Rekall Agent User Manual

Michael Cohen

scudette@rekall-forensic.com

Why a Rekall Agent?

Data representation

How does Rekall Agent avoid these problems?

Alternatively: OSQuery

Alternatively: EFilter + Rekall

Rekall Agent Overview

Comparison with existing frameworks

The Rekall command line

Basic concepts

EFilter queries

Authentication

GCP IAM Access Controls

Rekall Agent Server Access Controls.

Conditions

Roles

Clients

Client Labels

Searching and viewing clients.

Permissions and Access Control

Auditing

Flows

Result Collections

Uploaded files

Canned Flows

Exporting a Flow

Hunts

Installation.

Client installation

Windows Installation

Ubuntu Linux Installation

Client configuration and deployment.

---

1 Rekall is an open source project released under the GPL. It is not an official Google product, and does not necessarily reflect the views of Google.
Polling mode
Development workflow

API access
Token delegation
Notes about security
Why a Rekall Agent?

While Rekall started off being a purely forensic framework operating on dead memory and later disk images, in recent releases Rekall has incorporated a full-fledged incident response (IR) capabilities. Rekall has always put emphasis on live analysis, and extending the framework to be a complete Incident Response solution, was an obvious choice.

However, incident response by its nature, is most useful when done remotely and automatically, scaling to large numbers of machines easily.

There are a number of other excellent frameworks for remote incident response available, such as GRR, OSQuery, and MIG. Rekall aims to be a best of breed incident response platform - we looked at some of the other open source offerings and examined some of the weaknesses and desired features of each. For Rekall our main goals are:

1. The server should be easy to deploy and scale.
2. The design should be very simple and obvious.

Data representation

When performing incident response at scale, a significant amount of data is typically collected. Not only does every system generate information about various forensic entities (such as Processes, User accounts, Event log entries etc), but typically we also wish to collect files from each system, such as configuration files, and suspicious binaries. We observed some of the more frequently used features in existing tools:

1. While endpoint collection systems do offer some analysis capabilities (e.g. GRR offers timeline construction directly in the GUI), in practice, users rarely use the endpoint tool for performing analysis directly. Users typically just export the results of flows or hunts to some other system, construct their own analysis pipeline, or write custom scripts which operate on the exported data. There are many powerful analysis and data mining systems (e.g. BigQuery, Hadoop or Spark) and any specific analysis that the endpoint tool can do can generally be done more rapidly and easily by other systems.
2. Many existing solutions have elaborate data modeling. For example, GRR uses protobufs to permanently store information about different entities (e.g. Processes, Files etc) found on the endpoint. Maintaining this data model in permanent storage presents its own challenges. As the data model evolves to include additional fields, the database schema needs updating to include the new information. This requires code changes, with a subsequent release cycle.
3. Data conversion can be a resource demanding task and not easily scalable. Tools that use elaborate server side data modelling often perform several unnecessary data conversions steps. Data is collected from the client using a certain data model, it is then...
copied into the server side data model, and when exported often converted again to the required export format. For example, the GRR client encodes a forensic entity in a certain way filling in some fields (e.g. a User protobuf is sent to describe different aspects of a user account such as username, home directory etc.). The data is then transformed on the server into a database representation to populate the persistent data model. Finally, when exported, the data is transformed again to convert it into a suitable export format (e.g. to CSV or JSON). All these transformations are pointless and resource intensive.

4. Version matching between server and client is a real issue - as the data model used to interchange entities between client and server evolves, the client code needs to evolve in step (e.g. client code needs to change to include another piece of data such as User's home directory). Unfortunately, client binaries are hard to update once deployed and there are many older clients around with different versions or capabilities.

How does Rekall Agent avoid these problems?

Rekall takes a different approach. The Rekall Agent system is merely a collection system. Its primary task is to collect information from clients, store it and export it (Without modification) when required. The system merely takes care of scheduling, resource management, access control and auditing. This means that Rekall does not maintain or track a specific, static, data model of what each forensic entity represents (i.e. there is no Users table in the database which contains information about all the users from a client, or an Interfaces table which records all the network interfaces of each client etc.).

Instead, Rekall collects the information required on a case by case basis by specifying a flexible query from the server. The query is run by the client each time, producing data in the required format. This flexibility comes about because Rekall contains the efILTER query engine, allowing the server to issue arbitrary filtering and selecting queries from the client.

Let's look at a more concrete example. Our sysadmin tells us that machine monitoring for our web servers say that CPU use is very high for some reason. It seems that some of our web server processes are stuck using up much CPU.

We therefore wish to list all processes launched from the web server which have been using significant CPU resources. We then wish to collect some context about these processes to figure out what they are.

Processes launched from the web server's CGI handler typically have environment strings that reveal a lot of information about the request. In particular the environment string GATEWAY_INTERFACE is indicative of a CGI script.
However, normal CGI scripts run for a short time (say a few seconds). We are not interested in them so we need to filter them out.

GRR provides a means of listing all processes (the \texttt{ListProcesses} client action), and models data about each process in a specialized \texttt{Process} protobuf. GRR prepares the protobuf in the client and sends it to the server, including many fields, such as username, process working directory, open files etc. For each process the GRR client completely fills all information into the Process protobuf because it has no idea if the user wants this information. So with the GRR approach we have to collect ALL the information about all the processes, then post process the data to filter out only those processes with high CPU usage.

At the time of writing, the GRR data model contains CPU usage statistics in the \texttt{Process} protobuf, but not a list of environment strings. However, for the current query, we really need to enumerate environment strings so we can collect context around the CGI query.

We have two options:

1. \textbf{GRR's server side processing model:} Construct a server side flow to run this query: We can list all processes from the client in one step. This list then will need to be filtered on the server, which issues subsequent queries to enumerate environment strings for each process. It works but requires dedicated server side code (termed a FLOW) to be developed specifically. This approach also requires significant server side processing and has a lot of latency, since each step requires a full client/server round trip (often by the time we figure out which process to dump the process has exited). Before we can use this we need to push the new software server side (Typically need to wait for a release cycle).

2. \textbf{GRR's client side code:} Alternatively, we can modify GRR client code to add another field to the Process protobuf, say \texttt{environment_strings}, and have the GRR client populate this field in the \texttt{ListProcesses} client action.

If we use option 2 above, GRR will still deliver all the data for all the processes. We need to use post processing to filter the data, unless we add specialized filtering in the GRR client on an environment string. Implementing additional client side filtering requires coding new filtering code in the client action, and adding potentially another parameter to the client action (e.g. "filter\_by\_environment\_strings").

Now there is a problem - the new code needs to be pushed to all clients in some way before this new client feature can be used. In practice, with many thousands of deployed clients, a client version update is very slow and risky\textsuperscript{2}. At any one time, there will be a mix of old and new

\textsuperscript{2} GRR supports a mechanism called \texttt{python hacks} which are snippets of arbitrary python code sent by the server to be evaluated on the client on a case by case basis. This essentially provides arbitrary code
clients in the wild, and issuing a flow which depends on new client features has a large chance of failing (because old clients do not know how to interpret the new fields).

Alternatively: OSQuery

The OSQuery platform solves this problem very nicely. OSQuery has a set of virtual tables in the client which collect different information about different forensic entities. For example, there is a processes table and an process_envs table. The user can then issue an SQL statement which performs a JOIN between the two tables and issue a where clause which filters only those processes we need. The query can be changed as the user requires in a straightforward and logical way without requiring client code updates. The ability to combine different tables in arbitrary ways essentially increases the tool's capabilities exponentially without requiring each such feature and combination to be coded into the client code.

For example an OSQuery to discover CGI processes is:

```sql
select * from processes, process_envs where processes.pid == process_envs.pid and process_envs.key == 'GATEWAY_INTERFACE'
```

In the above screenshot we see that the two OSQuery tables are joined (on pid) and filtered (with environment string = GATEWAY_INTERFACE). We immediately recognize a process called john which has been running for significant time.

Since OSQuery is already included in Rekall, we can use either tool - for more complex queries we can use EFilter, but simple queries can use OSQuery. As far as the Rekall Agent Server is concerned there is not much difference since both methods generate the same result collections.

execution capabilities and can help in situations where waiting for code pushes is not possible. It is a sledgehammer mechanism though and should be avoided as much as possible.
Alternatively: EFilter + Rekall

Let’s examine how to solve this with Rekall using an EFilter query. Rekall provides the pslist() plugin for returning information about processes. Using this plugin, we can just request exactly the information we need (see below how to get to this query):

```
select proc, proc.cpu_times.user, pid, ppid, proc.environ from pslist()
where proc.environ.GATEWAY_INTERFACE and proc.cpu_times.user > 10
```

The client sends a list of processes which match the condition instead of all the processes. Not only do we not need to do any post processing, but the amount of information sent is tailored precisely to what is required. Most importantly is that we do not need complex server side processing to get the result we are after - the information retrieved is precisely what we want: All the CGI processes which have high CPU usage.

In this case we see the additional environment strings reveal that the process was launched from a web shell and we can also see the remote address of the user who launched the shell!

The EFilter query allows us to send information tailored to our needs. Most importantly though, since we do not need server side processing the server component does not generally need to understand any of the queries returned. The Rekall server simply collects the results and stores them without parsing or modeling them at all.

Rekall Agent Overview

The Rekall Agent is designed around Google Cloud Platform - a highly scalable and cost effective platform. Specifically we use Google AppEngine and Google Cloud Storage to provide the resources we need to collect files and data from clients.
Rekall Agent uses the AppEngine NDB database (powered by Google Bigtable) to store metadata about clients, collected result collections, and uploaded files. Bulk data (such as uploaded files and result collections), are stored directly in the Google Cloud Storage. The GUI which the user interacts with is also served by AppEngine.

Comparison with existing frameworks

There are several existing and very capable open source frameworks we drew our inspiration from.

The GRR framework is our direct inspiration. It is a mature incident response framework. GRR has a static data model encoded in protobuf and database schema and so it requires code changes to update, as described above. However, GRR excels at enterprise features. With a powerful API access can be automated and easily integrated into a larger pipeline. A modern GUI allows analysts to monitor the progress of hunts and flows. GRR also offers enterprise grade access management with client and hunt approval workflows. We like GRR so much that we emulated these great features: Rekall has a powerful API, client and hunt approval workflows and exporting capabilities.

OSQuery is a very powerful incident response framework, in spirit very much like EFIlter and Rekall. It also offers the ability to construct arbitrary queries to customise analysis using an SQL like syntax. OSQuery is currently missing many of the server side features (such as collection management, access controls, auditing etc) but it does offer a similar SQL like interface for client side operations. We like OSQuery so much, that we integrated it as a plugin within Rekall. It is possible to launch OSQuery queries on clients, and collect them into result collections, leveraging the best of both worlds - powerful client side queries and enterprise grade management.
The Rekall command line

At its heart, the Rekall Agent is just a management system built around the regular Rekall binary. Therefore, anything that Rekall can do, can also be done by the Agent. Similarly, the Agent Server simply issues those same commands to the agent itself, as one would issue to the Rekall console. This makes it very convenient to test and understand how Rekall will work for a given query.

This section gives a brief outline of the regular Rekall console and specifically explains how one could use the console to figure out exactly what kind of EFilter query should be run.

Basic concepts

Rekall commands are called plugins. A plugin is a specific command that can be used to do some work. It typically takes plugin specific arguments, and produces a single table of results. From the interactive console, we can always see information about each plugin, by following the plugin with the "?". For example, we can see information about the `pslist` command:

```
[1] Live (API) 03:51:01> pslist
Plugin: APILslist (pslist)
This is a Typed Plugin.
Positional Args: pids: One or more pids of processes to select. (type: ArrayIntParser)
Keyword Args:
    proc_regex: A regex to select a process by name. (type: RegEx)
    verbosity: An integer reflecting the amount of desired output: 0 = quiet, 10 = noisy. (type: IntegerParser)
Docstring: A live pslist plugin using the APIs.
```

Rekall uses the concept of a session. A session is an object which groups together a single analysis session including configuration and cached results. It is possible to run multiple different instances of Rekall in different sessions at the same time.

Usually the Rekall Agent does not operate on dead images, instead it uses live mode. There are currently 2 main flavors of live mode. The Live Memory mode (specified with `--live=Memory`) uses memory analysis to perform deep analysis of the machine's physical memory. In this mode, Rekall will load the relevant drivers to gain access to physical memory. On the other hand, the Live API mode (specified with `--live=API`) loads plugins which access the machine's state via the documented APIs.

Depending on the loaded mode, Rekall loads different plugins. For example, listing processes using memory analysis is different than using the API.
results, since variations in kernel and application versions will cause it to break. It is also more CPU intensive to process physical memory.

We recommend API live mode be used routinely, while memory analysis only be used as additional deeper investigation warrants.

EFilter queries

We have described above how EFilter can be used to specifically tailor the desired output from the Rekall Agent. But how did we come up with the above query? In this section, we describe how to iteratively figure out the exact query we need and look at the anatomy of the EFilter query in more details.

Since EFilter and Rekall make so much data accessible from the host, it is not practical for us to document all the different fields available to be exported and filtered on. It is therefore easiest to experimentally determine the correct query by starting Rekall using the interactive console. Let us consider the following example: We wish to see which processes have an open file whose filename matches ".sqlite".

Since we want to list processes, a good starting point is the pslist plugin. But at this point we are not quite sure what information the pslist() plugin offers. Let's start by asking Rekall to describe the pslist plugin for us:

```
[1] Live (API) 11:02:06> describe pslist
Field                        Type
-------------------------------------------------- ----
proc                                             LiveProcess
Name                                             str
pid                                              int
ppid                                             int
Thds                                             int
Hnds                                             NoneType
wow64                                            bool
start                                            UnixTimeStamp
binary                                           str

10
```

These are the same fields we see when we run the plugin, but we notice that each column has a type. Some of the types are obvious (str, int etc) but some are more complex such as LiveProcess, UnixTimeStamp etc.

The complex types are likely to have additional fields or operations we can call on with efilter queries. To drill down we specify the max_depth argument:

```
[1] Live (API) 12:39:05> describe pslist, max_depth=4, args=dict(proc_regex='rekall')
Field                        Type
-------------------------------------------------- ----
proc                                             LiveProcess
. as_dict                                          instancemethod
  . cmdline                                          list
```
The describe plugin runs the plugin to determine the plugin's output from the first row emitted. Under some situations, some fields will not be filled in (for example if the first process listed does not happen to have any network connections). You can control how the plugin is run by passing plugin args - this is most useful to select an interesting row which might contain valid data for some field.

We notice that the proc member has a field "open_files" with a subfield of "path". We can use the efilter regex matcher (i.e. the =~ operator) to match on the file path.

```
[1] Live (API) 12:47:39> select proc, proc.open_files from pslist() where
    proc.open_files.path =~ ".sqlite"
  proc open_files
  ----------------- -----------------------------------------------
  rekal (9454)  - 0:
    fd      3
    flags   557058
    mode    r+
    path    /home/scudette/.ipython/profile_default/history.sqlite
    position 8192
  - 1:
    fd      4
    flags   557058
    mode    r+
    path    /home/scudette/.ipython/profile_default/history.sqlite
    position 114688
```
Try to answer the following questions by constructing suitable EFilter queries:

1. Show all processes which are listening on sockets.
   a. Show their command lines, which sockets they are listening on and any open files.
2. Show processes that were started remotely from an SSH connection.
   a. Where did the SSH connection come from (IP address)?

In this section, we used the local interactive Rekall console to understand how efilter queries work locally. This is a useful aid to help us develop and test suitable queries, but the really interesting thing is running these queries remotely at scale on a large fleet of machines. In the following section, we see how EFilter queries are integrated into the Rekall Agent software via the search plugin, and how we may run these remotely.

Authentication

The Rekall agent has a significant level of access to endpoints. In most organizations, this access needs to be carefully controlled and audited to prevent abuse and provide an audit trail.

Rekall provides a role based access control mechanism to various securable objects. The access control mechanism is modelled after the Google Cloud Platform's own IAM offering which makes it easier to understand and explain to new users.

Rekall has two access control mechanisms:

GCP IAM Access Controls

As a regular GCP App Engine application, the Rekall server is subject to the usual controls offered by the GCP IAM mechanism. Within the GCP console, one can assign a user the AppEngine Admin role allowing the user to manage the AppEngine app itself.
Note that users who have these privileges can control the App Engine app deployment itself (e.g. update to a new version of the code). This provides users with the highest level of access for the Rekall Application and should be assigned very sparingly. Typically, the AppEngine Admin role is only assigned to a single person or small team responsible for code pushes.

Within the Rekall Agent application itself, users that have the GCP AppEngine Admin role are treated as super users. They can do any action and obtain all permissions. This property is important to allow the initial application deployer to start assigning user roles within the application (solving the deployment bootstrapping problem).

In addition to the additional user administration privileges, application admins receive an additional database management interface which allows directly reading and manipulating the internal raw database tables. This feature should be treated with extreme care!

Rekall Agent Server Access Controls.

The Rekall application itself maintains a role based access control mechanism. Note that Rekall does not need to manage user accounts (like creating accounts, resetting passwords etc) because user accounts are already managed within the GCP platform (i.e. users typically log into Google for identity management). Rekall receives the authenticated user's email address from the platform without needing to deal with logging the user in.

To perform an application action, the user must possess a permission. There are numerous permissions for different parts of the application, where permissions roughly correspond to application actions. For example, in order to search for clients, the user must have the "clients.search" permission.

The Rekall access control mechanism is role based. A role is simply a collection of permissions, that represent a coherent set of related operations. Roles are assigned to users thereby implicitly granting the user with a set of permissions. For example, the Viewer role allows the user to log into the application, and to search for general metadata about clients. However, this role does not grant the user the ability to inspect flows on the client, or manage other users.

A single user may be assigned multiple roles. Each such assignment is treated separately. You can see which roles are assigned to each user by clicking the "Users" toolbar icon. Clicking on each role will show a description of the role's purpose and all the permissions granted by it.
Roles

The Rekall application checks for specific permissions granted to each user when executing a task. However, users are not given individual permissions. They are instead granted a role which is a collection of permissions. Roles are described in the file roles.yaml. It is possible for deployers to change this file to provide customized roles possessing different sets of permissions. In this way, the specific deployment can allow for granular access control requirements.

Here is an example of a role definition from roles.yaml

```
- role: Approver
  description: >-
    This role allows the user to approve access to clients. Users with this role can grant Examiner or Investigator role on clients to other users.
  grantable_on: Application
  permissions:
    - clients.approve
    - artifacts.viewer
    - hunts.approve

# Also include Viewer Role so approvers can login.
- application.login
- clients.search
- hunts.view
```
Clients

The Rekall Agent software consists of two parts, the server code runs in the cloud on App Engine, and the client code is deployed on individual endpoints. In this document, we refer to the client as the endpoint software.

The client code is broken up into a set of small Rekall plugins:

- The **rekall_agent** plugin: This plugin is responsible for communicating with the server, receiving notifications and startup registration.
- The **run_flow** plugin: This plugin runs a flow and stores the results. It is usually called by the rekall_agent plugin for each new flow that is received.

This division is useful if the agent needs to be integrated into another endpoint management system. The endpoint system can be used to run flows by providing the appropriate JSON object to the run_flow plugin, bypassing the rekall agent server itself.

The client preserves local state using two files:

1. The configuration file. Contains general configuration for the client, such as which server to connect to and where to store the **writeback** file.
   a. This file is typically created by the administrator when deploying the agent, and it is not changed by the Rekall Agent.

2. The writeback file contains local client state which the client writes periodically.
   a. This file contains the client's private key, which is also its client ID. The key is used to sign all client messages.
   b. The writeback also records currently running flows allowing the client to recover from crashes.

Client Labels

To better target hunts, Rekall uses client labels. There are three types of client labels:

1. **Intrinsic labels** are baked into the client in its configuration file. These labels are immutable (cannot be changed) and usually set by the administrator during deployment. Intrinsic labels are usually used to group machines by general classes such as servers, laptops, departments and sub organizations.

2. **Custom Labels** are assigned to clients by the administrator as needed. These can be changed at will using the GUI or API (i.e. added or removed). Custom Labels are typically used to group machines into dynamic groups - such as all machines involved in a specific investigation.

3. **Automated labels** are labels automatically assigned by the system based on the client properties. For example, the **Windows** label is assigned to all windows machines.
Automated Labels are never written but can be used to target general groups of machines - e.g. all Windows machines.

Labels can be seen in the client search screen. It is also possible to add or remove labels from a set of clients from this screen.

First, select the clients you wish to label, then click the "label" button.

The dialog shows tri-state buttons for all existing labels. Clearing the checkbox will remove the label from all selected clients, Checking the checkbox will add it and deactivating the checkbox (shown with a horizontal line) will leave that label unchanged. A new label may also be added to all selected clients.

Note that this dialog can only manipulate Custom Labels (maintained in the server database). Intrinsic and Automated labels are left unchanged.

It is also possible to manipulate labels via the API. This is most useful when performing larger, bulk update operations. See the API section below for more information.

Searching and viewing clients.

Clients can be searched using the UI. The search query can take on the following forms:

- Bare name would search for a client's hostname prefix. Searching for XYZ will retrieve hostnames starting with XYZ.
- Prefixing the term with a C. (For example, C.12345) will retrieve exactly the specified client id.
- Prefixing with user:XYZ retrieves machines that report that user account.
● Prefixing with *label:*XYZ retrieves machines that have that client label. The label can be Intrinsic, Automated or Custom.

Currently the UI is limited to searching for 1000 clients at a time. Each time a search is performed, all the results are stored in the browser table. This allows the result to be further filtered or sorted using the usual mechanism. Specifying more than one search term at the same time will further limit the search.

Permissions and Access Control

In order to search for clients, a user must have the "clients.search" permissions. This permission is given by the application "Viewer" role.

Examining a client's flows however, is considered more sensitive as it reveals investigative information about the client and exposes collected data. Therefore, a user must have either the "Examiner" (provides read only access) or "Investigator" (allows launching new flows) role on each client. Note that the same user may have different roles granted on different clients.

Usually users do not possess these roles by default, and so must request the relevant role via the Client approval workflow. If a user attempts to view the client's flows, and they do not have the required permission, the GUI directs them to request approval from another user and specify their reason for accessing this information, as well as the required role.

![Client approval workflow](image)

Not all users can grant client approval. Only users with the "Approver" role are allowed to grant client approval. Those users will get an email notification as well as an indication in their inbox.
They may then approve access to the client. This notifies the requester and allows them to view the client's collected data:

The above client approval process is used by the system to grant roles to users. The administrator may view all the roles granted (as well as remove and add grants) by clicking on the Users menu.
In the above screenshot, we see that approver@example.com has the Approver role granted, while test@example.com has the Auditor role (Can view audit logs) and Hunter role (Can propose hunts). We also see that this user has Examiner role (read only access to data) on several hunts and an Investigator role (can launch new flows) on a client.

A user with administrative access to the App Engine application automatically has all permissions and so they do not need to request client approval.

Auditing

One of the critical aspects of the application is to audit access to protected resources. Rekall has a detailed audit log generated for each security critical action. For example, when a flow is created, a hunt is proposed or approved etc. Additionally, if a resource is accessed via a token, the application logs the delegating user as well as the token used.

To view the audit log, simply click on the "Audit" menu action. It is possible to filter the set of audit events returned using the filter expression. Currently, the following syntax is supported:

- Bare search term matches a user. For example, to view all events from test@example.com.
- Prefixing by type: only retrieves events of the specified type. For example, `type:CollectionDownload` will show all users who downloaded collections.
Audit event logs can also be retrieved using the API (just like any other part of the application). Each event stores critical information about the event in a structured way. The JSON object for each event can be seen by clicking on the event in the table, and that is the same JSON object which is returned using the API.

To access audit event logs, the user must have the "Auditor" role. The permission can also be delegated using a token for automated retrieval of audit logs. Note that accessing the audit logs is an auditable event, and therefore will generate an additional entry!

Flows

A flow is a request sent to the client to run some Rekall plugins, collect their results and capture any generated or uploaded files to the server. The flow is the container for collections and uploaded files - the server does not analyze the results in any way, it just stores them and manages viewing and export of the files.

In the below example, we launch the **APIPsList** flow on our client. We first search for the client in the UI.

Then we click the "View Flows" button to see all the flows recently ran on this client (Note that normally non-admin users would need client approval but we skip this step for brevity).
We now select "Add Flow" to add a new plugin flow to the client.

We are presented with a table containing all known Rekall plugins. We can search the table for the APIPsList flow (which lists processes using the APIs).

We can specify any plugin parameters here. Opening the session allows us to specify session parameters for the Rekall session that is running the plugin.

Rekall offers many plugins, documented through the Rekall Plugin Reference. However many of these plugins are best used interactively or under specialized conditions. The most useful plugins are the "osquery" plugin and the "search" plugin which provide the mechanism for issuing queries to the client.
Note the flow precondition input box. If this is filled with an EFiter query, the query will be run before the flow is executed by the client. The flow will only be executed if the query returns any results. This mechanism allows flows to be conditioned on arbitrary things. For example, in this case we will only run the flow (list all processes) if there is a process called "chrome".

Result Collections

As mentioned above, the Rekall Server does no modelling of client data itself. Instead the server is only concerned with storing the results of the plugins which were executed on the client. A result collection is a JSON file containing a set of tables. Each table specifies several columns and data stored in each column.

Each Rekall plugin produces a table of results. Each cell in the table contains a Rekall object. Normally when running Rekall from the console, the cells are rendered using the text renderer (e.g. using the flag --format text at the Rekall command line). However, Rekall also supports a more detailed JSON representation of each object using the DataExportRenderer (i.e. using the --format data at the Rekall command line).

The Rekall Agent collects both representations into the result collection. Any logging messages are also collected into a separate "log" table too.
Although the Rekall Agent server does not process the result collections, the application allows users to inspect the collection in their browsers. In this context, users can view both the text representation, as well as the richer Rekall data export representation.

The example below shows the output from the APIPsList plugin we ran in the previous example. First, we view the list of flows run on our client (Note we need to have at least an Examiner Role on this client). We then click the collections icon:

The two tabs show the different tables present in the collection (In this case the data and logging). Note that it is possible to filter and sort any of the columns (This is done in the browser so it is very fast).

Rekall also exports object specific information for each object in the table. For example, for the proc column, Rekall exports the command line args, environment strings, linked DLLs and much more. You can see this extra information (again as a JSON object) by clicking on each process.

Rekall has specialized encoding for each entity displayed in a table cell. So for example, any time a process is presented, the same fields are exported. If you find that this exports too much information, or to tailor the level of detail exported, simply issue a different EFilter query to more selectively export only some of the fields. For
example, rather than running the `pslist()` plugin, simply run the `search()` plugin with the query "select proc.name, proc.pid from pslist()" to only export the process name and its pid.

It is also possible to export any of the table views into PDF, CSV or even copy into the clipboard. Note that using this method only exports the visible portions of the table. The detailed information for each object is not exported. We will see later how to programmatically retrieve and process the collection.

Uploaded files

If the Rekall plugin emits files (e.g. the `vaddump` plugin produces files containing dumped process memory regions) or when we specify the `also_collect_files` option when launching the flow, the client will upload relevant files emitted by the plugin, and store these in the flow.

The UI allows cursory inspection of the uploaded files via a hex viewer widget, but any more serious inspection of the file (e.g. searching, indexing etc) must be done offline by exporting them first.
Uploaded files are stored and served directly from Google Cloud Storage buckets. The maximum size for uploaded files is 5Tb, although it is not recommended to use such large sizes in practice.

Canned Flows

In our previous EFilter example, we formulated a query to retrieve all processes which had a sqlite file open. Perhaps we are looking for a malware which uses sqlite files to maintain state.

Unfortunately, months later, our colleague might be looking for the same thing, and they will have to figure out the exact incantation again. They might need to consult their notes to remember which malware this was and how to identify it. This is a large time waste! If we could save our flow as a “Pre-Canned Flow” for later use, we can reuse the flow easily!

A Pre-canned flow is a flow we store with sufficient context (such as a description and category keywords) which allows an investigator to refer to a previously useful flow, and re-use the same flow in a new context with a single click.

In order to create a pre-canned flow, we select one or more previously run flows to be saved (You can combine several existing flows into one pre-canned flow) and then click on the save pre-canned flow toolbar icon:
We can now name it, and provide a description as well as some keywords making it easier to find later.

If in future we wish to use this pre-canned flow, we simply click the **add pre-canned flow** toolbar icon. The table lists all the pre-canned flows in the system (and we can sort and search the table as usual). Selecting the correct flow will automatically add it to the client.

While being able to specify a flexible result collection format (through a user specified EFilter query) may seem like a benefit, in some situations this can lead to longer term difficulties. One example is when a user wants to collect historical information by re-running the same flow or hunt periodically, exporting each of these and inserting into a common database table.

If each flow/hunt produces slightly different result collections, it becomes difficult to coerce all the data into a single database table (since each result collection might have different fields).

In this case it is wise to store the flow/hunt into a single Pre-Canned flow - this saves the EFilter query so it can be used between invocations. Therefore, the pre-canned
flow itself becomes a definition for a long term stable schema. Ensure that only flows that were created from this pre-canned flow are exported and collated to make sure that data remains compatible across time.

Exporting a Flow

Ultimately users wish to do something with the data they collected using Rekall. The Rekall Agent Server manages the information collected which is stored in the cloud. A single flow will potentially contain multiple result collections and multiple uploaded files.

To export one or more flows, select them in the flow view table, then click the "Export" button.

Rekall now creates a shell script which can be run to retrieve all the flow's components directly from the cloud. The shell script must access the data without further authentication, and therefore the server mints special tokens to allow the script access to the flow's results.

The export script includes tokens that permit access to this flow for a limited time (by default 30 minutes). The script does not require any further authentication - this means that if the script is compromised the flow's data may be retrieved by unauthorized
parties. The script is equivalent to the entire content of the flows and should be treated with care.

The script appears in an editor window, allowing the user to copy/paste into a terminal or a user may just click the "save" button to save the script locally.

Note that to access a flow's data, the user must have the "Examiner" or "Investigator" role on the client. Usually this role is assigned via the peer approval workflow described previously.

Hunts

While the Rekall agent can run interesting flows on specific clients, this ability is most useful when run at scale against many clients. The ability to run a flow on many clients is termed a Hunt. Hunts are typically used to search for an anomaly across the entire fleet.

Hunts are launched from one or more existing flows. This ensures that the flows we intend to run across the fleet are properly tested at least once.

For this example, suppose we want to run our clever flow on all machines in the fleet. From the client's flow screen, we highlight one or more flows (multiple flows can be combined in the same hunt) and select "Propose a hunt" button:

When proposing a hunt, we need to specify a list of labels we wish to target and an approver to approve it. This creates a new hunt in the "Proposed" state. The hunt is not active but it appears in the hunts table and can be examined. Additionally, a notification is sent to the approver.

Once the approver approves the hunt, it becomes active and all clients with that label respond. We can see how the hunt is progressing using the Hunts view, showing which clients have participated in this hunt, and their status. You can drill down into each client to view the files and collections uploaded by that specific client.
To view a hunt, the user must be granted the **Viewer** role on this hunt. This will allow them to view all files collected by the hunt and export them.

**Installation.**

The software is available under the GPL licence on the rekall-innovations github organization.

**Client installation**

The Rekall agent is implemented as a single plugin within the regular Rekall command line tool. The plugin is called "agent" and takes a single argument which is its configuration file. Therefore, for the Rekall Agent client to run it must have Rekall installed and a valid configuration file.

Rekall offers installer packages for both Windows and Linux (debian packages). Additionally, Rekall is also available from PyPi, making it easily installable via the pip installer.

**Windows Installation**

For Windows, it is easiest to download the binary Rekall installer from the Releases page. This binary is built using PyInstaller, making it suitable to run on a Windows system without needing external dependencies.

The installer simply copies the Rekall binaries into `C:\ Program Files\ Rekall`. There are no permanent changes to the registry and so if you need to repackage Rekall in a different way (e.g. for distribution via a Zip file), simply copy this directory in its entirety.

**Ubuntu Linux Installation**

For debian based systems (such as Ubuntu), Rekall offers a debian package. The package is built with dh-virtualenv and so it is self-contained and does not require any dependencies to be installed on the system. The virtual environment is completely contained in the Debian package in `/usr/share/rekall-forensic/`. 

<table>
<thead>
<tr>
<th>View</th>
<th>Created</th>
<th>State</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sat, 29 Jul 2017 03:38:02 GMT</td>
<td>Started</td>
<td>Total 0 (0 files)</td>
</tr>
<tr>
<td></td>
<td>Fri, 21 Jul 2017 05:36:47 GMT</td>
<td>Stopped</td>
<td>Total 0 (0 files)</td>
</tr>
<tr>
<td></td>
<td>Fri, 21 Jul 2017 05:43:33 GMT</td>
<td>Started</td>
<td>Total 0 (0 files)</td>
</tr>
<tr>
<td></td>
<td>Fri, 21 Jul 2017 05:49:33 GMT</td>
<td>Started</td>
<td>Total 0 (0 files)</td>
</tr>
<tr>
<td></td>
<td>Fri, 21 Jul 2017 05:58:33 GMT</td>
<td>Started</td>
<td>Total 0 (0 files)</td>
</tr>
</tbody>
</table>

29
Client configuration and deployment.

When deploying the Rekall Agent for the first time, a system configuration is generated within the database. You can always edit the configuration by hand (or if deleted, a new configuration will be generated). The configuration controls both the server and client. For example, the configuration file needs to specify a URL to connect to, and a shared deployment secret.

Polling mode

Typically, agent based systems use polling to task the clients. For example, GRR uses a 10-minute polling interval by default. Choosing the polling interval is a compromise between client responsivity and server resources.

If the polling interval is reduced, the client contacts the server more frequently and it is therefore more responsive, but with a large deployment the load on the server increases (In a cloud deployment this might also result in more cost since more server resources are utilized).

Rekall can optionally use the Firebase real time database to notify clients of pending messages. Firebase is a scalable cloud service designed to cater for millions of users on mobile and web platforms. They offer a real-time database with server side push notifications.

When used with Rekall, the client starts a listening thread to monitor its own queue inside the Firebase database. When the server schedules new flows targeting the client, the server also updates the Firebase database for the client - which, if the client is watching this entry, will notify the client of a change in its state via a server side push notifications.

We do not actually store any data in the Firebase database, we simply use it as an elaborate notification mechanism. The client still contacts the correct Rekall agent server to receive tasking information when it notices a change in the Firebase database.

Development workflow

We have seen above how the Rekall Agent can be deployed in production on the Google Cloud Platform. However, sometimes it is convenient to be run the rekall server locally on our development workstation. This is useful when developing the client or the server component.

AppEngine provides a development server which runs the Rekall Agent Application locally and emulates much of the Google Cloud Platform. The nice thing about this development server is the ability to quickly login and logout arbitrary users - this helps in testing how the application deals with access controls.

To start a local development server, simply install the Google Cloud Platform SDK and launch the development server:
Configure the clients to talk with the local server by creating a client configuration file:

```yaml
__type__: Configuration
client:
  __type__: GAEClientPolicy
  # Baked in labels for client.
  labels:
    - MySpecialClients
    - ExtraSuspicious
  manifest_location:
    __type__: HTTPLocation
    # Local address of development server's client control handler.
    base: http://127.0.0.1:8080/api/control
    path_prefix: /manifest
  # Persistent path to keep the same client.
  writeback_path: /home/scudette/tmp/rekall_agent_test.json
```

We can now start the client using this configuration file (Note that the agent is just a plugin of the normal Rekall command line tool):

```bash
(Dev) scudette:~/rekall$ rekall -v agent --agent_config ~/.rekall-agent.yaml
2017-07-28 13:32:15,841:DEBUG:rekall.1:Logging level set to 10
2017-07-28 13:32:15,845:DEBUG:rekall.1:Running plugin (agent) with args (()) kwargs ({'verbosity': 1})
2017-07-28 13:32:15,861:DEBUG:urllib3.util.retry:Converted retries value: 10 -> Retry(total=10, connect=None, read=None, redirect=None, status=None)
2017-07-28 13:32:15,861:DEBUG:urllib3.util.retry:Converted retries value: 10 -> Retry(total=10, connect=None, read=None, redirect=None, status=None)
127.0.0.1
```

```
2017-07-28 13:32:15,938:DEBUG:rekall:You are not root. It is likely that some operations may not be available.
```

```
(Dev) scudette:~/rekall$ dev_appserver.py . --storage_path ~/tmp/
INFO 2017-07-28 20:25:23.619 api_server.py:313] Starting API server at:
  http://localhost:53596
INFO 2017-07-28 20:25:23.635 dispatcher.py:226] Starting module "default" running at:
  http://localhost:8080
  http://localhost:8000
```
In the above debug output, we see the client is uploading its startup message and then proceeds to poll the control handler for jobs to run.

**API access**

While the GUI is nice to use, it is important to be able to automate some of the tasks. The Rekall server's design exposes a complete featured API (in fact the GUI uses that same API so there is no functionality difference between using the GUI and the raw API).

To call the API one typically makes a HTTP request to an API endpoint (i.e. a URL). Parameters can be provided either with GET or POST. The responses are well structured JSON objects which may be processed in any modern language.

You can see the different API endpoints exposed through the "API" menu. The table shows a short description of any method, the parameters it expects and their descriptions. This table is searchable and sortable so you can quickly narrow down the APIs of interest.

You can now use the API playground to launch specific API calls through the browser. For example, search for clients.
The API’s results are just JSON objects which may be consumed by most modern languages.

Token delegation

As described in the API section above, Rekall exposes a fully featured set of API endpoints for automating access to the application. However, each API endpoint is secured using a specific permission. I.e. only users which possess roles that grant those permissions are able to access the endpoint. In fact, no API endpoint is accessible without at least the "application.login" permission.

Since permissions are granted based on user roles, and user authentication is typically performed using Google OAuth exchange it can be difficult to be able to use the API automatically from a script (One must log in using a valid user account and perform an oauth exchange).

For example, imagine a user wants to write a script to see when a client last checked in. The API table shows the relevant endpoint as /api/client/search with a query parameter of the client id. So, if we run wget with the following URL we should be able to use the API:

```
```

Unfortunately, this request will be rejected because there is no user associated with it, and therefore the server cannot determine if the request is authorized.

The solution to this problem is to grant a delegation. The user is really trying to delegate some of their roles to an automated process to be able to act on the API on their behalf. In Rekall, delegations are done by minting a token, which the automated process may present (A bearer token). This token acts as a shared secret for authentication.
The token is attached to the request, proving to the server that the request is authorized. Rekall examines the token and authorizes the request based on the user that granted it:

```
```

Clearly the delegating user must have the original permission to be able to delegate in the first place, but at the same time, the permissions granted by the delegation may be reduced from the full set of available permissions. If we did not limit the scope of the delegation, and the token was compromised, the adversary would be able to act in the full capacity of the delegator. If the delegating user was an administrator, then the token can grant many more permissions to the attacker than intended (including adding a new user, exporting data etc).

Therefore, each minted token is delegating a role on a resource. In our example above, the script only needs the "client.search" permission. Therefore, a **Viewer** role is sufficient. If the token is compromised it cannot be used to do much more than search for clients.

Consider another example: The user `alice@example.com` has the role **Investigator** on client `C.a9c334b292f4ad2b` (i.e. she can issue new flows on this client). She wants to use a script to download the data from that client. She mints a token with an **Examiner** role on the resource `/C.a9c334b292f4ad2b`, but going to the API viewer and clicking the "Mint a Token" button.

This token can only be used to view data from the specific client `C.a9c334b292f4ad2b` - even if Alice is also authorized to investigate other clients!

Note that tokens are only valid for a fixed time period after which they expire. It is recommended that token expiration be set as short as possible. Additionally, it is impossible to mint new tokens using a token access - tokens can only be minted by a properly logged in user.

Note that the system may issue tokens on behalf of the user in certain situations - e.g. when the user requests to download all the collections from a flow or hunt. In this case, they do not need the Delegator access, instead the system checks they have the Viewer access on the hunt.
Not every user is able to mint tokens for delegations. Users must have the **Delegator** role before they can mint tokens. This restriction does not apply for minting export tokens (for exporting flows and hunts) which are minted by the system on behalf of the user.

**Notes about security**

The Rekall Server application is divided into two parts. The first part is the GUI which produces the visible HTML and Javascript you can see in the browser. However, the real business logic of the application is implemented in the API handlers. This approach reduces the security surface of the application, since ultimately the browser makes API calls, receiving JSON objects - and so we only need to secure the API to ensure proper access controls are enforced in all cases.

Cross Site Request Forgery (CSRF) protection is provided by requiring a special CSRF token to be provided to API calls. The token is generated in the UI part and ensures that the request indeed originated from the application itself.

However, when users use automated scripts to access the API they must provide an access token. In this case Rekall considers the access token to be a kind of CSRF token. It is important to keep in mind that the access token (by design) contains all that an attacker needs to gain access to the API, so it should be protected appropriately. If you are going to embed the token in another web application which makes direct API calls, beware that CSRF protection will be disabled and the application may be vulnerable.