Forensic Reverse Engineering with Rekall

Workshop notes

Introduction
What do we mean by "Forensic Reverse Engineering?" Not rigorous reverse engineering by reading disassembled code to decipher the inner workings of an application, but analysis techniques that allow us to observe the internal state of an application and figure out just enough of how it works. The goal here is to extract relevant information from memory with potential evidentiary value. Since that mostly relies on context, this tends to work well with software which is not deliberately trying to avoid reverse engineering. Hence the techniques discussed might be less appropriate for analysis of malware.

Like a good movie plot, this workshop's scenario is a bit contrived. We do not aim to be realistic on all fronts - especially the legal and practical aspects are a bit contrived. Please suspend your disbelief temporarily at our rather poor imagination and crime-novel like plot development. The main focus of the workshop is to provide you insight into techniques that can be applied on real cases.


The solutions can be found here.

Why Rekall?
Rekall is more than just a memory analysis tool - it is a complete framework for memory analysis, and also includes a powerful interactive interface. In previous workshops we covered many of the existing plugins that Rekall already comes with. While this is useful when you encounter common situations already covered by these plugins, however most investigations have a unique aspect to it not covered by existing plugins.

This type of analysis is much easier to perform within an interactive interface. So in this workshop we focus on Rekall's interactive interface and the additional powerful analysis techniques it provides. We will not be covering the normal set of plugins that are covered in other workshops, rather we want participants to learn how to help themselves in situations that they might encounter. Especially in those cases where an existing plugin is not written already. Rekall allows an investigator to drill down into memory, and with a few basic skills, enables anyone to write their own plugin in order to address their specific needs.
Scenario - Part 1

You are a DEA agent Hank Schrader and you have just received a call from customs officials about an intercepted package containing some of the bluest, purest meth samples they had ever seen. The package was sent via a courier to the offices of Madrigal Electromotive GmbH - A large company based in Germany. Their local offices employ about 500 employees. The package is addressed to a Mr. Homer Simpson, a machine operator with a sad doughnut addiction.

"This might be the breakthrough I am looking for!" you think in delight... Finally a chance to break this large operation.

It is 10am on a Monday. You organise a controlled delivery operation and prepare to arrest Mr. Simpson when he opens the package. At 10:15 Mr. Simpson picks up the package with a puzzled look, takes it to his desk and immediately falls asleep at the keyboard.

For the next two hours a large number of people walk past Homer's desk, some borrow things from his desk. One anonymous guy can be seen stealing a doughnut from Homer's Krispy Kreme box - right under his sleeping nose.

Finally at noon, Homer Simpson wakes up and opens the package. To his surprise, and the surprise of the armed DEA agents, who by this stage stormed the room and arrested Homer - the package is completely empty! Mr. Simpson claims to have no idea what the package was, never ordered it and is just crying out for someone called Marge - perhaps his lawyer.

Unfortunately we have no case against Homer - he just does not look like the dreaded Heisenberg... That guy can't hurt a fly.

Someone must have intercepted the package while Homer was asleep. But who? Was it one of the people rummaging through Homer's desk while he was sleeping? Was it someone in the mail room?

You scramble your favourite judge and get together a search warrant. The warrant allows you to search all computer systems in the local offices of Madrigal Electromotive GmbH under the following restrictions:

1. You may only search for information relevant to the package:
   a. The Tracking number is FAX12748583623-UZ
   b. The package is addressed to a "Homer Simpson"
   c. The package was sent from "Herr Hertzog"

2. You may not acquire any forensic evidence without probable cause!

Time is of the essence!
You speak with the IT manager and he indicates that all corporate systems are running GRR.
   1. You use GRR to search memory for the keywords covered under the warrant.
   2. Do you have probable cause for acquisition? Use GRR to acquire the system's memory.

Schedule a GRR Memory Collector hunt. Select a literal match for the package tracking number. Ensure that GRR makes a local copy first, and downloads the memory image.

Have GRR send an email for each hit:
Schedule the hunt on all the windows systems:

Jesse Pinkman's machine returns a positive hit. There is probable cause now to acquire his machine's memory, however, we can not really tie Jesse to the package. There is not enough context to charge Jesse.

A suspicious user called Jesse Pinkman has the tracking number found in physical memory. You detain Jesse for questioning but his lawyer, Saul Goodman is challenging! Do you have enough to finally lock him up?

Exercise 1:
Copy Rekall to your workstation and begin inspecting the acquired image. In this workshop we will use the stand-alone version of Rekall - this does not require an installation. The image is called hank.aff4.

Search the physical memory for the tracking number.

What potential defences can Saul Goodman come up with to explain why the tracking number is in physical memory?
Can you refute these explanations? Consider possible counter arguments and test these theories.

Exercise 2:
You managed to pinpoint the tracking number to the process memory of the **Miranda Instant Messaging** application. This is an application used to chat with other users and a working theory is that the tracking number was delivered to Jesse over chat from the real perpetrator - maybe even **Heisenberg** himself!

Do we have enough to charge Jesse now?

Exercise 3:
Before we can charge anyone we need to prove knowledge and intent. In particular we need to show that Jesse received, and responded to the message. We also need to know the exact time the message appeared and proof that Jesse himself responded.

In order to do this we need to work out how **Miranda IM** stores messages in memory. We have installed **Miranda IM** on our test machine and joined some IRC channels. The image is called **MirandaTest.aff4**. For the next few exercises we will use that image.

As investigators we have never even heard of Miranda IM which is not terribly popular. We search the net for Rekall plugins that can analyze it but came up empty. Looks like we have to write our own plugin.

The miranda screenshot is shown below:
We see a number of important aspects:

1. The timestamp a message was sent.
2. Who the message was sent from.
3. The message text itself.
4. The list of users in the channel.
5. The channels currently subscribed to.

Our goal is to understand enough of how Miranda IM works so as to write a Rekall plugin to reconstruct the above data from the memory image. Using this information we can find the context around the keyword hit and thereby ensure that the suspect had **Mens Rea**.

- Switch to the miranda process context and enumerate all heap allocations.
- Find the strings in the screenshot above within the Miranda process. The following plugins are useful (you can read their documentation on the Rekall documentation site):
  - grep
  - yarascan
  - sigscan
- Inspect the heap allocations (using the `inspect_allocation` plugin) that contain these strings. Note that the string would appear multiple times as part of a number of different subsystems (can you tell them apart?):
  - Miranda IM internal data structures
GUI structures maintaining the visual elements themselves (part of the windows UI toolkit).

- Inspect references to these allocations using the `show_refferer_alloc` plugin. Try to determine a pattern between the messages, and the data structures which refer to them. It is useful to draw a picture to visualize relationships between structures. Common data structures include (try to recognize these by sight):
  - Doubly linked lists
  - Singly linked lists
  - Counted strings (i.e. length followed by string).

- Write Rekall Struct definitions to define a specific profile for the Miranda IM program. Use the profile interactively to inspect allocations.
  - Below is an example skeleton file to get you started (you can run it from the console using "run -i test.py"):

```python
from rekkall import obj
from rekkall.plugins.overlays import basic

# These are the private type definitions... Add fields to this as you figure out
# more about Miranda's private types.
TYPES = {
    "MESSAGE_RECORD": [0x50, {
        "Message": [0, ["Pointer", dict(
            target="UnicodeString"
        )]],
    }],
}

# This is needed so you can reload this file over and over. We remove previous
# of the Miranda profile from the registry.
obj.Profile.classes.pop("MirandaProfile", None)

class MirandaProfile(basic.ProfileLLP64, basic.BasicClasses):
    ":""A basic profile for Miranda IM."":"

    @classmethod
    def Initialize(cls, profile):
        super(MirandaProfile, cls).Initialize(profile)
        profile.add_types(TYPES)

miranda = MirandaProfile(session=session)
```

These are some screenshots of the Miranda user interface. In this case we only care about IRC messages but for extra credit... our plugin could extract all configured information (e.g. user nicks, passwords for private channels etc). Try to explore how Miranda keeps its own configuration information so our plugin can extract this from memory.
Exercise 4: Write a reusable plugin

We can write a reusable plugin using our test image.

You can load an external plugin using the --plugin command line flag:
from rekall import scan
from rekall.plugins.overlays import basic
from rekall.plugins.windows import common

TYPES = {
    "MESSAGE_RECORD": [0x50, {
        "Message": [0, ["Pointer", dict(
            target="UnicodeString"
        )]],
        .....
    }],
    "CHANNEL_RECORD": [0x50, {
        .....
    }],
    "CHAT_RECORD": [96, {
        .....
    }],
    "USER_RECORD": [0x28, {
        .....
    }],
    "CHATS": [48, {
        .....
    }]
}

class MirandaProfile(basic.ProfileLLP64, basic.BasicClasses):
    """A basic profile for Miranda IM."""

    @classmethod
    def Initialize(cls, profile):
        super(MirandaProfile, cls).Initialize(profile)
        profile.add_overlay(TYPES)

class Miranda(common.WindowsCommandPlugin):
    name = "miranda"

    def FindChannels(self):
        """Finds channel records."""
        .....

$ rekall -v -f MirandaTest.aff4 --plugin miranda.py

The skeleton of a plugin is:
def render(self, renderer):
    # Create a specialized miranda profile with our reversed structs.
    self.miranda_profile = MirandaProfile(session=self.session)

    # We need to switch into the Miranda process address space!
    with self.session.plugins.cc(proc_regex="miranda") as cc:
        cc.SwitchContext()

    # Find all the channel records.
    for channel in self.FindChannels():
        # For each channel we start a new section.
        renderer.section("Channel {0} {1:#x}".format(
            channel.Channel, channel))

        # Gather all the users
        users = []
        for x in channel.FirstUser.walk_list("NextUser", True):
            users.append(unicode(x.Nick.deref()))

        # Make a new table for users so it gets wrapped.
        renderer.table_header([("Users", "users", "120")])
        renderer.table_row("", ",".join(users))

        # Make a table for messages.
        renderer.table_header(["Timestamp", "timestamp", "30"],
            ["User", "user", "20"],
            ["Message", "message", "80"],
        )

        for record in channel.FirstMessage.walk_list("Next"):
            renderer.table_row(..., ..., ...)
maxlen = session.GetParameter("highest_usermode_address")
for hit_offset, string_hit in scanner.scan(maxlen=maxlen):
    ........

**Hint 2: Scanning for references:**
To efficiently scan for pointers to a set of addresses, we can use the PointerScanner:

```python
from rekall import scan

scanner = scan.PointerScanner(
    session=session, profile=session.profile,
    pointers=[x+4 for x in irc_hits])

for referrer in scanner.scan(maxlen=maxlen):
    ........
```