into many other accounts and devices and gain more critical data about the person or organisation.

- Attackers can also perform social engineering and gain access to more critical data inside an organisation by carefully crafting the content of the email and sending it to important people in an organisation.
- Recently, ransomware are the first choice when it comes to attacking an organisation and most of ransomware have been delivered through emails such as:
  - ★Recently, Maastricht University paid a ransom of 30 BTC to Russian hackers, origin of the ransomware attack was a phishing email
  - ★iNSYNQ was hit by a ransomware attack, the origin of the attack again was a phishing email
  - ★NHS cyber attack in which WannaCry ransomware was used, emails were the top delivery methods
  - ★A police department employee opened an email containing ransomware that affected the whole city of Riviera Beach and they spent $941,600 in recovery, which consists of hundreds of laptops and desktops.

Both Blue and Red teams can benefit from pwnedOrNot by testing bulk email addresses in a fast and automated way for possible breaches and publicly available passwords to avoid such damage caused by malicious actors.

Availability

pwnedOrNot is open source and available on GitHub. It has also been added to two Penetration Testing Distributions, BlackArch Linux and a new upcoming distribution SecBSD. It supports most linux distributions along with **Termux** and **Kali Nethunter** on **Android.** It has been tested on the following Linux distributions:

- Kali Linux
- BlackArch Linux
- Ubuntu
- Arch Linux

- Manjaro
- Kali Nethunter
- Termux

Installation

It is Python 3 based and uses most of the packages from the Standard Python Library, along with standard packages it requires a **requests** library, which can be installed from pip easily. Fewer and simple dependencies make installation very fast and simple and expands support to most Linux distributions.

These instructions can be used for any Linux distribution and I am assuming you have Python 3 already installed, if not then please install Python 3 and pip before following these instructions below:

Step 1

Clone the repo from github into a directory of your choice:

```
git clone https://github.com/thewhiteh4t/pwnedOrNot.git
```

Step 2

Install requests package using pip3 command:

```
pip3 install requests
```

Step 3

In case I update the script, you can update it on your end with one simple command. First change to the directory where you cloned pwnedOrNot in step 1 and then simply execute:

```
git pull
```

Installation on BlackArch Linux is even simpler as it's available in their tools repository:

```
pacman -S pwnedornot
```

Tool Usage

There is a help menu in pwnedOrNot that can be accessed by executing:

```
python3 pwnedornot.py -h
```

You will be presented with this menu:

```
usage: pwnedornot.py [-h] [-e EMAIL] [-f FILE] [-d DOMAIN] [-n] [-l]
           [-c CHECK]

optional arguments:
 -h, --help         show this help message and exit
 -e EMAIL, --email EMAIL   Email Address You Want to Test
 -f FILE, --file FILE    Load a File with Multiple Email Addresses
 -d DOMAIN, --domain DOMAIN Filter Results by Domain Name
 -n, --nodumps        Only Check Breach Info and Skip Password Dumps
 -l, --list         Get List of all pwned Domains
 -c CHECK, --check CHECK   Check if your Domain is pwned
```

We can test a single email or in bulk if we use **-f** command line argument to use a file containing bulk email addresses.

-d command line argument allows us to filter breaches by domain, for example, a certain email account is compromised in multiple breaches but we want to check results for a specific domain.

When **-n** command line argument is used, pwnedOrNot only requests breach information for a requested email address or for bulk email addresses and skips the search for passwords.

If you want only a list of all breached domains that HIBP has information about, **-l** command line argument is useful for this; it simply prints a list of all breached domains. Domains from this list can be used with -d if you want the exact name of the domain you want to filter.

Example:

```
                    ______     _  __     __
     ____ _    _____ ___ ____/ / __ \_____/ | / /___ / /_
  / __ \ | /| / / __ \/ _ \/ __ / / / / ___/ |/ / __ \/ __/
 / /_/ / |/ |/ / / / / __/ /_/ / /_/ / / / /| / /_/ / /_
 / .___/|__/|__/_/ /_/\___/\__,_/\____/_/ /_/ |_/\____/\__/
/_/

[>] Created by : thewhiteh4t
[>] Version    : 1.2.8

[-] API Key Not Found...

[+] Fetching List of Breached Domains...

[+] 000webhost.com
.
.
.
[+] 500px.com
.
.
.
[+] hltv.org
.
.
.
[+] patreon.com
.
.
.
[+] shein.com
.
.
.
[+] zomato.com

[+] Total : 408

[+] Completed in 0.8135755062103271 seconds.
```

So as you can see, a total of **408 domains** were returned by HIBP in **0.8 seconds.**

-c enables us to check if a domain was ever breached by any malicious actor, if a domain is breached it returns the name of breach, date on which it was pwned, number of email accounts pwned in that breach, fabrication status, verification status, spam status and the type of information it contains, such as email addresses, passwords, usernames, etc.

Example:

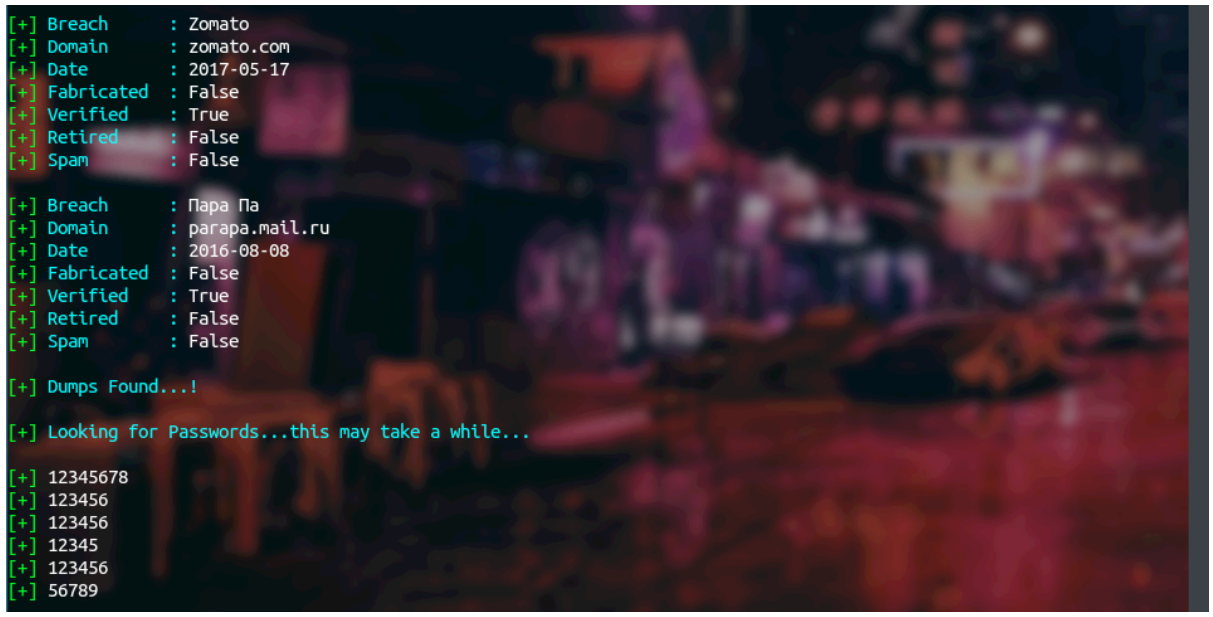
```
[+] Domain Name : adobe.com [ pwned ]

[+] Breach      : Adobe
[+] Domain      : adobe.com
[+] Date        : 2013-10-04
[+] Pwn Count   : 152445165
[+] Fabricated : False
[+] Verified    : True
[+] Retired     : False
[+] Spam        : False
[+] Data Types : ['Email addresses', 'Password hints',
'Passwords', 'Usernames']

[+] Completed in 0.6063365936279297 seconds.
```

Let's take a look at a few possible combinations of command line arguments we can use in pwnedOrNot:

```
# Check Single Email
python3 pwnedornot.py -e <email>
#OR
python3 pwnedornot.py --email <email>

# Check Multiple Emails from File
python3 pwnedornot.py -f <file name>
#OR
python3 pwnedornot.py --file <file name>

# Filter Result for a Domain Name [Ex : adobe.com]
python3 pwnedornot.py -e <email> -d <domain name>
#OR
python3 pwnedornot.py -f <file name> --domain <domain name>

# Get only Breach Info, Skip Password Dumps
python3 pwnedornot.py -e <email> -n
#OR
python3 pwnedornot.py -f <file name> --nodumps

# Get List of all Breached Domains
python3 pwnedornot.py -l
#OR
python3 pwnedornot.py --list

# Check if a Domain is Pwned
python3 pwnedornot.py -c <domain name>
#OR
python3 pwnedornot.py --check <domain name>
```

Now let's come check the fun part of pwnedOrNot, i.e. password hunting for a certain email address…

The email is hidden for obvious reasons. Here we can see two breaches and information about them such as domain and date, there can be a single or multiple breaches for a given email address or none if it's not compromised yet. After listing breaches, pwnedOrNot proceeds to find passwords in available dumps.



And it is done! pwnedOrNot found multiple passwords for the given email address, this can be slow at times so some patience is required.

- For this demo, I chose an email address that is compromised and dumps are available for it, there are multiple cases when it comes to results:
- An email address may not be compromised and so pwnedOrNot will show status as **not pwned.**
- An email address may not be compromised but a public dump is available for it. This happens when the breach is not known to HIBP and it's missing in their database and so pwnedOrNot will show status as **not pwned** but **dumps available**.
- An email address is compromised and both **breach information** and **dumps** are **available** for it. This is the best case we are looking for.
- An email address is compromised but no dumps are available for it. This is a common case and it means that a public dump is not available or it has been **deleted by pastebin.**

- An email address is compromised and both breach information and dumps are available but no password is returned. This happens when pwnedOrNot encounters an **email only** list, i.e. it does not contain any passwords.
- It is possible that multiple dumps are found and some of them have the same password in them and so pwnedOrNot may show the same passwords multiple times.

HIBP API Versions

To counter abusive API usage, HIBP upgraded APIv2 to APIv3 and the major change is the addition of API Authorisation. This solves the following issues:

- The rate limit could be applied to an API key thus solving the problem of abusive actors with multiple IP addresses
- Abuse associated to an IP, ASN, user agent string or country no longer has to impact other requests matching the same pattern
- The rate limit can be just that - a limit rather than also dishing out punishment via the 24 hour block

But a small drawback is that the API is no longer free and there is a US$3.50 per month fee. You can read the original article here: https://www.troyhunt.com/authentication-and-the-have-i-been-pwned-api/

So, when you run pwnedOrNot for the first time, it will ask for an API key which you can purchase from here : https://haveibeenpwned.com/API/Key

About the Author

- **Lohitya Pushkar (thewhiteh4t)**
- Cyber Security Researcher & Analyst
- I have created multiple open source tools for the infosec community and I am learning about Red Teaming, Network Penetration Testing, Exploit Analysis and OSINT.
- **Blog:** https://thewhiteh4t.github.io
- **GitHub:** https://github.com/thewhiteh4t
- **Twitter:** https://twitter.com/thewhiteh4t
- **LinkedIn:** https://www.linkedin.com/in/lohityapushkar

Velociraptor - Digging deeper

by Mike Cohen

Velociraptor was released in 2019. Similar to GRR, Velociraptor also allows for hunting across many thousands of machines. Inspired by OSQuery, Velociraptor implements a new language dubbed VQL (Velociraptor Query Language), which is similar to SQL but extends it in a more powerful way. Velociraptor also emphasizes ease of installation and very fast efficient operation and scalability.

Digital forensics is primarily focused on answering questions. Most practitioners limit their cases around high level questions, such as did the user access a particular file? Was malware run on the user's workstation? Did an attacker crack an account?

Over the years, DFIR practitioners have developed and refined methodologies for answering such questions. For example, by examining the timestamps stored in the NTFS filesystem we are able to build a timeline tracing an intruder's path through the network. These methodologies are often encoded informally in practitioners' experience and training. Wouldn't it be great to have a way to formally document and encode these methodologies?

In many digital evidence based cases, time is of the essence. The forensic practitioner is looking to answer questions quickly and efficiently, since the amount and size of digital evidence is increasing with every generation of new computing devices. We now see the emergence of triage techniques to quickly classify a machine as worthy of further forensic analysis. When triaging a system, the practitioner has to be surgical in their approach - examining specific artifacts before even acquiring the hard disk or memory.

Triaging is particularly prevalent in enterprise incident response. In this scenario it is rare for legal prosecution to take place, instead the enterprise is interested in quickly containing the incident and learning of possible impacts. As part of this analysis, the practitioner may need to triage many thousands of machines to find those machines that were compromised, avoiding the acquisition of bit-for-bit forensically sound images.

The rise of the endpoint DFIR agent

This transition from traditional forensic techniques to highly scalable distributed analysis has resulted in multiple offerings of endpoint agents. An agent is specialized software running on enterprise endpoints that provide forensic analysis and telemetry to central servers. This enabled detection of attackers from different endpoints as they traverse through the network and provides a more distributed detection coverage for more assets simultaneously.

One of the first notable endpoint agents was GRR, a Google internal project open sourced around 2012. GRR is an agent installed on many endpoints controlled by a central server. The agent is able to perform some low level forensic analysis by incorporating other open source tools such as the Sleuthkit and The Rekall Memory forensic suite. The GRR framework was one of the first to offer the concept of hunting - actively seeking forensic anomalies on many endpoints at the same time. For the first time, analysts could pose a question - such as "Which endpoints contain this registry key?", to thousands of endpoints at once, and receive an answer within minutes or hours.

Hunting is particularly useful for rapid triaging - we can focus our attention only on those machines that show potential signs of compromise. GRR also provides interactive remote access to the endpoint, allowing for user inspection of the endpoint.

As useful as GRR's approach was at the time, there were some shortfalls, mainly around lack of flexibility and limited scale. GRR features are built into the agent making it difficult to rapidly push new code updates or new capabilities in response to changing needs. It is also difficult to control the amount of data transferred from the endpoint, which often ends up being much more detailed than necessary, leading to performance issues on the server.

The next breakthrough in the field was the release of Facebook's OSQuery. This revolutionary tool allows one to query the endpoints using a SQL-like syntax query. By querying the endpoint, it is possible to adapt the results sent, apply arbitrary filtering and combine different modules in new creative ways. OSQuery's approach proved to be very flexible in the rapidly evolving stages of incident response, where users need to modify their queries rapidly and on short notice.

Introducing Velociraptor

Learning from these early projects, Velociraptor was released in 2019. Similar to GRR, Velociraptor also allows for hunting across many thousands of machines. Inspired by OSQuery, Velociraptor implements a new language dubbed VQL (Velociraptor Query Language), which is similar to SQL but extends it in a more powerful way. Velociraptor also emphasizes ease of installation and very fast efficient operation and scalability.

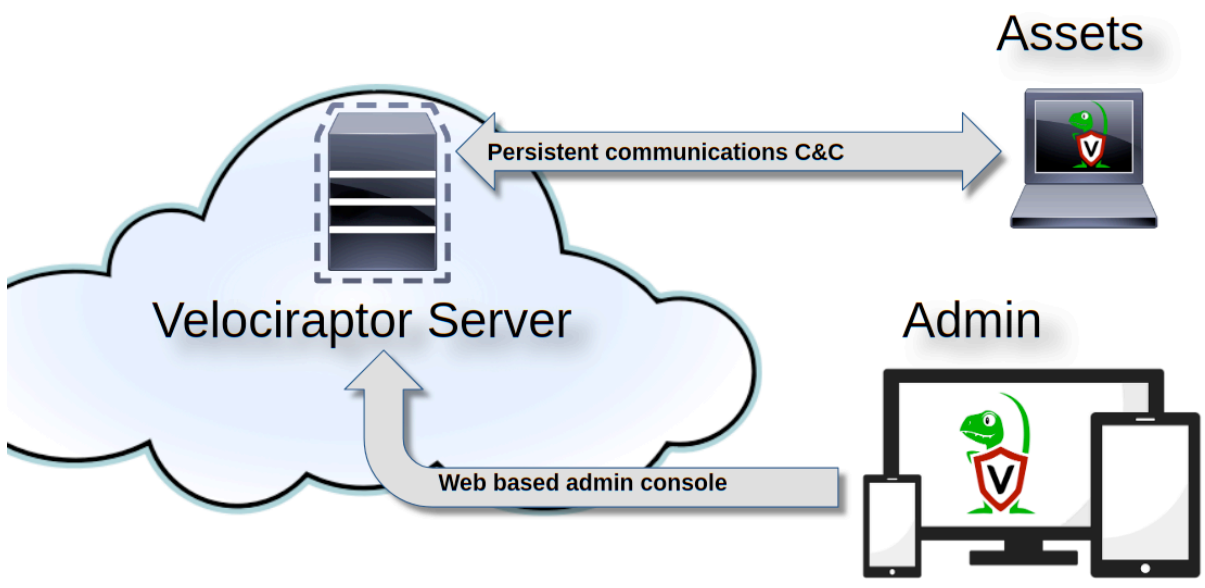


Fig.1

Figure 1 above shows an overview of the Velociraptor architecture. The Velociraptor server maintains communications with the endpoint agents (called Clients) for command and control. The web based administration user interface is used to task individual clients, run hunts and collect data.

Ultimately, Velociraptor agents are simply VQL engines - all tasks to the agent are simply seen as VQL queries that the engine executes. VQL queries, just like database queries, result in a table, with columns (as dictated by the query) and multiple rows. The agent will execute the query, and send back the results to the server, which simply stores them as files. This approach means the server is not really

processing the results other than just storing them in files. Therefore, the load on the server is minimal, allowing for vastly scalable performance.

Velociraptor artifacts

Writing free-form queries is a powerful tool, but from a user experience perspective, it is not ideal. Users will need to remember potentially complex queries. Velociraptor solves this by implementing "**Artifacts**". An artifact is a text file written in YAML that encapsulates the VQL, adds some human readable descriptions and provides some parameters allowing users to vary the operation of the artifact to some extent.

As an example of this process, we consider the Windows Scheduled Tasks. These tasks are often added by attackers as a way of gaining persistence and a backdoor to a compromised system (See Att&ck Matrix T1053). Velociraptor can collect and analyse these tasks if provided with the appropriate VQL query. By writing the query into an artifact, we make it possible for other users to simply re-use our VQL.

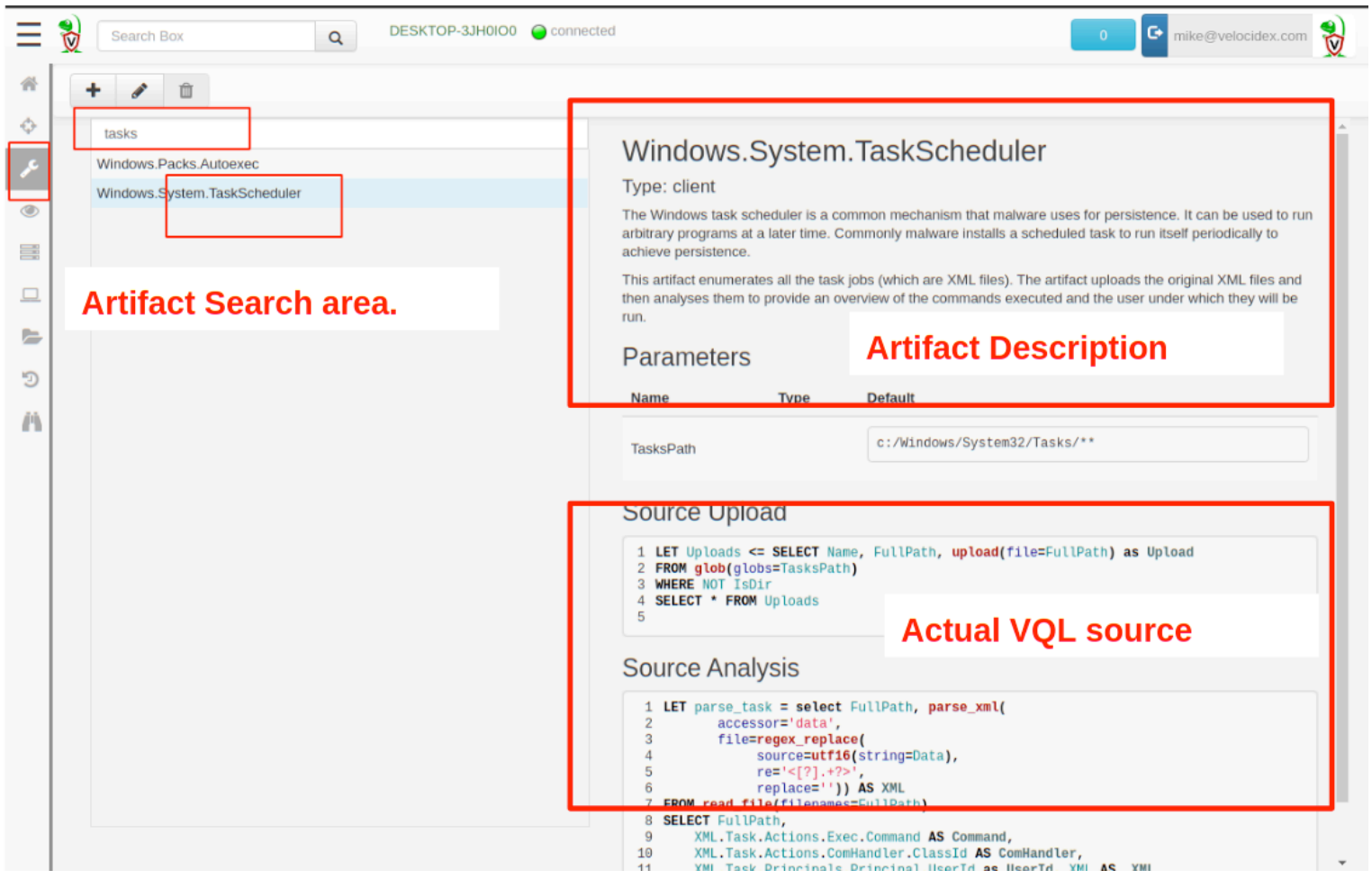


Fig. 2

Figure 2 shows the Windows.System.TaskScheduler artifact as viewed in the GUI. The artifact contains some user readable background information, parameters and the VQL source. As **Figure 3** below shows, in the GUI, one simply needs to search for the scheduled tasks artifact, select it and collect it from the endpoint.

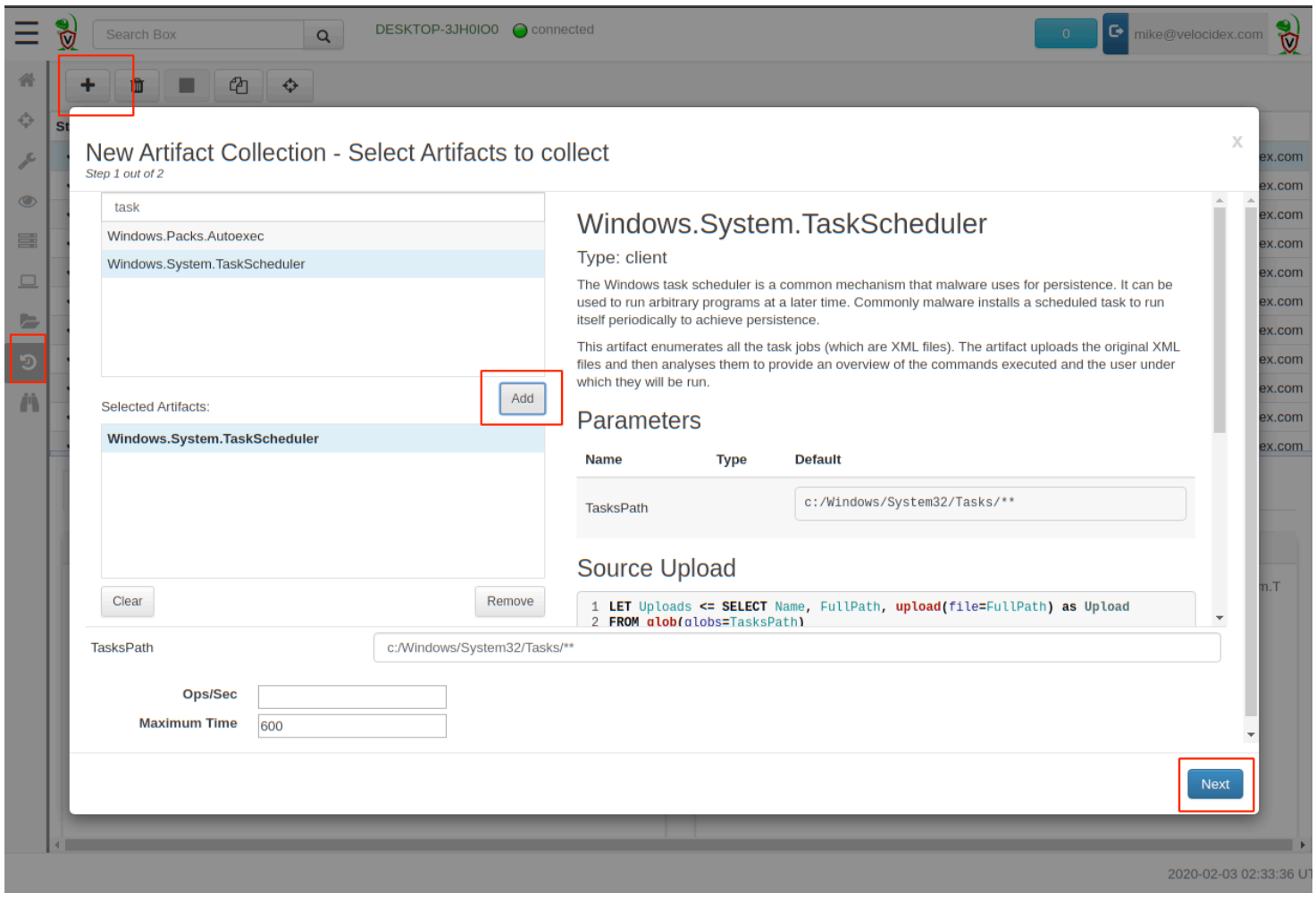


Fig.3

As we the soon as issue collection request, the client will run the VQL query, and send the result to the server within seconds. If the agent is not online at the time of the query, the task will be queued on the server until the endpoint comes back online, at which time the artifact will be collected immediately.

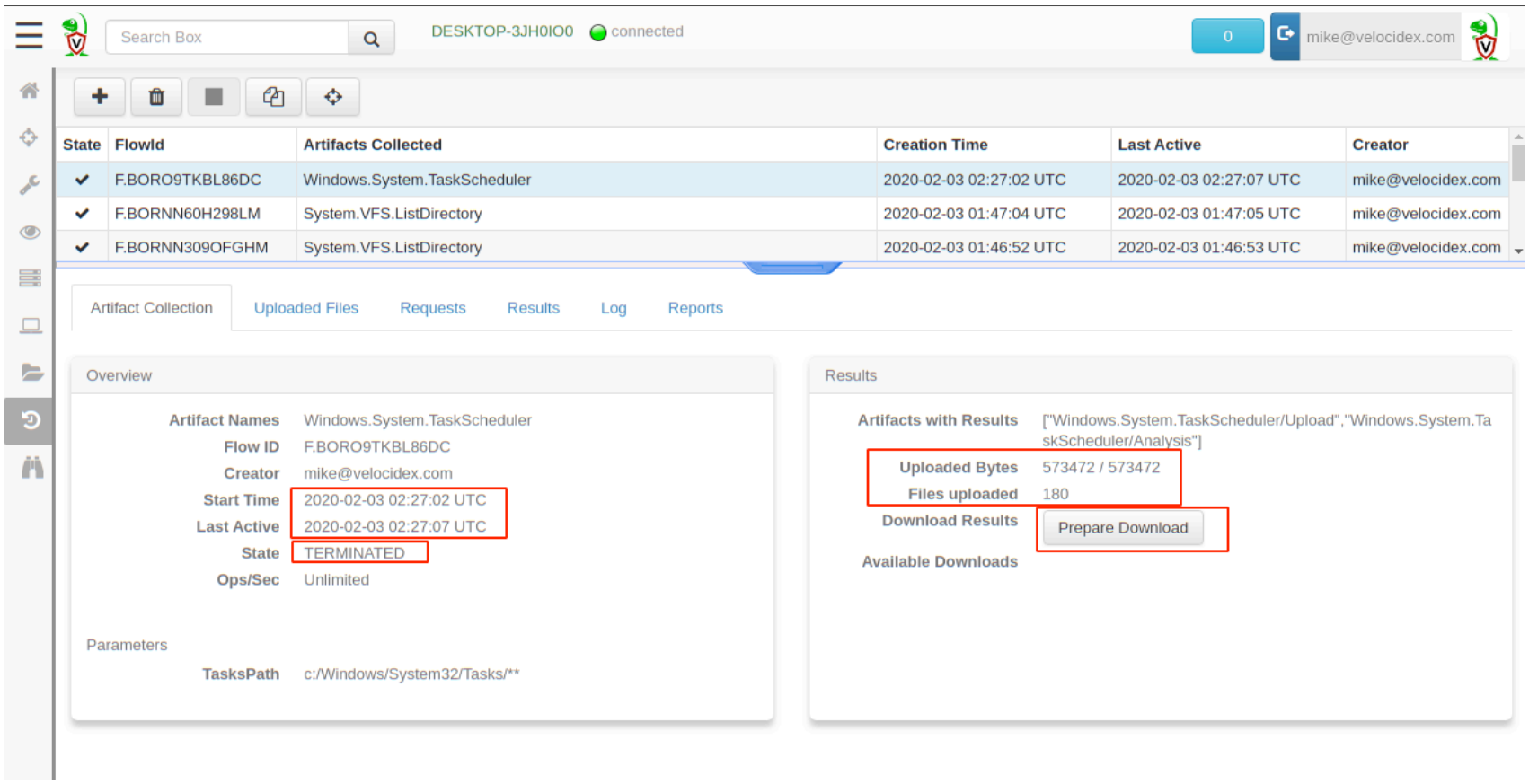


Fig.4

Figure 4 shows the result of this collection. We see the agent took 5 seconds to upload the 180 scheduled task XML files, which took a total of 5.7mb. We can click the "**Prepare Download**" button now to prepare a zip file containing these files for export. We can then download the zip file through the GUI and store it as evidence as required.

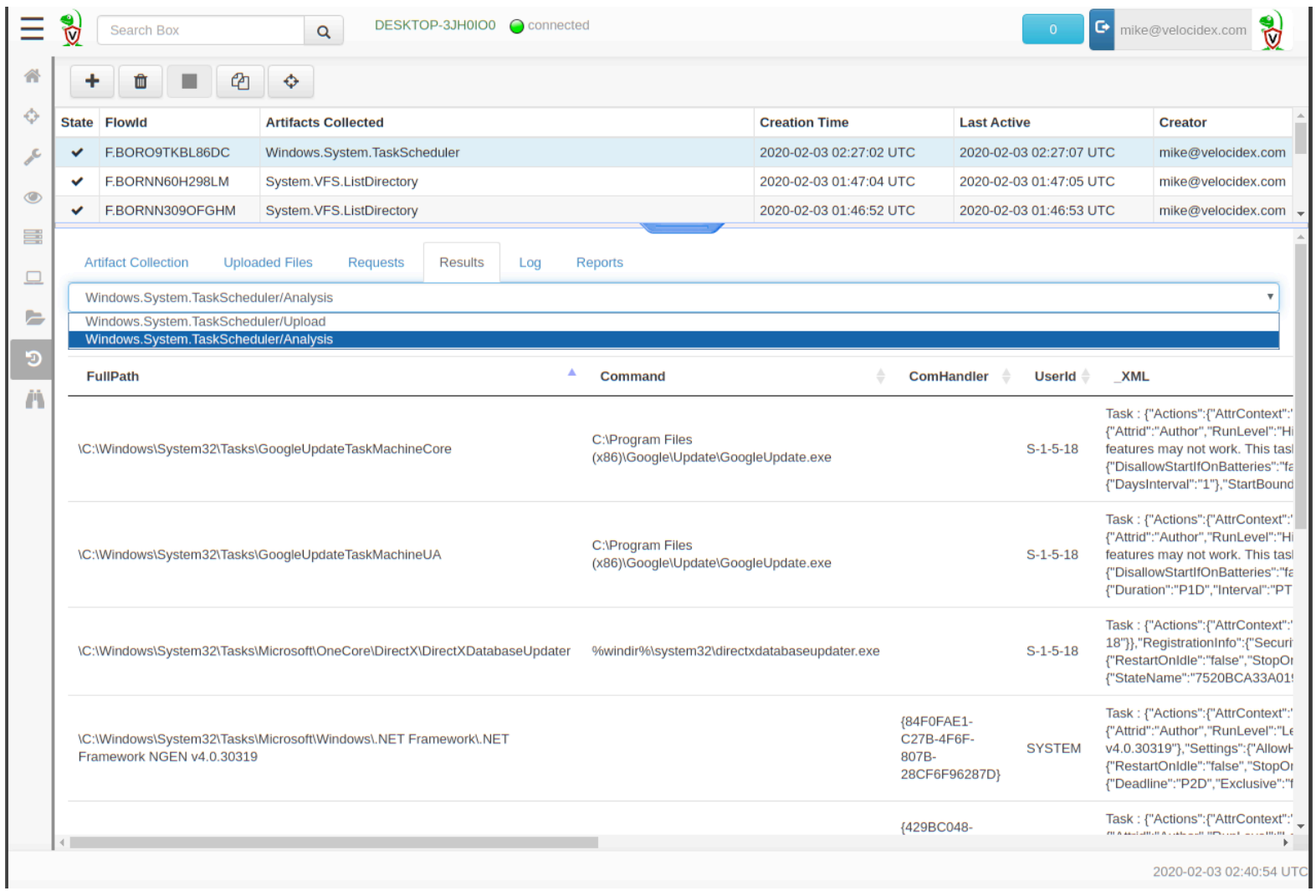


Figure 5 shows the results from this artifact. The VQL query also instructed the endpoint to parse the XML files on the endpoint and extract the launched command directly. It is now possible to quickly triage all the scheduled tasks looking for unusual or suspicious tasks. The exported zip file will also contain the CSV files produced by this analysis and can be processed using any tool that supports CSV formatted data (e.g. Excel, MySQL or Elastic through Logstash).

Hunting with Velociraptor

Continuing our example of scheduled tasks, we now wish to hunt for these across the entire enterprise. This captures the state of the deployment at a point in time when the hunt was collected and allows us to go back and see which new scheduled tasks appeared at a later point in time.

Hunting is simply a way to collect the same artifact from many machines at the same time. The GUI simply packages the results from these collections into a single exported file.

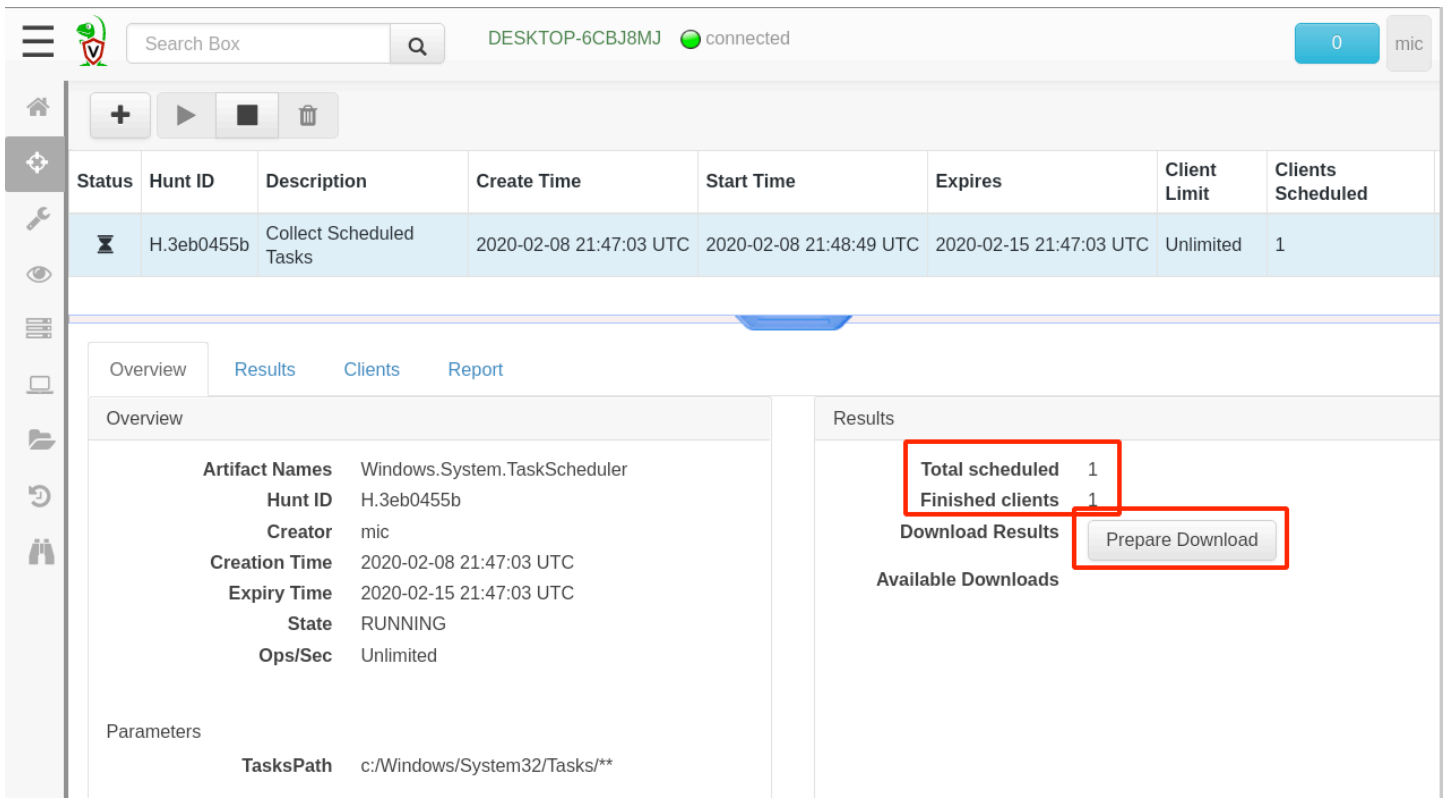


Fig. 6

Figure 6 shows a hunt created to collect Windows.System.TaskScheduler artifacts. We can see the total number of clients scheduled and completed and that the hunt will expire in one week. If new machines appear within this time, they will also have that artifact collected. Finally, we can prepare an export zip file for download that contains all the client's collected artifacts.

Forensic analysis on the endpoint

To be really effective, Velociraptor implements many forensic capabilities directly on the endpoint. This allows for writing artifacts that can leverage this analysis, either in a surgical way - identifying directly the relevant data - or in order to enrich the results by automatically providing more context to the analyst. In this section, we examine some of these common use cases and see how they can be leveraged through use of artifacts.

Searching for files

A common task for analysts is to search for particular filenames. For example, in a drive by download or phishing email case, we already know in advance the name of the dropped file and we simply want to know if the file exists on any of our endpoints.

The **Windows.Search.FileFinder** artifact is designed to search for various files by filename. **Figure 7** below illustrates the parameters that can be used to customize the collection. For a typical drive by download, we might want to search for all binaries downloaded recently within the user's home directories. We can also collect matching files centrally to further analyse those binaries. The artifact also allows us to filter by keywords appearing within file contents.

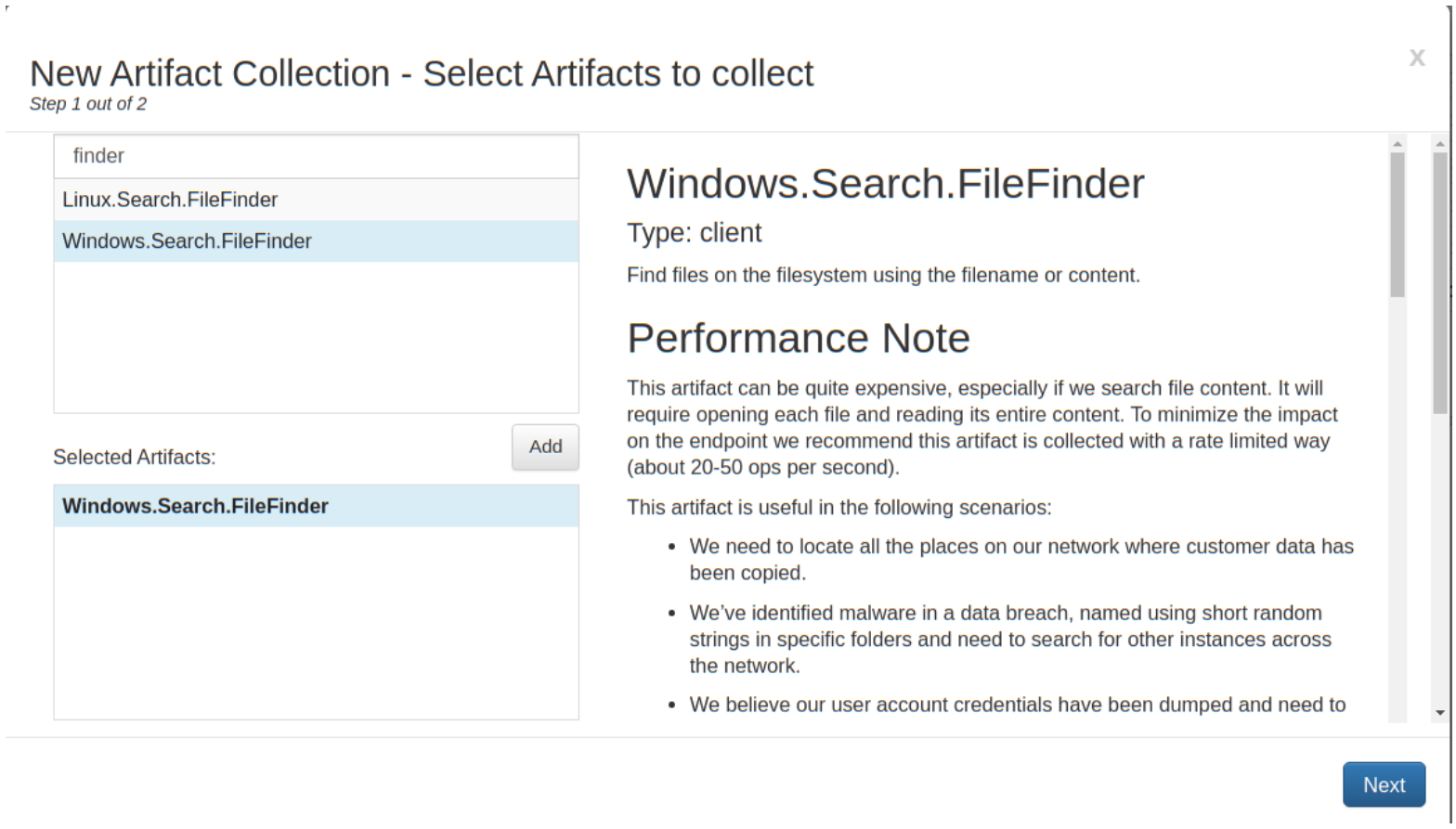


Fig. 7

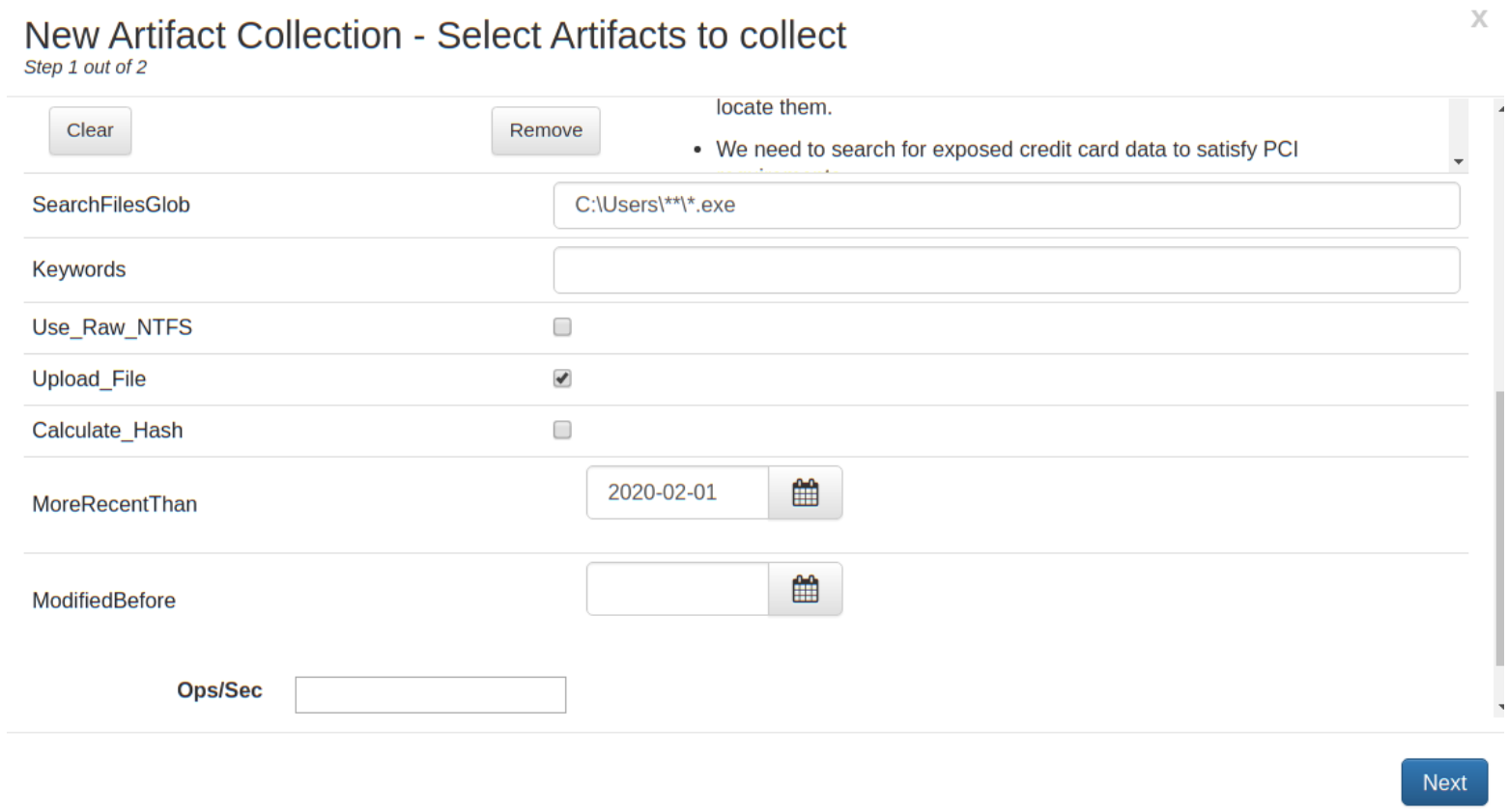


Fig. 8

Searching for files is a very common operation, which covers many of the common use cases, but it is limited to finding files that are not currently deleted. Velociraptor also includes a complete NTFS filesystem parser available through a VQL query. This allows us to extract low level information from every MFT entry.

Fig. 9

Figure 9 shows a sample of this output. We can see details like the **FILE_NAME** timestamps, as well as the **STANDARD_INFORMATION** stream timestamps (useful for detecting time stomping).

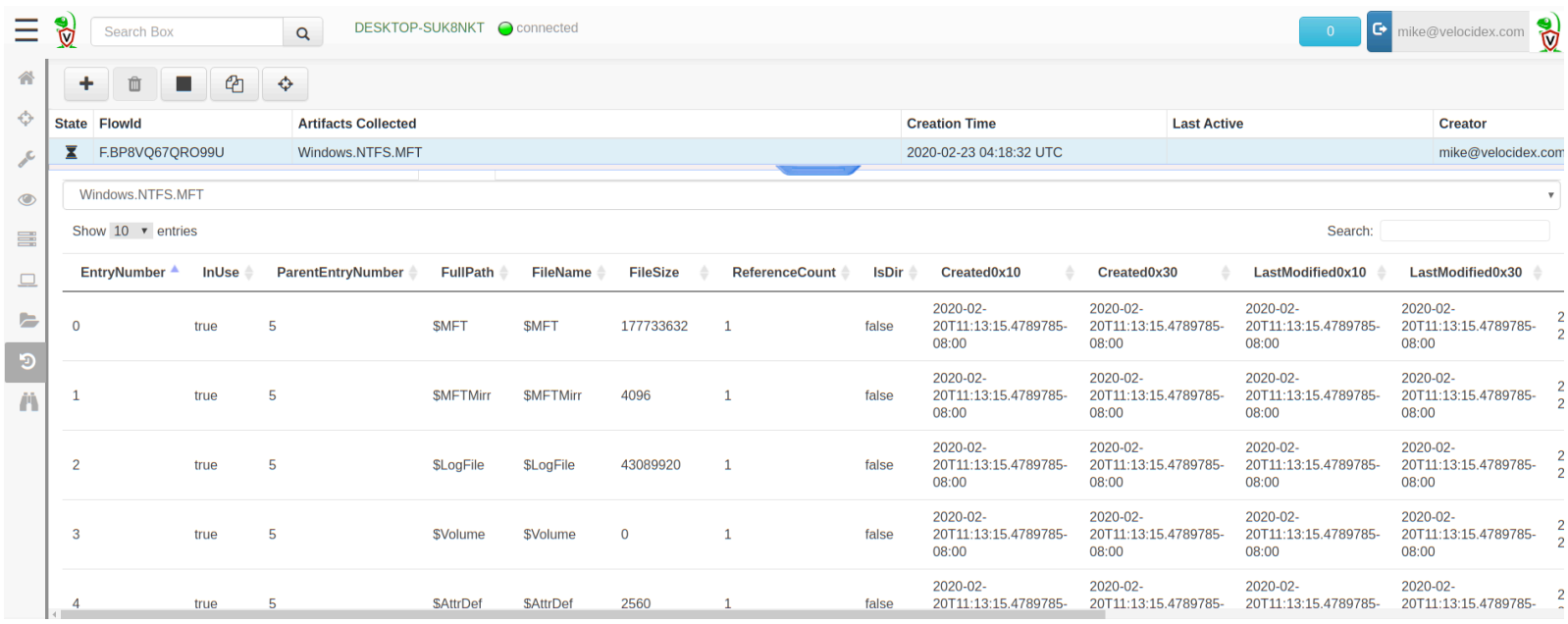


Fig. 10

While the Windows.NTFS.MFT artifact dumps all MFT entries from the endpoint, we can make this more surgical and specifically search for deleted executables. To do this we would need to modify the VQL query to add an additional filter.

Modifying or customizing an artifact is easy to do through the GUI. Simply search for the artifact in the "**View Artifacts**" screen, and then click the "Modify Artifact" button to bring up an editor allowing the YAML to be directly edited (note that all customized artifacts, automatically receive the prefix "**Custom**" in their name setting them apart from curated artifacts).

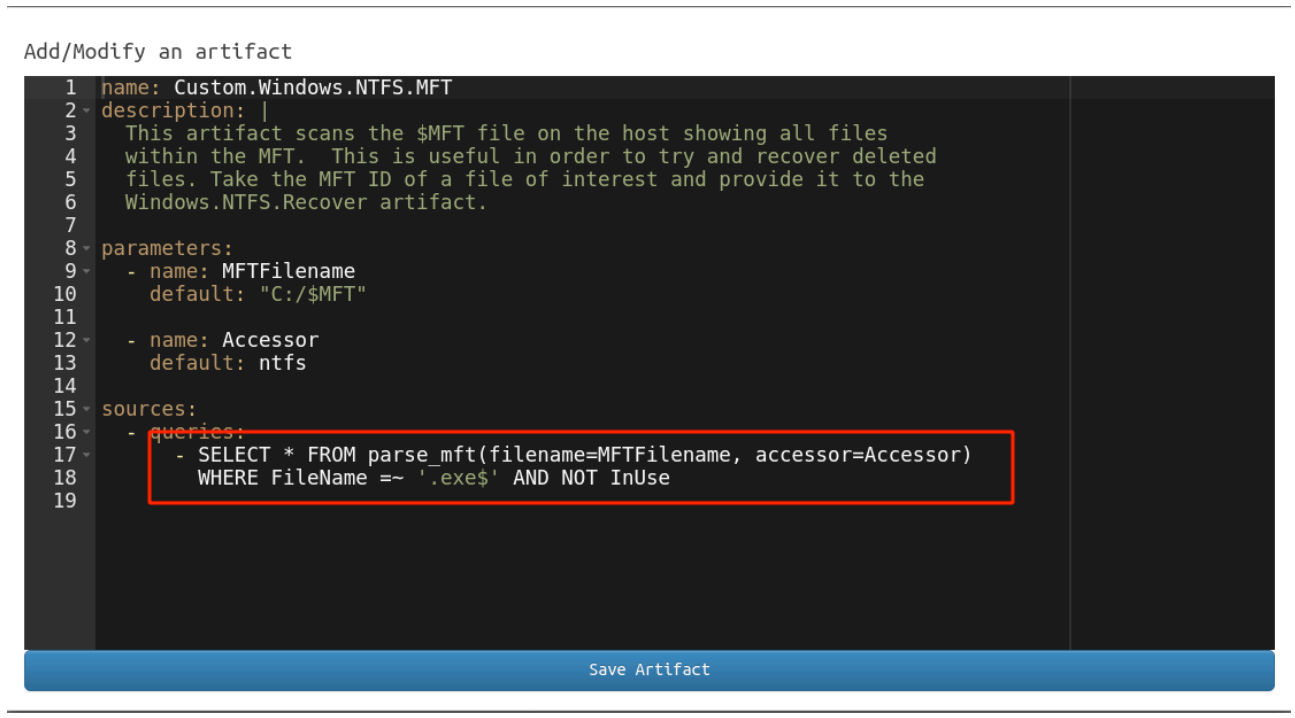


Fig. 11

In the figure above, we added the condition **"WHERE FileName =~ '.exe$' AND NOT InUse"** to restrict output only to deleted executables. We now select this customized version and collect it on the endpoint as before. Since we have filtered only those executables deleted in this query, the result set is much smaller and somewhat quicker to calculate. **Figure 12** below shows a single binary was found on our test system still recoverable in unused MFT entry.

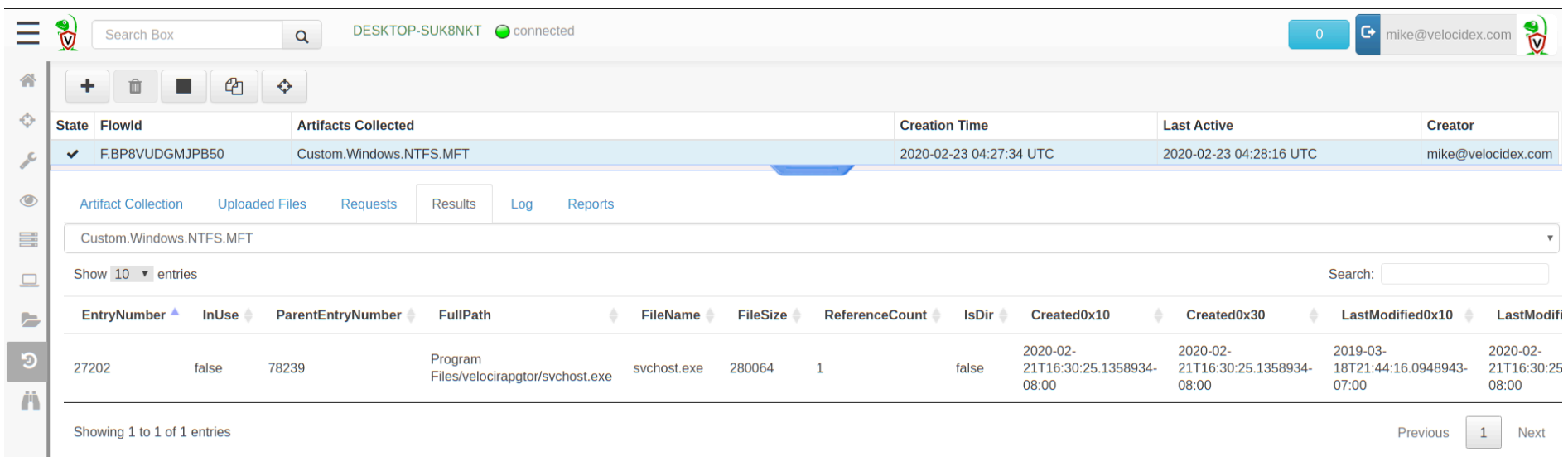


Fig. 12

Figure 12 shows an MFT entry for a binary that had been removed from disk. If we are lucky, we can attempt to recover the deleted file using the **Windows.NTFS.Recover** artifact. This artifact simply dumps out all the attribute streams from the specified MFT entry (including the **$DATA** attribute) and uploads them to the server. **Figure 13** below shows how we can select to collect this artifact, and specify the MFT entry reported in the previous collection as a parameter to the artifact.

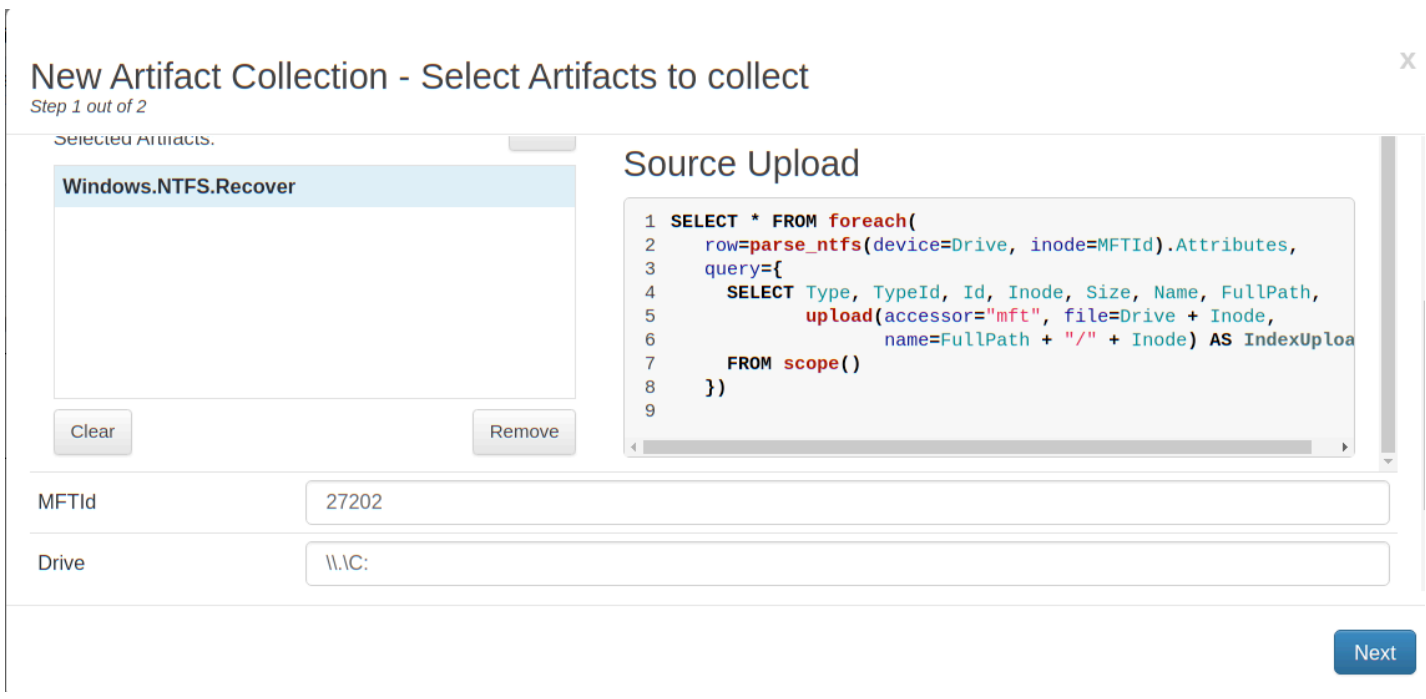


Fig. 13

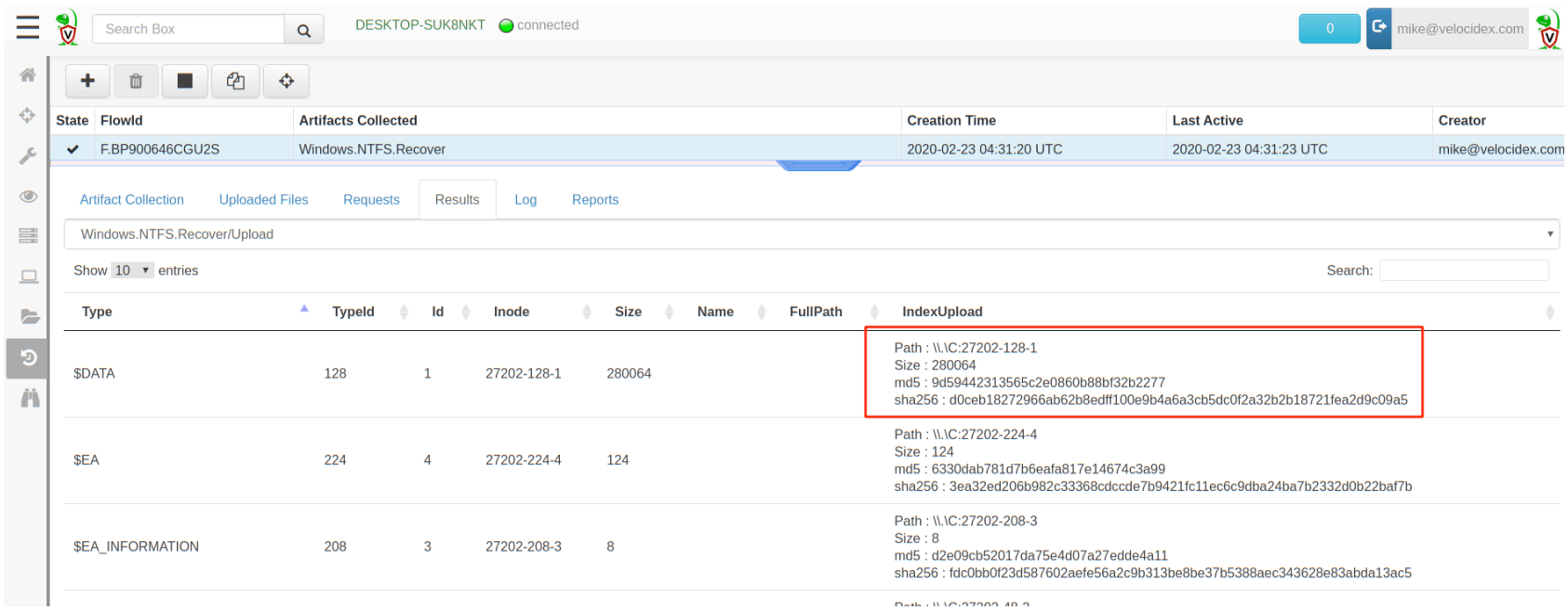


Fig. 14

Figure 14 shows the output from the **Windows.NTFS.Recover** artifact, showing the **$DATA** stream was correctly recovered as verified by its hash.

The previous example demonstrates how having advanced forensic analysis capabilities is valuable during endpoint monitoring. The example of a drive by download required us to confirm if a particular executable is present on any of our endpoints. We started off by performing a simple filename search for executables. But realizing this will only yield currently existing files, we move onto deep level NTFS analysis dumping all MFT entry information. We then modified the VQL query to restrict the output to only the subset of results of interest in our case.

This modified query can now run as a hunt on the entire fleet to determine which executables have recently been deleted anywhere, which would confirm if the malware was run on other machines we are not aware of. We can then potentially use NTFS recovery techniques to recover the binary for further analysis. Without the flexibility of the powerful Velociraptor Query Language, it would be difficult to adapt to such a fluid and rapidly developing incident.

Conclusions

Velociraptor includes many other low level analysis modules, such as parsing prefetch files, raw registry access (for AMCache analysis), ESE database parser (facilitating SRUM database forensics and Internet Explorer history analysis), SQLite parsers (for Chrome and Firefox history) and much more.

The true power of Velociraptor is in combining these low level modules with other VQL queries to further enrich the output or narrow down queries making them more surgical and reducing the amount of false positives. This more targeted approach is critical when hunting at scale in order to reduce the amount of data collected and assist the operator focus on the truly important evidence quickly and efficiently.

The type of analysis performed is driven by a flexible VQL query, written into an artifact by the user. This unprecedented level of flexibility and scale in a forensic tool allows for flexible and novel response and collection. It is really only limited by the imagination of the user.

We opened this article by imagining a world where experienced forensic practitioners could transfer and encode their knowledge and experience into actionable artifacts. Velociraptor's artifacts help to bring this vision to life - allowing experienced users to encode their workflow in VQL artifacts opens these techniques up to be used by other practitioners in a more consistent and automated fashion. We hope to inspire a vibrant community of VQL Artifact authors to facilitate exchange of experience, techniques and approaches between practitioners and researchers alike.

Velociraptor is available under an open source license on GitHub. You can download the latest Velociraptor release and use it immediately, or clone the source repository and contribute to the project. You can also contribute VQL snippets or artifacts directly to the project in order to share commonly used artifacts with the larger community.

About the author:

Dr. Mike Cohen is a renowned digital forensic researcher and senior software engineer. He has supported leading open-source DFIR projects including as a core developer of Volatility and lead developer of both Rekall and Google's Grr Rapid Response. Mike founded Velocidex in 2018 after working at Google for the previous 8 years in developing DFIR tools. Velocidex is the company behind the Velociraptor open source endpoint visibility tool.

HookCase, An Open Source Tool for Reverse Engineering macOS and its Applications

by Steven Michaud

HookCase (https://github.com/steven-michaud/HookCase) is an open-source tool for reverse engineering and debugging macOS (aka OS X), and the applications that run on it. It re-implements and extends Apple's DYLD_INSERT_LIBRARIES functionality, while avoiding all of Apple's restrictions. It can be used to hook any function in almost any module. I'm the author and maintainer of this tool, and in the following, I'll show how I used HookCase to resolve a particularly difficult bug in Mozilla's Firefox browser, which turned out to be caused by a macOS bug.

Introduction

Software engineers write code. But many of us actually spend most of our time debugging code written by others, running in environments created by others. To do this successfully, you need to know how that code works, and how it interacts with its environment. You normally start by reading the source code, if you have it. But beyond a certain level of complexity, you need to see the code running to understand it properly. If you're working with a self-contained application, you can run it in a debugger, like lldb or gdb. But this imposes serious constraints on your application. You need to stop and restart it, which can throw off the timing of its interactions with code that isn't running in your debugger. This is especially troublesome if your application has multiple processes, as many now do. You gain more flexibility by adding logging to the application's source code — if you have the source code and are able to compile it. But sometimes you don't have all the source code, or any of it — especially if the

code you're trying to understand belongs to the operating system. The next step is to inject "hooks" into a process for one or more of its functions. This approach is very flexible. Each hook can log whatever it wants, and (sometimes) even alter the behavior of the function that it hooks.

Many operating systems have built-in support for function hooking. For example, macOS, Unix and (to some extent) Linux have support for dtrace. But dtrace has "probes" for only a limited set of functions, and no probe can alter its function's behavior. Another step up is the LD_PRELOAD (Unix and Linux) or DYLD_INSERT_LIBRARIES (macOS) environment variables. These support hooks that can alter the behavior of their original functions. But the only functions that can be hooked in this way are those whose code is dynamically linked from one module (say a dynamically loaded library) into another (say a process's main executable). Furthermore, Apple and some other application vendors (notably Google) have blocked or disabled the use of DYLD_INSERT_LIBRARIES on their macOS applications (for example, Safari and Chrome).

HookCase (https://github.com/steven-michaud/HookCase) is an open-source tool for reverse engineering and debugging macOS (aka OS X), and the applications that run on it. It re-implements and extends Apple's DYLD_INSERT_LIBRARIES functionality, while avoiding all of Apple's restrictions. It can be used to hook any function in almost any module. I'm the author and maintainer of this tool, and in the following, I'll show how I used HookCase to resolve a particularly difficult bug in Mozilla's Firefox browser, which turned out to be caused by a macOS bug.

Using HookCase To Diagnose and Work Around a macOS Bug

Bug 1201401 in the Mozilla bug database (https://bugzilla.mozilla.org/show_bug.cgi?id=1201401) was opened almost five years ago. The bug reported there caused a large number of crashes. Even though several people described how they could trigger it on their systems, no consistently reliable steps to reproduce it were ever established. It didn't seem like there was anything we could do about it, and so the bug languished. At the time, I worked for Mozilla, and was one of the people desperately searching for clues. But my efforts failed, and I gave up.

Shortly afterwards, I retired from Mozilla and turned my mind to other things. But I remained interested in the business of fixing bugs, and eventually wrote the tool that I would most liked to have had while I worked for Mozilla — HookCase. For the first couple of years, most of my HookCase time was spent improving it and fixing bugs in it. But more recently, I've been looking for ways to show off what it can do. When the umpteenth "me too" comment showed up at bug 1201401, I decided it was worth taking another look.

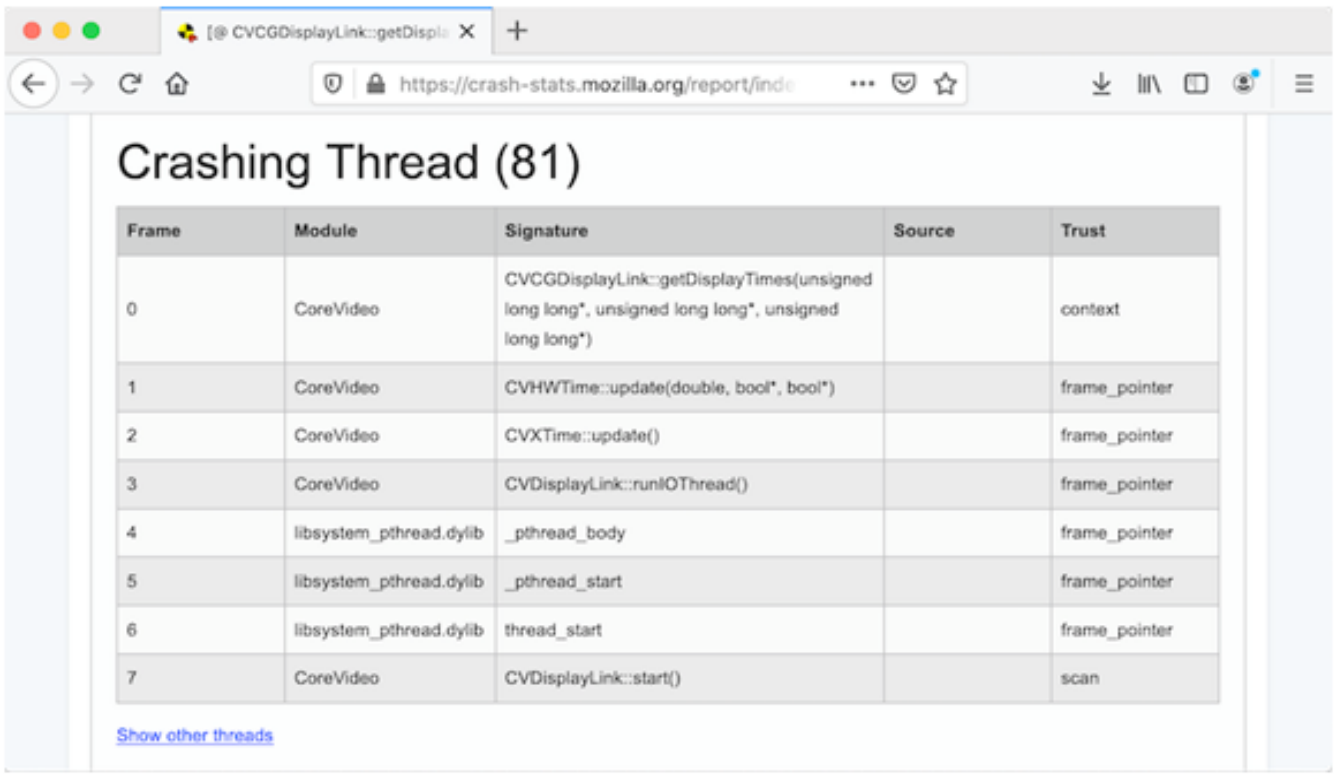


Figure 1: Bug 1201401 Crash Stack

From the start, it seemed likely to be a bug in the macOS operating system: The crashes occurred deep in system code. But without adequate tools, macOS operating system code is a closed book. Source code is unavailable for most of its binaries. The only way to "read" them is in a disassembler. And without a way to insert hooks wherever you like, you can't confirm or refute the hunches you make as to how the machine code works (or doesn't).

HookCase has changed all that, though you still have to pore over system code in a disassembler. The one I use is Hopper Disassembler (https://www.hopperapp.com/). It's not open source, but it has many convenient features. I find it indispensable. I will often quote assembly code from one or more macOS system binaries. This tends to be specific to a particular version of macOS. The version I use in my examples is macOS 10.5.3 build 19D76.

The crashes happened in the CoreVideo framework, in the CVCGDisplayLink::getDisplayTimes() function. So to start with, I looked at the location in Mozilla source code where the "CVDisplayLink" functions are used.

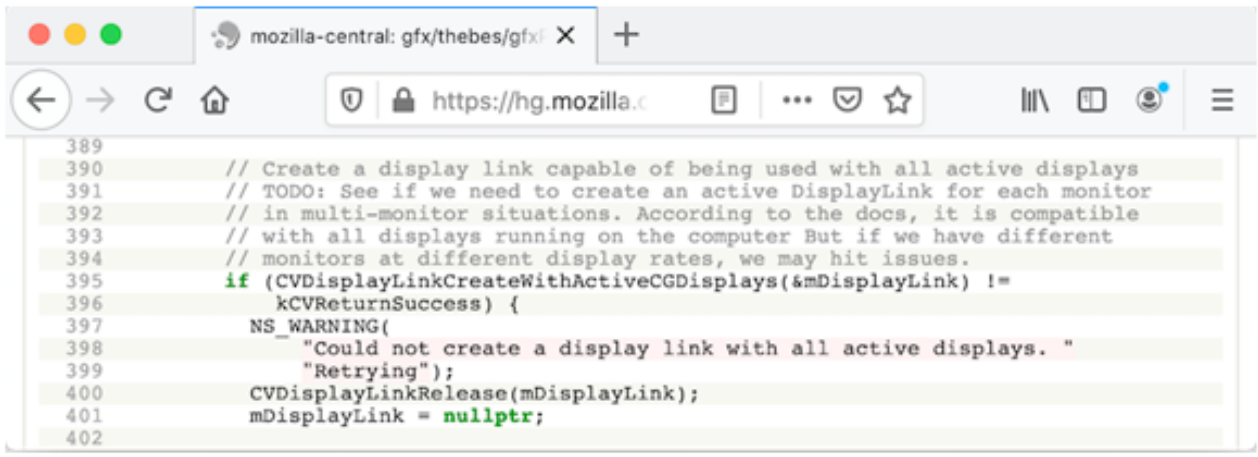


Figure 2: Where CVDisplayLink Functions Are Used In Mozilla Source Code (https://hg.mozilla.org/mozilla-central/file/3c1b35be6e660ac223cf76858530915a60ee6d67/gfx/thebes/gfxPlatformMac.cpp#l390)

I also loaded the CoreVideo framework binary (/System/Library/Frameworks/CoreVideo.framework/CoreVideo) into Hopper Disassembler and looked for code labels matching "CVDisplayLink" and "CVCGDisplayLink". After some trial and error, I found that hooking the following functions is a good way to track the life cycle of a CVDisplayLinkRef object. I included CVCGDisplayLink::getDisplayTimes() because that's where the crashes happened.

```
CVDisplayLinkStart();

CVDisplayLinkStop();

CVDisplayLinkRelease();

CVDisplayLinkCreateWithCGDisplays();

CVDisplayLinkSetOutputCallback();

CVDisplayLink::stop();

CVCGDisplayLink::getDisplayTimes();
```

To hook these functions in HookCase, first you need to write a hook library. The easiest way to do this is to start from HookCase's hook library template (https://github.com/steven-michaud/HookCase/tree/master/HookLibraryTemplate). It contains several different kinds of example hooks (commented out). Based on them, here's a hook for CVDisplayLinkStart():

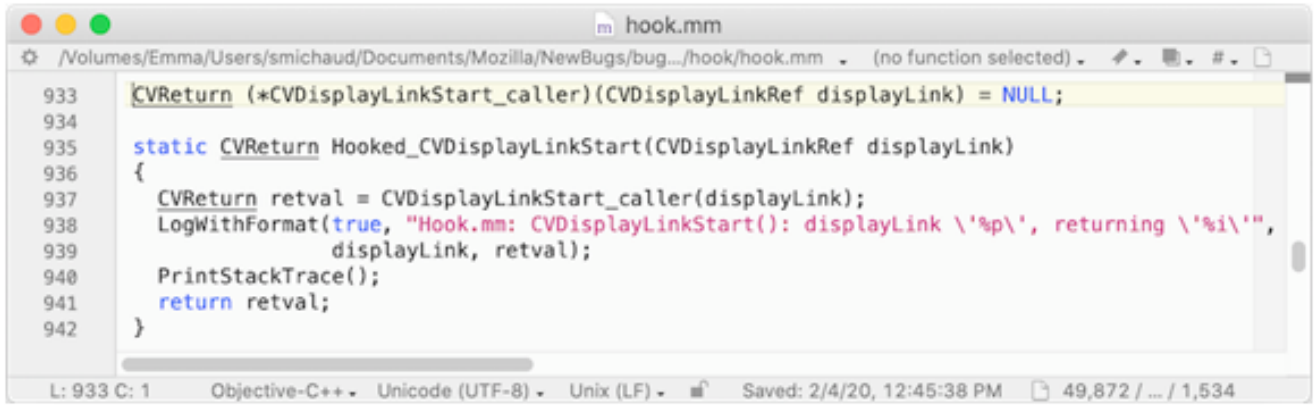


Figure 3: Hook for CVDisplayLinkStart()

To register this hook with the HookCase kernel extension, you need to add a line for it to the user_hooks[] structure:

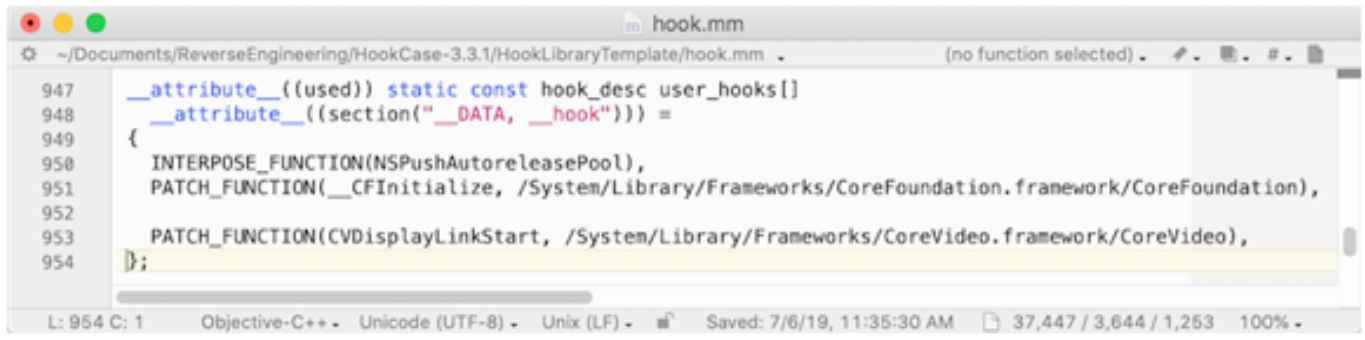


Figure 4: Register CVDisplayLinkStart() Hook In user_hooks[]

HookCase supports two different kinds of hooks — interpose hooks and patch hooks. An interpose hook only catches calls to a function from other modules. A patch hook catches all calls, including those from the same module. There can be circumstances when an interpose hook is preferable. But generally speaking, it's best to use a patch hook, to ensure that it will catch all calls. Patch hooks use the PATCH_FUNCTION() macro.

Build the hook library using make from a Terminal prompt. Then insert it into a process from the command line, as follows:

```
HC_INSERT_LIBRARY=/full/path/to/hook.dylib /path/to/application/binary
```

The full source code for my bug 1201401 hook library is available here https://github.com/steven-michaud/HookCase/tree/master/Examples/bugzilla1201401/hook-library.

The output for a single call to CVDisplayLinkStart() looks like this:

Figure 5: Logging Output for a Single Call to CVDisplayLinkStart()

Fuller logging output for Firefox Nightly, Safari and Google Chrome is available here https://github.com/steven-michaud/HookCase/tree/master/Examples/bugzilla1201401/output.

From my hook library's logging output, you can see that the life cycle of a CVDisplayLinkRef object is roughly as follows:

1) The CVDisplayLinkRef is created and initialized, on the main thread, by a call to CVDisplayLinkCreateWithCGDisplays().

2) CVDisplayLinkSetOutputCallback() is called, on the main thread, to register the callback in which most of the actual display work takes place.

3) CVDisplayStart() is called on the main thread. The first time it's called it creates and starts the special secondary thread on which calls to the callback take place.

4) The CVDisplayLink thread routine (CVDisplayLink::runIOThread()) loops many times on its dedicated thread, and on each iteration it calls CVCGDisplayLink::getDisplayTimes() and the callback.

5) CVDisplayLinkStop() is called on the main thread, which stops calls to the callback that was registered by CVDisplayLinkSetOutputCallback().

6) The CVDisplayLinkRef object is released (by a call to CVDisplayLinkRelease()) and destroyed, on the main thread.

This life cycle happens much more quickly in Firefox than in Safari or Chrome: CVDisplayLinkRefs are created and destroyed much faster and more often. More on this later.

In the assembly code for CVCGDisplayLink::getDisplayTimes(), bug 1201401's crashes happened at the following line (at offset 0x3A on macOS 10.15.3):

```
mov rax, qword [rdi+0x578]
```

This code dereferences a pointer-to-qword at offset 0x578 in a CVDisplayLink object (a special internal object, a pointer to which is stored at offset 0x10 in a CVDisplayLinkRef object). Its value is currently null, so this triggers an EXC_BAD_ACCESS / KERN_INVALID_ADDRESS fault. To find other references to this variable, I searched on the string "+0x578]" in the CoreVideo module's assembly code. I only found one, at offset 0x8F in CVCGDisplayLink::setCurrentDisplay(), where its value is set:

```
mov qword [r12+0x578], rax
```

So when the crashes happened, it's presumably because this variable had not yet been set. Discovering this prompted me to add another hook to my hook library:

```
CVCGDisplayLink::setCurrentDisplay();
```

Note that the search on "+0x578]" might easily have turned up references to a variable in some other object, or to a pointer in its vtable. In fact it didn't. But you need to make sure that the references, when you find them, refer to the same kind of object.

Now I'm afraid I wandered off the trail a bit. I assumed that CVCGDisplayLink::setCurrentDisplay() was being called out of order, or not at all. So I looked for a way to provoke this, by hooking various system calls and making them behave incorrectly. Eventually, I hit the jackpot with get_current_display_system_state(), though not in the way I expected.

This is a function in the SkyLight private framework (/System/Library/PrivateFrameworks/SkyLight.framework/SkyLight), where it's called by many different functions, including SLSGetDisplayOpenGLDisplayMask(). This is in turn called (indirectly) from CGDisplayIDToOpenGLDisplayMask() in the CoreGraphics framework (/System/Library/Frameworks/CoreGraphics.framework/CoreGraphics).

CGDisplayIDToOpenGLDisplayMask() is called by several different functions in the CoreVideo framework. Among them is CVCGDisplayLink::setCurrentDisplay(), though this isn't what drew my attention to it initially. In all of these functions but one, a 0 return from CGDisplayIDToOpenGLDisplayMask() provokes an error return from the function that calls it, with the error 0xffffe5f2/-6670 in the $EAX register. In CVReturn.h this translates to kCVReturnInvalidDisplay.

Figure 6: Preliminary Hook for get_current_display_system_state()

So, as an experiment, I installed a hook for get_current_display_system_state() that made it return null every 10th time it was called. This reliably triggered bug 1201401's crashes in Firefox, but not in Safari or Chrome — exactly the pattern observed by people affected by the "real" bug. Clearly, I'd hit a nerve, but I wasn't quite there yet.

Eventually, I re-read the assembly code for CVCGDisplayLink::setCurrentDisplay(), and noticed that when CGDisplayIDToOpenGLDisplayMask() returns 0 and setCurrentDisplay() does an error return, it skips over the code that sets the qword pointer at offset 0x578 in the internal CVDisplayLink object. This leaves the pointer set to its initial value of null.

Furthermore, CVCGDisplayLink::initWithCGDisplays() ignores the return value of CVCGDisplayLink::setCurrentDisplay(). So, then, does its caller CVDisplayLinkCreateWithActiveCGDisplays(), when that's called from application code. Now I knew this was Apple's bug: CVDisplayLinkCreateWithActiveCGDisplays() (indirectly) ignores an error condition, which leads to bug 1201401's crashes.

untitled text

(New Document) (no function selected)

```
1  void *(*get_current_display_system_state_caller)() = NULL;
2
3  void *get_current_display_system_state_hook()
4  {
5  #ifdef __LP64__
6    const struct mach_header_64 *mh = NULL;
7  #else
8    const struct mach_header *mh = NULL;
9  #endif
10   intptr_t vmaddr_slide = 0;
11   GetModuleHeaderAndSlide("crashreporter", &mh, &vmaddr_slide);
12   bool is_crashreporter = (mh != NULL);
13
14   void *retval = get_current_display_system_state_caller();
15
16   if (!is_crashreporter) {
17     static unsigned int counter = 0;
18     unsigned int remainder = 0;
19     bool docrash = false;
20     // Revised test, which shows that these crashes happen only when
21     // get_current_display_system_state() returns NULL during a call to
22     // SLDisplayIDToOpenGLDisplayMask() (via CGDisplayIDToOpenGLDisplayMask()),
23     // during a call to CVCGDisplayLink::setCurrentDisplay(unsigned int), which
24     // itself happens during a call to CVDisplayLinkCreateWithCGDisplays()
25     // (which is called by CVDisplayLinkCreateWithActiveCGDisplays()).
26     // This test crashes Firefox (unpatched), Safari and Google Chrome (and
27     // presumably all other apps that use the CVDisplayLink methods). It can
28     // also be used to test the patch for bug 1201401.
29     if (get_in_setCurrentDisplay() && get_in_DisplayIDToOpenGLDisplayMask()) {
30       OSAtomicIncrement32((int32_t *) &counter);
31       remainder = (counter % 5);
32       if (remainder == 0) {
33         docrash = true;
34         retval = NULL;
35       }
36     }
37     if (docrash) {
38       LogWithFormat(true, "get_current_display_system_state(): returning \'%p\', counter \'%u\', remainder \'%u\'",
39                     retval, counter, remainder);
40       PrintStackTrace();
41     }
42   }
43
44   return retval;
45 }
```

L: 46 C: 1 Objective-C++ Unicode (UTF-8) Unix (LF) (never saved) 1,680 / 152 / 46 100%

Figure 7: Better Targeted Hook for get_current_display_system_state()

With a more carefully targeted hook for get_current_display_system_state(), I was able to trigger bug 1201401's crashes in Firefox, Safari and Chrome. No, Apple and Google hadn't worked around the bug, or even been aware of it. It's just that the bug is triggered far more often in Firefox than in the other browsers. Safari and Chrome are also affected. But they call CVCGDisplayLink::setCurrentDisplay() so much less often that bug 1201401 is basically invisible in them.

Here is logging output from just before the crash, from an old Firefox Nightly in which bug 1201401 is not yet fixed https://github.com/steven-michaud/HookCase/blob/master/Examples/bugzilla1201401/output/Hook-library-crash-output-Firefox-Nightly.txt. Notice that get_current_display_system_state() returns null, CGDisplayIDToOpenGLDisplayMask() returns 0, CVCGDisplayLink::setCurrentDisplay() returns -6670, and then the crash happens.

Now I had the bug, and needed to figure out how to work around it. At first I pored over the assembly code for get_current_display_system_state(), looking for the cause of its intermittent null returns. I also added several hooks to my HookCase hook library. But nothing plausible turned up. And in any case I already had enough information for a workaround. So I abandoned this line of inquiry.

```
+      // Workaround for bug 1201401: CGDisplayIDToOpenGLDisplayMask() is a
+      // documented method with undocumented behavior -- it returns '0' as an
+      // error condition. Through at least macOS Catalina, Apple relies on
+      // this behavior in the CoreVideo framework to indicate that a given
+      // display is (temporarily) invalid (kCVReturnInvalidDisplay). But a
+      // method called (indirectly) from CVDisplayLinkCreateWithCGDisplays()
+      // ignores this error return (again indirectly), eventually leading to
+      // crashes in CVCGDisplayLink::getDisplayTimes(). The workaround is to
+      // call CGDisplayIDToOpenGLDisplayMask() ourselves, and do an early
+      // return if it returns '0' (that is, if a succeeding call to
+      // CVDisplayLinkCreateWithCGDisplays() would trigger Apple's bug and
+      // lead to a crash). It also makes sense to return early if
+      // CGGetActiveDisplayList() fails or there are no active displays.
+      const UInt32 kMaxDisplays = 256;
+      uint32_t displayCount;
+      CGDirectDisplayID displays[kMaxDisplays];
+      if ((CGGetActiveDisplayList(kMaxDisplays, displays, &displayCount) !=
+           kCGErrorSuccess) ||
+          (displayCount == 0)) {
+        return;
+      }
+      for (uint32_t i = 0; i < displayCount; ++i) {
+        if (CGDisplayIDToOpenGLDisplayMask(displays[i]) == 0) {
+          return;
+        }
+      }
+
       // Create a display link capable of being used with all active displays
       // TODO: See if we need to create an active DisplayLink for each monitor
       // in multi-monitor situations. According to the docs, it is compatible
       // with all displays running on the computer But if we have different
       // monitors at different display rates, we may hit issues.
-      if (CVDisplayLinkCreateWithActiveCGDisplays(&mDisplayLink) !=
-          kCVReturnSuccess) {
+      if (CVDisplayLinkCreateWithCGDisplays(
+              displays, displayCount, &mDisplayLink) != kCVReturnSuccess) {
         NS_WARNING(
             "Could not create a display link with all active displays. "
             "Retrying");
         CVDisplayLinkRelease(mDisplayLink);
         mDisplayLink = nullptr;
```

Figure 8: My First, Unsuccessful, Workaround for Bug 1201401

But then, once again, I took a wrong turn. CGDisplayIDToOpenGLDisplayMask() is a documented method, though (as I had now found out) with an undocumented error return. I figured I could call it directly in Firefox code, and do my own error return if it returned 0, without calling CVDisplayLinkCreateWithActiveCGDisplays() or its equivalent. This way the "broken" CVDisplayLinkRef object would never be created, and CVCGDisplayLink::getDisplayTimes() would never crash when called on its internal CVDisplayLink object. I landed a patch based on this idea, but it quickly became apparent that it had almost no effect on the crash rate. Apparently the conditions that can cause get_current_display_system_state() to return null change very quickly. Given two consecutive calls to this function, there seems to be about a 50/50 chance that the first will return null and the second not, or vice versa. So back to the drawing board.

```
      // Create a display link capable of being used with all active displays
      // TODO: See if we need to create an active DisplayLink for each monitor
      // in multi-monitor situations. According to the docs, it is compatible
      // with all displays running on the computer But if we have different
      // monitors at different display rates, we may hit issues.
-     if (CVDisplayLinkCreateWithActiveCGDisplays(&mDisplayLink) !=
-         kCVReturnSuccess) {
+     CVReturn retval = CVDisplayLinkCreateWithActiveCGDisplays(&mDisplayLink);
+
+     // Workaround for bug 1201401: CVDisplayLinkCreateWithCGDisplays()
+     // (called by CVDisplayLinkCreateWithActiveCGDisplays()) sometimes
+     // creates a CVDisplayLinkRef with an uninitialized (nulled) internal
+     // pointer. If we continue to use this CVDisplayLinkRef, we will
+     // eventually crash in CVCGDisplayLink::getDisplayTimes(), where the
+     // internal pointer is dereferenced. Fortunately, when this happens
+     // another internal variable is also left uninitialized (zeroed),
+     // which is accessible via CVDisplayLinkGetCurrentCGDisplay(). In
+     // normal conditions the current display is never zero.
+     if ((retval == kCVReturnSuccess) &&
+         (CVDisplayLinkGetCurrentCGDisplay(mDisplayLink) == 0)) {
+       retval = kCVReturnInvalidDisplay;
+     }
+
+     if (retval != kCVReturnSuccess) {
        NS_WARNING(
            "Could not create a display link with all active displays. "
            "Retrying");
        CVDisplayLinkRelease(mDisplayLink);
        mDisplayLink = nullptr;
```

Figure 9: My Second, Successful Workaround for Bug 1201401

After some thought, I tried another approach. Instead of anticipating when Apple's bug will happen and avoid it, I'd detect it after it happens and avoid its consequences. I'd detect a "broken" CVDisplayLinkRef object and discard it unused. Application level code doesn't have access to the CVDisplayLink object's pointer at offset 0x578. But fortunately, CVCGDisplayLink::setCurrentDisplay() also fails to set another internal variable when it returns early (thanks to get_current_display_system_state() returning null and CGDisplayIDToOpenGLDisplayMask() returning 0) — the "current display" variable. This can be accessed using the documented CVDisplayLinkGetCurrentCGDisplay() function.

Here is logging output showing my workaround in action, from a current Firefox Nightly in which bug 1201401 is fixed https://github.com/steven-michaud/HookCase/blob/master/Examples/bugzilla1201401/output/Hook-library-workaround-output-Firefox-Nightly.txt. get_current_display_system_state() returns null, CGDisplayIDToOpenGLDisplayMask() returns 0, and CVCGDisplayLink::setCurrentDisplay() returns -6670. Then CVCGDisplayLink::getCurrentDisplay(), called from my patch, returns 0. Then my patch calls CVDisplayLinkRelease() on mDisplayLink and does an early return. Then everything proceeds normally, without the crash.

Conclusion

Mozilla bug 1201401 turned out to be quite easy to diagnose and work around, at least in retrospect. If I'd always managed to look in the right places, I could have done the job using Hopper Disassembler alone. But that's not how it looked at the beginning: Fixing this bug seemed more or less impossible. Assembly code is difficult to understand. You don't so much read it as decipher it, based on assumptions you make as you go along. Testing these assumptions is much easier if you have a way to hook any function in almost any module, as HookCase allows you to do.

There are surely many other "impossible" bugs that HookCase can make more approachable. It should also help those who want to reverse engineer part of macOS for its own sake. The HookCase distribution has several examples of this: https://github.com/steven-michaud/HookCase/blob/master/6-examples.md.

About the Author

Steven Michaud is enjoying his retirement. He has a life-long interest in photography, which for the first time he's able to indulge fully. He reads novels. He's trying to learn French. But for about eight years he worked for Mozilla as a macOS developer. Now he's hooked on reverse engineering, and will never fully escape it.

Deep Learning for Digital-Image-Forensics

by Akash Nagaraj, Bishesh Sinha, Mukund Sood, Vivek Kapoor and Yash Mathur

The primary issue faced during investigations of criminal activity with respect to video evidence, is determining the credibility of the video and ascertaining that the video is unedited. As of today, one of the most crucial ways to authenticate images or footage is to identify the camera that the image was taken on. A very common way to do this is by using image metadata which can easily be falsified itself, or by splicing together content from two different cameras. Many solutions have been proposed in the past, however, this was a problem yet to be solved to a reliable extent. Our intention was to build a system that identifies the camera model used to capture an image from traces intrinsically left in the image - a digital fingerprint of sorts. Solving this problem has a big impact on the verification of evidence in criminal and civil trials and even news reporting.

As with any Deep Learning based system, ours too required data. The dataset was composed entirely of raw images, with no explicitly labelled attributes. That being said, we manually extracted Source Pattern Noise, Interpolation Detection and Gray-Level Co-occurrence Matrix (GLCM) Features from each image. Ten different mobile camera models were used, namely:

- Sony NEX-7
- Motorola Moto X
- Motorola Nexus 6

- Motorola DROID MAXX
- LG Nexus 5x
- Apple iPhone 6
- Apple iPhone 4s
- HTC One M7
- Samsung Galaxy S4
- Samsung Galaxy Note 3

Images in the training set were captured with 10 different camera models, a single device per model, with 275 full images from each device. The images in the test set were captured with the same 10 camera models, but using a different device. For example, if the images in the training data for the iPhone 6 were taken with A's iPhone 6, the images in the test data were taken with B's iPhone 6. None of the images in the test data were taken with the same device as in the training data.

While the training data includes full images, the test data contains only single 512 x 512 pixel blocks cropped from the center of a single image taken with the device. No two image blocks come from the same original image.

Half of the images in the test set were altered, to provide a larger variance. Based on our findings through the Literature Survey, we decided to extract three features: Source Pattern Noise, Image Interpolation and GLCM.

To extract Source Pattern Noise, we passed the original image through a high-pass filter. We used various filters (Bilateral, Laplacian, Canny, etc.), which resulted in a denoised image. We then subtracted the denoised image from the original image to obtain only the noise for the image. The next step was to take only the largest components of the extracted noise, the top 4% of the values were taken in our case. The same procedure was followed for every image of the dataset.

To detect Image Interpolation, we followed the Gallagher and Chen Algorithm. First, convolving the image with a high pass filter to remove low-frequency information then, calculating the variance across diagonals to form a single dimensional signal of variances. The diagonals are used because it is clear from the pattern that the variance across diagonals varies periodically. For an interpolated image, we expected to find periodicity in this signal, and to find this periodicity, we used a Discrete Fourier Transform (DFT), and for Colour Filter Array (CFA) Interpolation, we expected to see a peak in the DFT. We again performed this procedure for every image in the dataset. This type of preprocessing is similar to Photo Response Non-Uniformity (PRNU) estimation, however, we did not use PRNU estimation because that is a direct result of a defect specific to a single camera; it can be thought of as a digital fingerprint of an individual camera.

To extract GLCM Features, we passed the original image through a Canny Filter as well as a Laplacian Filter separately, following which we compared each pixel value of the resulting two images (after being passed through the filters) and formed a new image, consisting of the maximum pixel values. This new image was subtracted from the original image. We applied thresholding to the image, and converted it to gray-scale. This final image was then used to extract GLCM features using the Gray-Level Dependency Matrix.

We used a novel preprocessing method for the ResNet50 Model. Our initial approach was to simply pass the image through a high pass filter, and then pass the resulting image to ResNet and to find intrinsic patterns in the image and give a higher accuracy. However, this did not give us very positive results. That being said, the results were promising, so our next step was to make a Probability Plot of the noise in each image. Our method was simple, given a single image, we took a random 256x256 crop of the image, and applied Gaussian Blur Filter to the image to remove noise. Once noise was removed, we subtracted the original crop from the Filtered crop to obtain the noise in that crop, following which we applied a Fast Fourier Transform to the resulting crop, with four different shifts, to obtain four different images. We did this 256 times for each image, and finally the result of the 256 random crops of the image was averaged out to give us four images, which were finally saved to disk.

This project was our entry to the IEEE's Signal Processing Society - Camera Model Identification Competition. We uploaded our predictions to Kaggle and got an overall accuracy of 89.85%

The test data used by kaggle contained only single 512 x 512 pixel blocks cropped from the center of a single image taken with the device. No two image blocks come from the same original image. It was also known that half of the images in the test set had been altered.

The results that we achieved were so because we used an immense amount of pre-processing. Given how large the dataset was (not in terms of images, but the size of each individual image), the training of each model was extremely time-consuming and computationally intensive. This project lies in the intersection of Digital Forensics and Digital Image Processing, hence the preprocessing applied was of great importance, to achieve a reliable and practical solution.

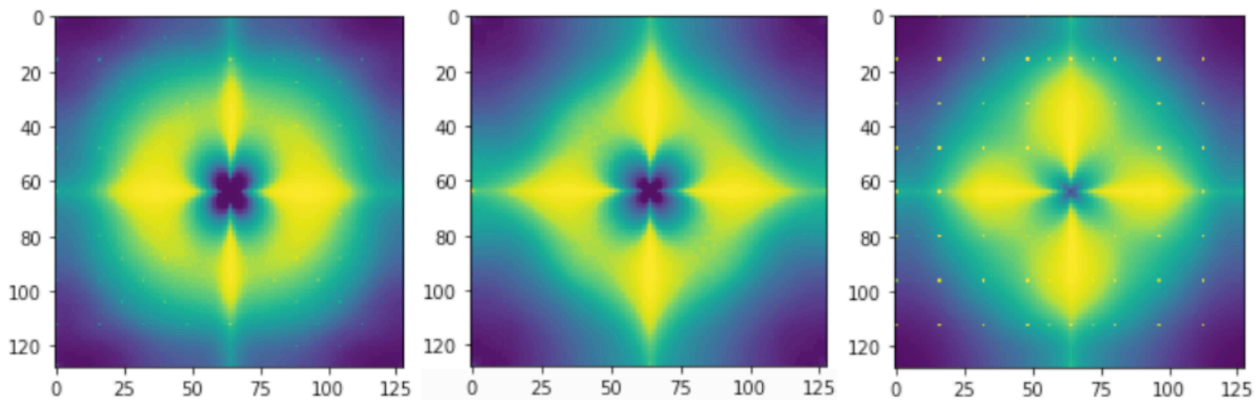


Figure: Probability plots of the noise generated from: i) iPhone 4S ii) Samsung Galaxy Note 3 and iii) iPhone 6

We also came up with a novel idea where we generated probability plots of the noise generated from the images of a particular camera. By visually inspecting this data, we could see that there was in fact, a distinct pattern that could be differentiated across cameras. Hence, this was what we used to build a more robust and reliable model.

From a deep learning perspective, the dataset was not as large as traditional datasets, hence we augmented it with generated 256x256 crops of each image to increase our dataset size, to improve the accuracy, and variance. The Noise Probability Plots were used to train our ResNet50 Model, and the 256x256 Image Dataset, we used to train our SVM Classifier.

As can be seen from our results, the ResNet50 Model outperformed the SVM Classification Model, keeping in mind the fact that the ResNet50 Model was being tested on a harder Dataset.

The SVM Classifier found it difficult to differentiate between images taken from a camera that was produced by the same company, however, the ResNet50 Model, being a very deep network, seems to have been able to avoid this issue, as it was able to find much more subtle relationships between images taken by cameras produced by the same company.

From our results, we saw that camera models that are of the same company have similar characteristics and the classifier has a harder time differentiating between them. This could be due to the fact that cameras built by the same company tend to have similar base defects that affect an image, as well as the fact that their underlying base algorithm to improve the image, such as mosaicing, would be similar across all phones developed by the same company.

In conclusion, we find that the preprocessing involved was imperative in achieving the results we achieved. We see that preprocessing is an integral part of any Machine Learning problem (or even a Deep Learning problem for that matter), and a lot of research must go into the problem statement before prototyping any sort of models.

References:

1. IEEE's Signal Processing Society - Camera Model Identification, https://www.kaggle.com/c/sp-society-camera-model-identification/notebooks

2. Deep Residual Learning for Image Recognition - Kaiming He et. al

3. Source Camera Identification using GLCM - Nilambari Kulkarni et. al

4. Detection of Linear and Cubic Interpolation in JPEG Compressed Images - Andrew C. Gallagher

5. Experiments on Improving Sensor Pattern Noise Extraction for Source Camera Identification - Giuseppe Cattaneo et. al

About the Authors

The members of the team that developed this system are: Akash Nagaraj, Bishesh Sinha, Mukund Sood, Vivek Kapoor and Yash Mathur, under the guidance of Dr. S Natarajan. They are former undergraduate students at PES University, India where they majored in Computer Science. They are currently working as software developers for leading companies. Amongst themselves, they have participated in and won multiple hackathons, presented multiple papers in top conferences on the fields of Machine Learning and Big Data. They are all very passionate about research in diverse domains related to Computer Science.