GRR Documentation

Release

GRR team

Nov 29, 2017

Contents

GRR Rapid Response is an incident response framework focused on remote live forensics.

It consists of a python client (agent) that is installed on target systems, and python server infrastructure that can manage and talk to clients.

The goal of GRR is to support forensics and investigations in a fast, scalable manner to allow analysts to quickly triage attacks and perform analysis remotely.

GRR is open source (Apache License 2.0) and is developed on GitHub: github.com/google/grr

CHAPTER 1

Contacts

- GitHub issues: github.com/google/grr/issues
- GRR Users mailing list: grr-users
- Follow us on twitter for announcements of GRR user meetups. We use a gitter chat room during meetups.

CHAPTER 2

Table of contents

2.1 What is GRR?

GRR Rapid Response is an incident response framework focused on remote live forensics.

The goal of GRR is to support forensics and investigations in a fast, scalable manner to allow analysts to quickly triage attacks and perform analysis remotely.

GRR consists of 2 parts: client and server.

GRR client is deployed on systems that one might want to investigate. On every such system, once deployed, GRR client periodically polls GRR frontend servers for work. "Work" means running a specific action: downloading file, listing a directory, etc.

GRR server infrastructure consists of several components (frontends, workers, UI servers) and provides web-based graphical user interface and an API endpoint that allows analysts to schedule actions on clients and view and process collected data.

2.1.1 Remote forensics at scale

GRR was built to run at scale so that analysts are capable of effectively collecting and processing data from large numbers of machines. GRR was built with following scenarios in mind:

- Joe saw something weird, check his machine *(p.s. Joe is on holiday in Cambodia and on 3G)*
- Forensically acquire 25 machines for analysis *(p.s. they're in 5 continents and none are Windows)*
- Tell me if this machine is compromised *(while you're at it, check 100,000 of them - i.e. "hunt" across the fleet)*

2.1.2 GRR client features

- Cross-platform support for Linux, OS X and Windows clients.
- Live remote memory analysis using YARA library.

- Powerful search and download capabilities for files and the Windows registry.
- OS-level and raw file system access, using the SleuthKit (TSK).
- Secure communication infrastructure designed for Internet deployment.
- Detailed monitoring of client CPU, memory, IO usage and self-imposed limits.

2.1.3 GRR server features

- Fully fledged response capabilities handling most incident response and forensics tasks.
- Enterprise hunting (searching across a fleet of machines) support.
- Fast and simple collection of hundreds of digital forensic artifacts.
- AngularJS Web UI and RESTful JSON API with client libraries in Python, PowerShell and Go.
- Powerful data export features supporting variety of formats and output plugins.
- Fully scalable back-end capable of handling large deployments.
- Automated scheduling for recurring tasks.
- Asynchronous design allowing future task scheduling for clients, designed to work with a large fleet of laptops.

2.2 Quickstart (have GRR running in 5 minutes)

This page describes how to get GRR clients and server components up and running for the first time. If you have Docker installed, and just want to experiment with a container that already has GRR set up, you can follow the instructions given here. It takes ~2 minutes to download the image and initialize the server.

To start, install the GRR server deb as described here.

On successful installation, you should have an admin interface running on port 8000 by default. Open the Admin UI in a browser, navigate to Manage Binaries -> Executables (details here) and download the installer you need.

Next, run the client installer on the target machine (can be the same machine as the one running the GRR server components) as root:

- For Windows you will see a 32 and 64 bit installer. Run the installer as admin (it should load the UAC prompt if you are not admin). It should run silently and install the client to `c:\windows\system32\grr\%version%\`. It will also install a Windows Service, start it, and configure the registry keys to make it talk to the URL/server you specified during repack of the clients on the server.
- For OSX you will see a pkg file, install the pkg. It will add a launchd item and start it.
- For Linux you will see a deb and rpms, install the appropriate one. For testing purposes you can run the client on the same machine as the server if you like.

After install, hit Enter in the search box in the top left corner of the UI to see all of your clients that have enrolled with the server. If you don't see clients, follow the troubleshooting steps.

2.3 Installing GRR Server

2.3.1 Overview

Depending on your use case, there are multiple ways of installing the GRR server components. Most users will want to install a release deb. The server deb includes client templates that can be downloaded from GRR's Admin UI after installation.

Alternatively, GRR can be installed from PIP.

You can install also GRR from source as described here. You should then be able to modify GRR code and build your own server/client installers.

GRR components

GRR consists of a client and a few server components.

Whenever GRR server is installed using one of the methods mentioned above, all the server components are installed. This is a simple way to setup smaller GRR server installations: each component will run as a separate server process on the same machine. However, when scaling GRR server to run on multiple machines, each component can exist on its own machine in the data center and run separately.

GRR components are:

Client

The GRR client **is not a server component** but it comes bundled with GRR server. GRR client is deployed on corporate assets using the usual mechanism for software distribution and updates (e.g. SMS, apt). The client communicates with the front-end server using a HTTP POST requests. GRR client sends and receives GRR messages from the server. All communication with the front end servers is encrypted.

Datastore

The data store acts both as a central storage component for data, and as a communication mechanism for all GRR server components.

Front End Servers

The front end servers' main task is to decrypt POST requests from the client, un-bundle the contained messages and queue these on the data store. The front end also fetches any messages queued for the client and sends them to the client.

Worker

In order to remain scalable, the front end does not do any processing of data, preferring to offload processing to special worker components. The number of workers can be tuned in response to increased workload. Workers typically check queues in the data stores for responses from the client, process those and re-queue new requests for the clients (See Flows and Queues).

Web UI

GRR Web UI is the central application which enables the incident responder or forensic analyst to interact with the system. It allows for analysis tasks to be queued for the clients, and results of previous stored analysis to be examined. It also acts as an API endpoint: GRR API can be used for automation and integration with other systems.

2.3.2 Securing access to GRR server (important!)

GRR is a powerful system. However, with power comes responsibility and potential security risks.

If you're running GRR for anything besides simple demo purposes, it's extremely important to properly secure access to GRR server infrastructure. Because:

1. Anybody who has root access to your GRR server effectively becomes root on all systems running GRR client talking to your GRR server.
2. Anybody who has direct write access to GRR datastore (no matter if it's SQLite or MySQL) effectively becomes root on all systems running GRR client talking to your GRR server.

Consequently, it's important to secure your GRR infrastructure. Please follow a security checklist below.

GRR Security Checklist

1. Generate new CA/server keys on initial install. Back up these keys somewhere securely (see Key Management).
2. Maximally restrict SSH access (or any other kind of direct access) to GRR server machines.
3. Make sure GRR web UI is not exposed to the Internet and is protected.

For a high security environment:

1. Make sure GRR's web UI is served through an Apache or Nginx proxy via HTTPS. If you're using any kind of internal authentication/authorization system, limit access to GRR web UI when configuring Apache or Nginx. See user authentication documentation.
2. If there're more than just a few people working with GRR, turn on GRR approval-based access control
3. Regenerate code signing key with passphrases for additional security.
4. Run the http server serving clients on a separate machine to the workers.
5. Ensure the database server is using strong passwords and is well protected.
6. Produce obfuscated clients (repack the clients with a different *Client.name* setting)

2.3.3 Installing from a release server deb (recommended)

This is the recommended way of installing the GRR server components. GRR server debs are built for Ubuntu Xenial. They may install on Debian or other Ubuntu versions, but compatibility is not guaranteed.

To start, download the latest server deb from https://github.com/google/grr/releases, e.g:

```
wget https://storage.googleapis.com/releases.grr-response.com/grr-server_3.2.0-1_
↪amd64.deb
```

Install the server deb, along with its dependencies, like below:

```
sudo apt install -y ./grr-server_3.2.0-1_amd64.deb
```

The installer will prompt for a few pieces of information to get things set up. After successful installation, the `grr-server` service should be running:

```
root@grruser-xenial:/home/grruser# systemctl status grr-server
 grr-server.service - GRR Service
   Loaded: loaded (/lib/systemd/system/grr-server.service; enabled; vendor preset:␣
↪enabled)
   Active: active (exited) since Wed 2017-11-22 10:16:39 UTC; 2min 51s ago
     Docs: https://github.com/google/grr
  Process: 10404 ExecStart=/bin/systemctl --no-block start grr-server@admin_ui.
↪service grr-server@frontend.service grr-server@worker.service grr-server@worker2.
↪service (code=exited, status=0/SUCCESS)
 Main PID: 10404 (code=exited, status=0/SUCCESS)
    Tasks: 0
   Memory: 0B
      CPU: 0
   CGroup: /system.slice/grr-server.service

Nov 22 10:16:39 grruser-xenial systemd[1]: Starting GRR Service...
Nov 22 10:16:39 grruser-xenial systemd[1]: Started GRR Service.
```

In addition, administrative commands for GRR, e.g `grr_console` and `grr_config_updater` should be available in your PATH.

2.3.4 Installing from a HEAD DEB

Instructions for installing the latest stable version of GRR can be found here. If you would like to experiment with the newest unstable server deb, you can download it from Google Cloud Storage as follows:

- Download the latest Google Cloud SDK tarball from https://cloud.google.com/sdk/downloads, e.g:

```
wget https://dl.google.com/dl/cloudsdk/channels/rapid/downloads/google-cloud-sdk-180.
↪0.1-linux-x86_64.tar.gz
```

- Extract it to somewhere on your filesystem:

```
tar zxf google-cloud-sdk-180.0.1-linux-x86_64.tar.gz "${HOME}"
```

- Copy the server deb from its folder in GCS to your local machine:

```
$HOME/google-cloud-sdk/bin/gsutil cp gs://autobuilds.grr-response.com/_latest_server_
↪deb/*.deb .
```

2.3.5 Installing from released PIP packages

If the templates included in release server debs are not compatible with the platform you would like to run them on, you have the option of installing GRR from PIP on your target platform, then building your own installers.

First, install the prerequisites:

- Ubuntu:

```
apt install -y debhelper dpkg-dev python-dev python-pip rpm zip
```

- Centos:

```
yum install -y epel-release python-devel wget which libffi-devel \
  openssl-devel zip git gcc gcc-c++ redhat-rpm-config

yum install -y python-pip
```

Next, upgrade pip and install virtualenv:

```
sudo pip install --upgrade pip virtualenv
```

Next, create a virtualenv and install the GRR server and template packages:

```
virtualenv GRR_ENV

source GRR_ENV/bin/activate

pip install grr-response-server

pip install --no-cache-dir -f https://storage.googleapis.com/releases.grr-response.
↪com/index.html grr-response-templates
```

During installation of `grr-response-server`, administrative commands e.g `grr_console` and `grr_config_updater` will be added to the virtualenv. After installation, you will need to initialize the GRR configuration with `grr_config_updater initialize`. Once that is done, you can build a template for your platform with:

```
grr_client_build build --output mytemplates
```

and repack it with:

```
grr_client_build repack --template mytemplates/*.zip --output_dir mytemplates
```

If you would like to experiment with the Admin UI or other server components, you can launch them from within the virtualenv as follows:

```
grr_server --component admin_ui --verbose
```

Note that GRR requires Python 2.7+, so for platforms with older default Python versions (e.g Centos 6), you need to build a newer version of Python from source and use that for creating the virtualenv.

2.3.6 Installing from source

Here's how to install GRR for development (from github HEAD):

First, install the prerequisites:

- Ubuntu:

```
sudo apt install -y fakeroot debhelper libffi-dev libssl-dev python-dev \
  python-pip wget openjdk-8-jdk zip git devscripts dh-systemd dh-virtualenv \
  libc6-i386 lib32z1 asciidoc
```

- Centos:

```
sudo yum install -y epel-release python-devel wget which java-1.8.0-openjdk \
  libffi-devel openssl-devel zip git gcc gcc-c++ redhat-rpm-config rpm-build \
  rpm-sign
```

```
sudo yum install -y python-pip
```

Next, upgrade pip and install virtualenv:

```
sudo pip install --upgrade pip virtualenv
```

Next, download the github repo and cd into its directory:

```
git clone https://github.com/google/grr.git

cd grr
```

If protoc is already installed, make sure it is present in the PATH, or set the environment variable PROTOC to the full path of the protoc binary.

If protoc is not installed, download it with:

```
travis/install_protobuf.sh linux
```

Finally, create a virtualenv at $HOME/INSTALL and install GRR in the virtualenv:

```
virtualenv $HOME/INSTALL

travis/install.sh
```

You should now be able to run GRR commands from inside the virtualenv, e.g:

```
source $HOME/INSTALL/bin/activate

grr_config_updater initialize # Initialize GRR's configuration
```

2.3.7 Installing via GRR Docker image

The GRR Docker image is is currently intended for evaluation/testing use, but the plan is to support simple cloud deployment of a stable production image in the future.

The instructions below get you a recent stable docker image. We also build an image automatically from the latest commit in the github repository which is more up-to-date but isn't guaranteed to work. If you want bleeding edge you can use `grrdocker/grr:latest` in the commands below.

How to use the image

```
docker run \
  -e EXTERNAL_HOSTNAME="localhost" \
  -e ADMIN_PASSWORD="demo" \
  --ulimit nofile=1048576:1048576 \
  -p 0.0.0.0:8000:8000 -p 0.0.0.0:8080:8080 \
  grrdocker/grr:v3.2.0.1 grr
```

Once initialization finishes point your web browser to localhost:8000 and login with admin:demo. Follow the final part of the quickstart instructions to download and install the clients.

EXTERNAL_HOSTNAME is the hostname you want GRR agents (clients) to poll back to, "localhost" is only useful for testing.

ADMIN_PASSWORD is the password for the "admin" user in the webui.

`ulimit` makes sure the container doesn't run out of filehandles, which is important for the sharded SQLite DB.

The container will listen on port 8000 for the admin web UI and port 8080 for client polls.

If you would like the database and logs to persist longer than the life of the container you could use something like (this also adds -d to run it as a daemon):

```
mkdir ~/grr-docker

docker run \
  --name sqlitedb -v ~/grr-docker/db:/var/grr-datastore \
  --name logs -v ~/grr-docker/logs:/var/log \
  -e EXTERNAL_HOSTNAME="localhost" \
  -e ADMIN_PASSWORD="demo" \
  --ulimit nofile=1048576:1048576 \
  -p 0.0.0.0:8000:8000 -p 0.0.0.0:8080:8080 \
  -d grrdocker/grr:v3.2.0.1 grr
```

Note that if you're running boot2docker on OS X there are a few bugs with docker itself that you will probably need to workaround. You'll likely have to set up port forwards for 8000 and 8080 as described here.

Running GRR binaries in the Docker container

When using Docker, GRR gets installed into a virtualenv in `/usr/share/grr-server`. Thus, the easiest way to run any of the GRR binaries inside the Docker container is to activate the virtualenv:

```
source /usr/share/grr-server/bin/activate
```

After that, commands such as `grr_server`, `grr_config_updater`, `grr_console`, etc become available on PATH.

2.3.8 Troubleshooting ("GRR server doesn't seem to run")

This page describes common issues encountered when installing the GRR server components.

ImportError: cannot import name jobs_pb2 or similar

If you see "ImportError: cannot import name jobs_pb2" or a similar error for any other _pb2 file, you need to regenerate the protobuf files. Just run

```
python setup.py build
sudo python setup.py install
```

The upstart/init.d scripts show no output

When I run an init.d script e.g. "/etc/init.d/grr-http-server start" it does not show me any output.

Make sure that the "START" parameter in the corresponding default file, e.g. "/etc/default/grr-http-server", has been changed to "yes".

I cannot start any/some of the GRR services using the init.d scripts

When I run an init.d script e.g. "/etc/init.d/grr-http-server start" it indicates it started the service although when I check with "/etc/init.d/grr-http-server status" it says it is not running.

You can troubleshoot by running the services in the foreground, e.g. to run the HTTP Front-end server in the foreground:

```
sudo grr_server --start_http_server --verbose
```

Any/some of the GRR services are not running correctly

Check if the logs contain an indication of what is going wrong.

Troubleshoot by running the services in the foreground, e.g. to run the UI in the foreground:

```
sudo grr_server --verbose --start_ui
```

Cannot open libtsk3.so.3

error while loading shared libraries: libtsk3.so.3: cannot open shared object file: No such file or directory

The libtsk3 library cannot be found in the ld cache. Check if the path to libtsk3.so.3 is in /etc/ld.so.conf (or equivalent) and update the cache:

```
sudo ldconfig
```

Cron Job view reports an error

Delete and recreate all the cronjobs using GRR console:

```
aff4.FACTORY.Delete("aff4:/cron", token=data_store.default_token)
from grr.server.aff4_objects import cronjobs
cronjobs.ScheduleSystemCronFlows(token=data_store.default_token)
```

Protobuf

Travis jobs for GRR's github repository use this script to install protobuf. Older versions of protobuf may not be compatible with the GRR version you are trying to install.

Missing Rekall Profiles

If you get errors like:

```
Error loading profile: Could not load profile nt/GUID/ABC123...
Needed profile nt/GUID/ABC123... not found!
```

when using rekall, you're missing a profile (see the Rekall FAQ and blogpost for some background about what this means).

The simplest way to get this fixed is to add it into Rekall's list of GUIDs, which is of great benefit to the whole memory forensics community. You can do this yourself via a pull request on rekall-profiles, or simply email the GUID to rekall-discuss@googlegroups.com. Once it's in the public rekall server, the GRR server will download and use it automatically next time you run a rekall flow that requires that profile. If your GRR server doesn't have internet access you'll need to run the GetMissingProfiles function from the GRR console on a machine that has internet access and can access the GRR database, like this:

```
from grr.server import rekall_profile_server
rekall_profile_server.GRRRekallProfileServer().GetMissingProfiles()
```

2.4 Deploying GRR clients

2.4.1 Overview

This document describes getting clients up and running for the first time.

Getting started

Once we've got the GRR server installed and running we'll want to start deploying some clients.

To do so we'll need to:

1. Download the specific client version we need to install on the system.
2. Decide on a deployment method.
3. Perform the deployment and verify the results.

Requirements

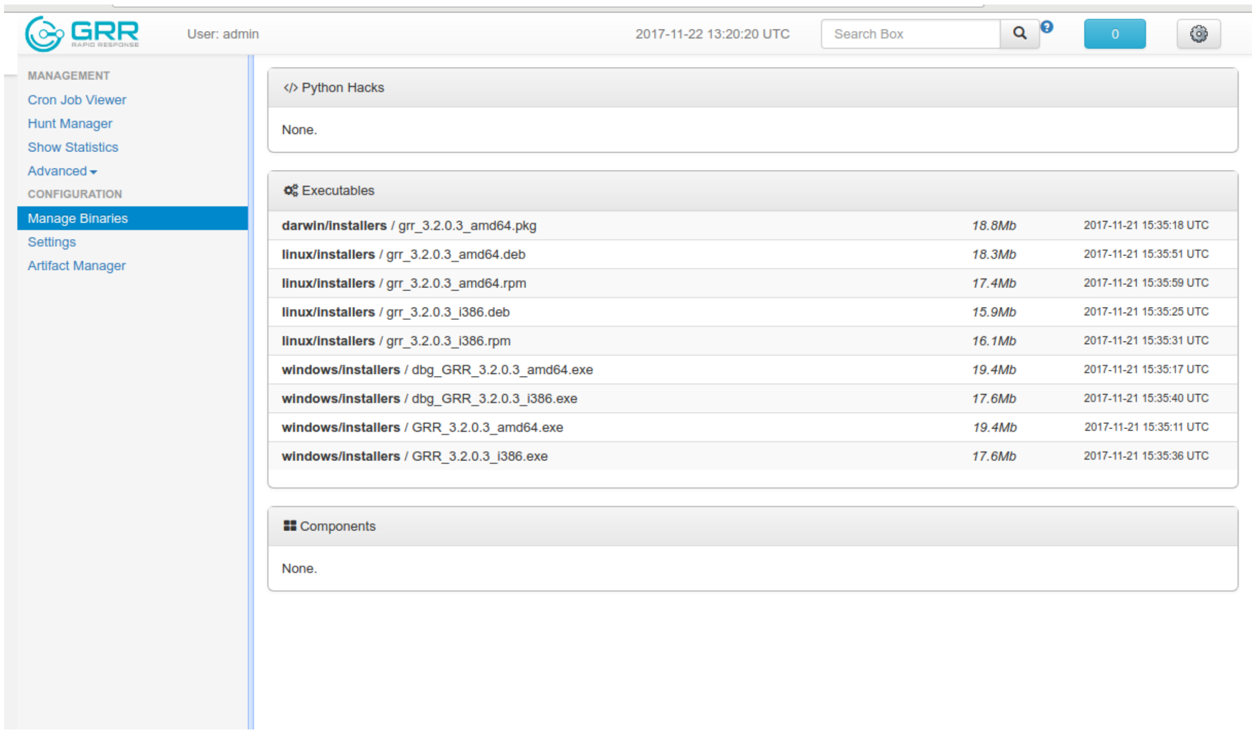
- A number of machines (or VMs) to talk to the server. OSX, Windows and Linux clients are supported. Client and server can run on the same host for testing purposes.

Installing the Clients

Downloading clients

If your server install went successfully, the clients should have been uploaded to the server with working configurations, and should be available in the Admin UI.

Look on the left for "Manage Binaries", the files should be in the executables directory, under installers.



Python Hacks

None.

Executables

darwin/installers / grr_3.2.0.3_amd64.pkg	18.8Mb	2017-11-21 15:35:18 UTC
linux/installers / grr_3.2.0.3_amd64.deb	18.3Mb	2017-11-21 15:35:51 UTC
linux/installers / grr_3.2.0.3_amd64.rpm	17.4Mb	2017-11-21 15:35:59 UTC
linux/installers / grr_3.2.0.3_i386.deb	15.9Mb	2017-11-21 15:35:25 UTC
linux/installers / grr_3.2.0.3_i386.rpm	16.1Mb	2017-11-21 15:35:31 UTC
windows/installers / dbg_GRR_3.2.0.3_amd64.exe	19.4Mb	2017-11-21 15:35:17 UTC
windows/installers / dbg_GRR_3.2.0.3_i386.exe	17.6Mb	2017-11-21 15:35:40 UTC
windows/installers / GRR_3.2.0.3_amd64.exe	19.4Mb	2017-11-21 15:35:11 UTC
windows/installers / GRR_3.2.0.3_i386.exe	17.6Mb	2017-11-21 15:35:36 UTC

Components

None.

If your server configuration has changed your clients will need to be repacked with an updated config. For details see the server documentation.

Installation steps differ significantly depending on the operating system so we split it into separate sections below.

Run the client on the target machine as administrator: Windows instructions, OSX instructions, Linux instructions.

See How to check if a deployed client talks back to the GRR server.

Uninstalling GRR

A quick manual on how to remove the GRR client completely from a machine is included in the platform-specific docs: Windows instructions, OSX instructions, Linux instructions

Notes

Deploying at scale

There shouldn't be any special considerations for deploying GRR clients at scale. If the server can't handle the load, the clients should happily back off and wait their turn. However, we recommend a staged rollout if possible.

Client and Server Version Compatibility and Numbering

We try hard to avoid breaking backwards compatibility for clients since upgrading can be painful, but occasionally we need to make changes that require a new client version to work. As a general rule you want to upgrade the server first, then upgrade the clients fairly soon after.

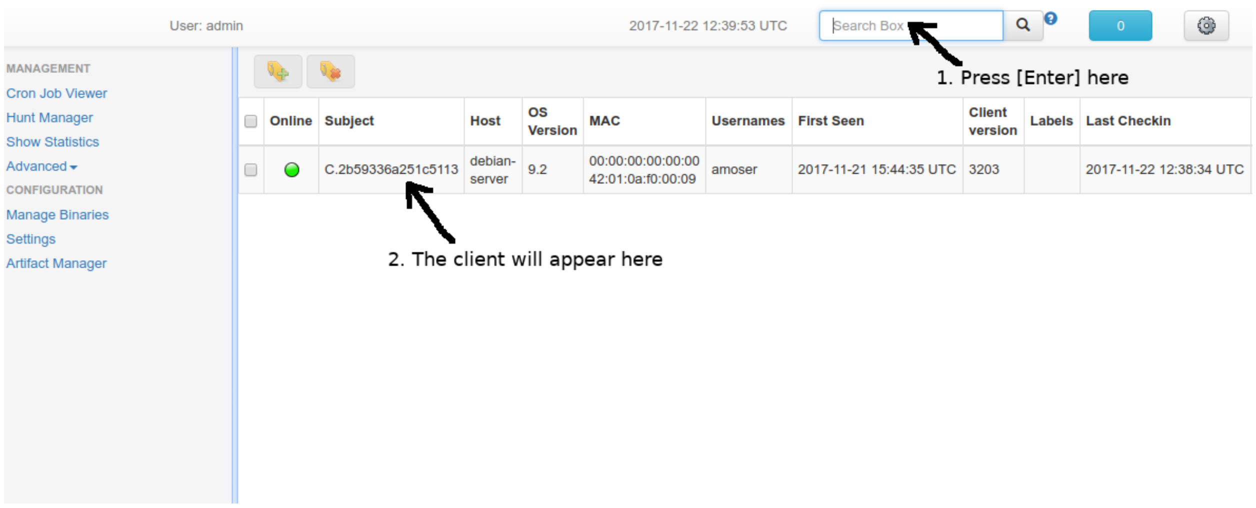
Matching major/minor versions of client and server should work well together. i.e. Clients 3.1.0.0 and 3.1.6.2 should work well with servers 3.1.0.0 and 3.1.9.7 because they are all 3.1 series. We introduced this approach for the 3.1.0.0 release.

For older servers and clients, matching the last digit provided similar guarantees. i.e. client 3.0.0.7 was released with server 0.3.0-7 and should work well together.

2.4.2 How to check if a deployed client talks back to the GRR server

If a client installation is successful, the client should appear in the web UI search within a few seconds.

After install, hit Enter in the search box in the top left corner of the UI to see all of your clients that have enrolled with the server.



	Online	Subject	Host	OS Version	MAC	Usernames	First Seen	Client version	Labels	Last Checkin
☐	●	C.2b59336a251c5113	debian-server	9.2	00:00:00:00:00:00 42:01:0a:f0:00:09	amoser	2017-11-21 15:44:35 UTC	3203		2017-11-22 12:38:34 UTC

If you don't see the clients, follow the troubleshooting steps.

2.4.3 Installing GRR clients on Windows

For Windows you will see a 32 and 64 bit installer. Run the installer as admin (it should load the UAC prompt if you are not admin). It should run silently and install the client to `c:\windows\system32\grr\%version%\`. It will also install a Windows Service, start it, and configure the registry keys to make it talk to the URL/server you specified during repack of the clients on the server.

The Windows clients are special self extracting zipfiles. Just double click or otherwise execute the binary. If you are not an administrator it will prompt you for credentials. It should then install silently in the background, unless you enabled the verbose build.

The most straightforward way to deploy a GRR client to a Windows machine is to use PsExec. PsExec allows one to execute commands on a remote system if credentials for a valid user are known.

To do so, start by downloading psexec and placing in a directory of your choice, we'll call it CLIENT_DIRECTORY here. Store the version of the client you want to download to the same directory.

Once you have both, you have to make sure you know the username and password of an Administrator user in the remote system. Once all these requirements are met, just start a cmd.exe shell and type:

```
cd C:\CLIENT_DIRECTORY\
net use \\MACHINE\IPC$ /USER:USERNAME *
psexec \\MACHINE -c -f -s client-version.exe
```

Note: The `NET USE` command will ask for a password interactively, so it's not suited for using in scripts. You could Switch the `*` for the PASSWORD instead if you want to include it in a script.

You'll need to replace:

- C:\CLIENT_DIRECTORY\ with the full path you chose.
- MACHINE with the name of the target system.
- USERNAME with the user with administrative privileges on the target system.

This will copy the client-version.exe executable on the target system and execute it. The installation doesn't require user input.

The expected output is something along these lines:

```
C:\> cd C:\CLIENT_DIRECTORY\
C:\> net use \\127.0.0.1\IPC$ /USER:admin *
Type the password for \\127.0.0.1\IPC$:
The command completed successfully

C:\CLIENT_DIRECTORY> psexec \\127.0.0.1 -c -f -s client.exe
PsExec v1.98 - Execute processes remotely
Copyright (C) 2001-2010 Mark Russinovich
Sysinternals - www.sysinternals.com

The command completed successfully.

client.exe exited on 127.0.0.1 with error code 0.

C:\CLIENT_DIRECTORY>
```

For even less footprint on installation you could host the client on a shared folder on the network and use this psexec command instead:

```
cd C:\CLIENT_DIRECTORY\
net use \\MACHINE\IPC$ /USER:USERNAME *
psexec \\MACHINE -s \\SHARE\FOLDER\client-version.exe
```

This requires the USERNAME on the remote MACHINE be able to log into SHARE and access the shared folder FOLDER. You can do this either by explicitly allowing the user USERNAME on that share or by using an Anonymous share.

The best way to verify whether the whole installation process has worked is to search for the client in the GUI.

Uninstalling on Windows

On Windows the client does not have a standard uninstaller. It is designed to have minimal impact on the system and leave limited traces of itself such that it can be hidden reasonably easily. Thus it was designed to install silently without an uninstall.

Disabling the service can be done with the Uninstall GRR flow, but this does not clean up after itself by default.

Cleaning up the client is a matter of deleting the service and the install directory, then optionally removing the registry keys and install log if one was created.

On Windows, GRR lives in

```
%SystemRoot%\system32\grr\*
```

The service can be stopped with

```
sc stop "grr monitor"
```

Or via the task manager.

The GRR config lives in the registry, for a full cleanup, the path

```
HKEY_LOCAL_MACHINE\Software\GRR
```

should be deleted.

Removing the GRR client completely from a machine:

```
sc stop "grr monitor"
sc delete "grr monitor"
reg delete HKLM\Software\GRR
rmdir /Q /S c:\windows\system32\grr
del /F c:\windows\system32\grr_installer.txt
```

2.4.4 Installing GRR clients on Mac OS X

For OSX you will see a pkg file, install the pkg. It will add a launchd item and start it.

See Linux instructions. They apply also to OSX.

On OSX you can also use the Uninstall GRR flow.

Uninstalling on Mac OS X

This is a quick manual on how to remove the GRR client completely from a machine.

On OSX, pkg uninstall is not supported. The files to delete are:

```
/usr/local/lib/grr/*
/etc/grr.local.yaml
/Library/LaunchDaemons/com.google.code.grr.plist
```

The service can be stopped using

<<<<<<< HEAD sudo launchctl unload /Library/LaunchDaemons/com.google.corp.grr.plist =======

```
sudo launchctl unload /Library/LaunchDaemons/com.google.corp.grr.plist
```

markdown-new

2.4.5 Installing GRR clients on Linux

For Linux you will see a deb and rpms, install the appropriate one. For testing purposes you can run the client on the same machine as the server if you like.

The process depends on your environment, if you have a mechanism such as puppet, then building as a Deb package and deploying that way makes the most sense. Alternatively you can deploy using ssh:

```
scp client_version.deb host:/tmp/
ssh host sudo dpkg -i /tmp/client_version.deb
```

Uninstalling on Linux

This is a quick manual on how to remove the GRR client completely from a machine.

On Linux the standard system packaging (deb, pkg) is used by default. Use the standard uninstall mechanisms for uninstalling.

```
dpkg -r grr
```

This might leave some config files lying around, if a complete purge is necessary, the list of files to delete is:

```
/usr/lib/grr/*
/etc/grr.local.yaml
/etc/init/grr.conf
```

The GRR service can be stopped using

```
sudo service grr stop
```

2.4.6 Life of a GRR client (what happens after deployment)

1. When a new client starts, it notices it doesn't have a public/private key pair.
2. The client generates the keys. A hash of the public key becomes the client's ID. The ID is unique.
3. The client enrolls with the GRR server.
4. The server sees the client ID for the first time & interrogates the new client.

Client Robustness Mechanisms

We have a number of mechanisms built into the client to try and ensure it has sensible resource requirements, doesn't get out of control, and doesn't accidentally die. We document them here.

Heart beat

The client process regularly writes to a registry key (file on Linux and OSX) with a timer. The nanny process watches this registry key called HeartBeat, if it notices that the the client hasn't updated the heartbeat in the time allocated by UNRESPONSIVE_KILL_PERIOD (default 3 minutes), the nanny will assume the client has hung and will kill it. In Windows we then rely on the Nanny to revive it, on Linux and OSX we rely on the service handling mechanism to do so.

Transaction log

When the client is about to start an action it writes to a registry key containing information about what it is about to do. If the client dies while performing the action, when the client gets restarted it will send an error along with the data from the transaction log to help diagnose the issue.

One tricky thing with the transaction log is the case of Bluescreens or kernel panics. Writing to the transaction log will write a registry key on Windows, but registry keys are not flushed to disk immediately. Therefore, writing a transaction log, and then getting a hard BlueScreen or kernel panic, the transaction log won't be persistent, and therefore the error won't be sent. We work around this by adding a Flush to the transaction log when we are about to do dangerous transactions, such as loading a memory driver. But if the client dies during a transaction we didn't deem as dangerous, it is possible that you will not get a crash report.

Memory limit

We have a hard and a soft memory limit built into the client to stop it getting out of control. The hard limit is enforced by the nanny, if the client goes over that limit it will be hard killed. The soft limit is enforced by the client, if the limit is exceeded the client will stop retrieving new work to do. Once it has finished its current work it will die cleanly.

Default soft limit is 500MB, but GRR should only use about 30MB. Some volatility plugins can use a lot of memory so we try to be generous. Hard limit is double the soft limit. This is configurable from the config file.

CPU limit

A ClientAction can be transmitted from the server with a specified CPU limit, this is how many seconds the action can use. If the action uses more than that it will be killed. The actual implementation is a little more complicated. An action can run for 3 minutes using any CPU it wants before being killed by nanny. However actions that are good citizens (normally the dangerous ones) will call the Progress() function regularly. This function checks if limit has been exceeded and will exit.

2.4.7 GRR Client Protection

The open source agent does not contain protection against being disabled by administrator/root on the machine. E.g. on Windows, if an attacker stops the service, the agent will stop and will no longer be reachable. Currently, it is up to the deployer of GRR to provide more protection for the service.

Obfuscation

If every deployment in the world is running from the same location and the same code, e.g. `c:\program files\grr\grr.exe`, it becomes a pretty obvious thing for an attacker to look for and disable. Luckily the attacker has the same problem an investigator has in finding malware on a system, and we can use the same techniques to protect the client. One of the key benefits of having an open architecture is that customization of the client and server is easy, and completely within your control.

For a test, or low security deployment, using the defaults open source agents is fine. However, in a secure environment we strongly recommend using some form of obfuscation.

This can come in many forms, but to give some examples:

- Changing service, and binary names
- Changing registry keys
- Obfuscating the underlying python code
- Using a packer to obfuscate the resulting binary
- Implementing advanced protective or obfuscation functionality such as those used in rootkits
- Implementing watchers to monitor for failure of the client

GRR does not include any obfuscation mechanisms by default. But we attempt to make this relatively easy by controlling the build process through the configuration file.

Enrollment

In the default setup, clients can register to the GRR server with no prior knowledge. This means that anyone who has a copy of the GRR agent, and knows the address of your GRR server can register their client to your deployment. This significantly eases deployment, and is generally considered low risk as the client has no control or trust on the server.

However, it does introduce some risk, in particular:

- If there are flows or hunts you deploy to the entire fleet, a malicious client may receive them. These could give away information about what you are searching for.
- Clients are allowed to send some limited messages to the server without prompting, these are called Well Known flows. By default these can be used to send log messages, or errors. A malicious client using these could fill up logs and disk space.
- If you have custom Well Known Flows that perform interesting actions. You need to be aware that untrusted clients can call them. Most often this could result in a DoS condition, e.g. through a client sending multiple install failure or client crash messages.

In many environments this risk is unwarranted, so we suggest implementing further authorization in the Enrollment Flow using some information that only your client knows, to authenticate it before allowing it to become a registered client.

Note that this does not give someone the ability to overwrite data from another client, as client name collisions are protected.

2.4.8 Troubleshooting ("I don't see my clients")

Debugging the client installation

If the installer is failing to run, it should output a log file which will help you debug. The location of the logfile is configurable, but by default should be:

- Windows: %WinDir%\system32\logfiles\GRR_installer.txt
- Linux/Mac OSX: /tmp/grr_installer.txt

Once you have done this, you can download the new binary from the Web UI. It should have the same configuration, but will output detailed progress to the console, making it much easier to debug.

Note that the binary is also a zipfile, you can open it in any capable zip reader. Unfortunately this doesn't include the built in Windows zip file handler but does include winzip or 7-zip. Opening the zip is useful for reading the config or checking that the right dependencies have been included.

Repacking the Windows client in verbose mode enables console output for both the installer and for the application itself. It does so by updating the header of the binary at PE_HEADER_OFFSET + 0x5c from value 2 to 3. This is at 0x144 on 64 bit and 0x134 on 32 bit Windows binaries. You can do this manually with a hex editor as well.

Interactively Debugging the Client

On each platform, the client binary should support the following options:

- –verbose: This will set higher logging allowing you to see what is going on.
- –debug: If set, and an unhandled error occurs in the client, the client will break into a pdb debugging shell.

```
C:\Windows\system32>net stop "grr monitor"
The GRR Monitor service is stopping.
The GRR Monitor service was stopped successfully.

C:\Windows\system32>c:\windows\system32\grr\3.2.0.4\grr.exe --config grr.exe.yaml --
↪verbose
```

Note that on Windows, the GRR client is running completely in the background so starting it from the console will not give you any output. For this reason, we always also build a dbg_GRR[...].exe client that is a console application and can be used for this kind of debugging.

Changing Logging For Debugging

On all platforms, by default only hard errors are logged. A hard error is defined as anything level ERROR or above, which is generally reserved for unrecoverable errors. But because temporary disconnections are normal, an client failing to talk to the server doesn't actually count as a hard error.

In the client you will likely want to set:

```
Logging.verbose: True
```

And depending on your configuration, you can play with syslog, log file and Windows EventLog logging using parameters `Logging.path`, and `Logging.engines`.

Proxies and Connectivity

If an client can't connect to the server, there can be a number of reasons such as:

- Server Isn't Listening Confirm you can connect to the server and retrieve the server.pem file. E.g.

  ```
  wget http://server:8080/server.pem
  ```

- Proxy Required For Access If the environment doesn't allow direct connections GRR may need to use a proxy. GRR currently doesn't support Proxy Autoconfig or Proxy Authentication. GRR will attempt to guess your proxy configuration, or you can explicitly set proxies in the config file, e.g.

  ```
  Client.proxy_servers: ["http://cache.example.com:3128/"]
  ```

  On Windows systems GRR will try a direct connection, and then search for configured proxies in all users profiles on the system trying to get a working connection. On Linux GRR should obey system proxy settings, and it will also obey environment variables. e.g.

  ```
  $ export http_proxy=http://cache.example.com:3128
  ```

- Outbound Firewall Blocking Connections GRR doesn't do anything to bypass egress firewalling by default. However, if you have a restrictive policy you could add this as an installer plugin.

If you look at the running config, the first time the client successfully connects to the server a variable `Client.server_serial_number` will be written to the config. If that exists, the client successfully made a connection.

Crashes

The client shouldn't ever crash… but it does because making software is hard. There are a few ways in which this can happen, all of which we try and catch, record and make visible to allow for debugging. In the UI they are visible in two ways, in "Crashes" when a client is selected, and in "All Client Crashes". These have the same information but the client view only shows crashes for the specific client.

Each crash should contain the reason for the crash, optionally it may contain the flow or action that caused the crash. In some cases this information is not available because the client may have crashed when it wasn't doing anything or in a way where we could not tie it to the action.

This data is also emailed to the email address configured in the config as `Monitoring.alert_email`

Crash Types

Crashed while executing an action

Often seen with an error “Client killed during transaction”. This means that while handling a specific action, the client died. The nanny knows this because the client recorded the action it was about to take in the Transaction Log before starting it. When the client restarts it picks up this log and notifies the server of the crash.

Causes

- Client segfaults, could happen in native code such as Sleuth Kit or psutil.
- Hard reboot while the machine was running an action where the client service didn’t have a chance to exit cleanly.

Unexpected child process exit!

This means the client exited, but the nanny didn’t kill it.

Causes

- Uncaught exception in python, very unlikely due to the fact that we catch Exception for all client actions.

Memory limit exceeded, exiting

This means the client exited due to exceeding the soft memory limit.

Causes

- Client hits the soft memory limit. Soft memory limit is when the client knows it is using too much memory but will continue operation until it finishes what it is doing.

Nanny Message - No heartbeat received

This means that the Nanny killed the client because it didn’t receive a Heartbeat within the allocated time.

Causes

- The client has hung, e.g. locked accessing network file
- The client is performing an action that is taking longer than it should.

2.4.9 I don’t see my clients

Here are common client troubleshooting steps you can work through to diagnose client install and communications problems.

Check that the client got installed

You should have something like:

- `C:\Windows\System32\GRR\3.1.0.2\GRR.exe` on windows;
- `/usr/lib/grr/grr_3.1.0.2_amd64/grrd` on linux; and
- `/usr/local/lib/grr_3.1.0.2_amd64/grrd` on OSX

with variations based on the GRR version and architecture you installed (32bit is i386). If you didn't get this far, then there is a problem with your installer binary.

Check the client is running

On linux and OS X you should see two processes, something like:

```
$ ps aux | grep grr
root       957  0.0  0.0  11792   944 ?        Ss   01:12   0:00 /usr/sbin/grrd --
↪config=/usr/lib/grr/grr_3.1.0.2_amd64/grrd.yaml
root      1015  0.2  2.4 750532 93532 ?        Sl   01:12   0:01 /usr/sbin/grrd --
↪config=/usr/lib/grr/grr_3.1.0.2_amd64/grrd.yaml
```

On windows you should see a `GRR Monitor` service and a `GRR.exe` process in taskmanager.

If it isn't running check the install logs and other logs in the same directory:

- Linux/OS X: `/var/log/grr_installer.txt`
- Windows: `C:\Windows\System32\LogFiles\GRR_installer.txt`

and then try running it interactively as below.

Check the client machine can talk to the server

The URL here should be the server address and port you picked when you set up the server and listed in `Client.control_urls` in the client's config.

```
wget http://yourgrrserver.yourcompany.com:8080/server.pem
# Check your config settings, note that clients earlier than 3.1.0.2 used Client.
↪control_urls
$ sudo cat /usr/lib/grr/grr_3.1.0.2_amd64/grrd.yaml | grep Client.server_urls
Client.server_urls: http://yourgrrserver.yourcompany.com:8080/
```

If you can't download that server.pem, the common causes are that your server isn't running or there are firewalls in the way.

Run the client in verbose mode

Linux: Stop the daemon version of the service and run it in verbose mode:

```
$ sudo service grr stop
$ sudo /usr/sbin/grrd --config=/usr/lib/grr/grr_3.1.0.2_amd64/grrd.yaml --verbose
```

OS X: Unload the service and run it in verbose mode:

```
$ sudo launchctl unload /Library/LaunchDaemons/com.google.code.grr.plist
$ sudo /usr/local/lib/grr_3.1.0.2_amd64/grrd --config=/usr/local/lib/grr_3.1.0.2_
↪amd64/grrd.yaml --verbose
```

Windows: The normal installer isn't a terminal app, so you don't get any output if you run it interactively.

- Install the debug `dbg_GRR_3.1.0.2_(amd64|i386).exe` version to make it a terminal app.
- Stop the `GRR Monitor` service in task manager

Then run in a terminal as Administrator

```
cd C:\Windows\System32\GRR\3.1.0.2\
GRR.exe --config=GRR.exe.yaml --verbose
```

If this is a new client you should see some 406's as it enrols, then they stop and are replaced with sending message lines with increasing sleeps in between. The output should look similar to this:

```
Starting client aff4:/C.a2be2a27a8d69c61
aff4:/C.a2be2a27a8d69c61: Could not connect to server at http://somehost:port/,␣
↪status 406
Server PEM re-keyed.
sending enrollment request
aff4:/C.a2be2a27a8d69c61: Could not connect to server at http://somehost:port/,␣
↪status 406
Server PEM re-keyed.
sending enrollment request
aff4:/C.a2be2a27a8d69c61: Could not connect to server at http://somehost:port/,␣
↪status 406
Server PEM re-keyed.
aff4:/C.a2be2a27a8d69c61: Sending 3(1499), Received 0 messages in 0.771058797836 sec.␣
↪Sleeping for 0.34980125
aff4:/C.a2be2a27a8d69c61: Sending 0(634), Received 0 messages in 0.370272874832 sec.␣
↪Sleeping for 0.4022714375
aff4:/C.a2be2a27a8d69c61: Sending 0(634), Received 0 messages in 0.333703994751 sec.␣
↪Sleeping for 0.462612153125
aff4:/C.a2be2a27a8d69c61: Sending 0(634), Received 0 messages in 0.345727920532 sec.␣
↪Sleeping for 0.532003976094
aff4:/C.a2be2a27a8d69c61: Sending 0(634), Received 0 messages in 0.346176147461 sec.␣
↪Sleeping for 0.611804572508
aff4:/C.a2be2a27a8d69c61: Sending 0(634), Received 8 messages in 0.348709106445 sec.␣
↪Sleeping for 0.2
```

If enrolment isn't succeeding (406s never go away), make sure you're running a worker process and check logs in `/var/log/grr-worker.log`.

2.5 Investigating with GRR

2.5.1 Overview

Once set up and configured, the GRR user interface is a web interface which allows the analyst to search for connected client (agent) machines, examine what data has been collected from the machines and issue requests to collect additional data.

The GRR server also provides access to this functionality through a JSON API. Client libraries to support scripting from python or go are provided.

Starting Points

Depending on the type of client data that the analyst is interested in, there are several places that they might start.

Virtual File System

The virtual file system shows the files, directories, and registry entries which have already been collect from a client. It shows when the entry was collected, and provides some buttons to collect additional buttons of this sort.

This is a natural starting point for ad-hock examination of an individual machine.

Flows

A Flow performs one or more operations on a client machine, in order to collect or check for data. For example, the data collection buttons shown by the virtual file system start flows to collect specific files and directories. However, flows can do many other things - from searching a directory for files containing a particular substring, to recording the current network configuration. The administrative interface shows for each client the flows which have been launched against it also the flow's status and any results returned.

When an analyst would like to collect a specific bit of information about a machine, they will need to directly or indirectly run a flow.

Hunts

A Hunt is a mechanism to run a Flow on a number of clients. For example, this makes it possible to check if any Windows machine in the fleet has a file with a particular name in a particular location.

Artifacts

An Artifact is a way to collect and name a group of files or other data that an analyst might want to collect as a unit. For example, an artifact might try to collect all the common linux persistence mechanisms.

2.5.2 Client-Server Communication

When a Flow needs to request information from a client, it queues up a message for the client. GRR clients poll the GRR server approximately every 10 minutes, and it will receive the message and begin responding to the request at the next poll.

After a client performs some work, it will normally enter 'fast-poll' mode in which it polls much more rapidly. Therefore when an analyst requests data from a machine, it might initially take some minutes to respond but additional requests will be noticed more quickly.

Protocol

The client poll is an HTTP request. It passes a signed and encrypted payload and expects the same from the GRR server. The client signs using its client key. This key is created on the client when first run, and the GRR ID is actually just a fingerprint of this key.

This means that no configuration is required by the client to establish an identity, but that clients cannot eavesdrop on or impersonate other clients.

2.5.3 Security Considerations

As described in Client-Server Communication, communication between the GRR client and server is secure against eavesdropping and impersonation. Furthermore, the server's record of flows run and files downloaded is not changed after the fact. It is intended to provide a secure archive of data that has been gathered from machines.

Root Privileges

The GRR client agent normally runs as root, and by design is capable of read any data on the system. Furthermore some flows can use quite a lot of resources on client (e.g. searching a large directory recursively) and in fact the GRR agent does have the ability to download and execute server provided code, though the code does need to be signed.

For these reasons, access to the GRR server should be considered tantamount to have root access to all GRR client machines. While we provide auditing and authorization support to mitigate the risks of this, the risks inherient in this should be understood when configuring and using GRR.

2.5.4 Searching For A Client

In order to start interfacing with a client, we first need to search for it in the GUI. The GRR search bar is located at the top of the GUI and allows you to search clients based on:

- **Hostname:** "host:myhost-name"
- **Fully Qualified Domain Name (FQDN):** "fqdn:myhost-name.organization.com", also prefixes of components, e.g. "fqdn:myhost-name.organization"
- **MAC address:** "mac:eeffaabbccdd".
- **IP address:** "ip:10.10.10.10", also prefixes of bytes "ip:10.10". Note that IP address is only collected during interrogate, which by default is run once per week.
- **User:** "user:john"
- **Label:** "label:testmachines". Finds hosts with a particular GRR label.
- **Time of Last Data Update:** Time ranges can be given using "start_date:" and "end_date:" prefixes. The data is interpreted as a human readable timestamp. Examples: start_date:2015, end_date:2018-01-01.

All of these keywords also work without the type specifier, though with less precision. For example "johnsmith" is both a user name and a hostname name, it will match both.

Furthermore there are additional keywords such as OS and OS version. So "Windows" will find all windows machines and "6.1.7601SP1" will match Windows 7 machines with SP1 installed, "6.1.7601" will match those without a service pack.

By default, the search index only considers clients that have checked in during the last six months. To override this behavior, use an explicit "start_date:" directive as specified above.

Interpreting Client Search Results

Searching returns a list of clients with the following information about each one:

- **Online**: An icon indicating whether the host is online or not. Green means online; yellow, offline for some time; red, offline for a long time.
- **Subject**: The client IDentifier. This is how GRR refers internally to the system.
- **Host**: The name of the host as the operating system sees it.
- **Version**: The operating system version.
- **MAC**: A list of MAC addresses of the system.
- **Usernames**: A list of user accounts the operating system knows about (usually users local to the system or that have logged in).
- **First Seen**: The time when the client first talked to the server.

- **OS install time**: The timestamp for the operating system install.
- **Labels**: Any labels applied to this client.
- **Last Checkin**: The last time the client communicated with the server.

Once you've found the client you were looking for, click on it and both the left panel and main panel will change to reflect you're now working with a client.

2.5.5 Flows

GRR Flows

When designing GRR, one of the main goals was achieving great scalability. One of the main resource hogs with the client-server model is that while a client is active all resources that might have been needed on the server side to communicate with it and do processing are held (think temporary buffers, sockets, file descriptors...). Even when the client itself is doing operations that take time such as heavy computations or waiting on I/O, resources are held on the server.

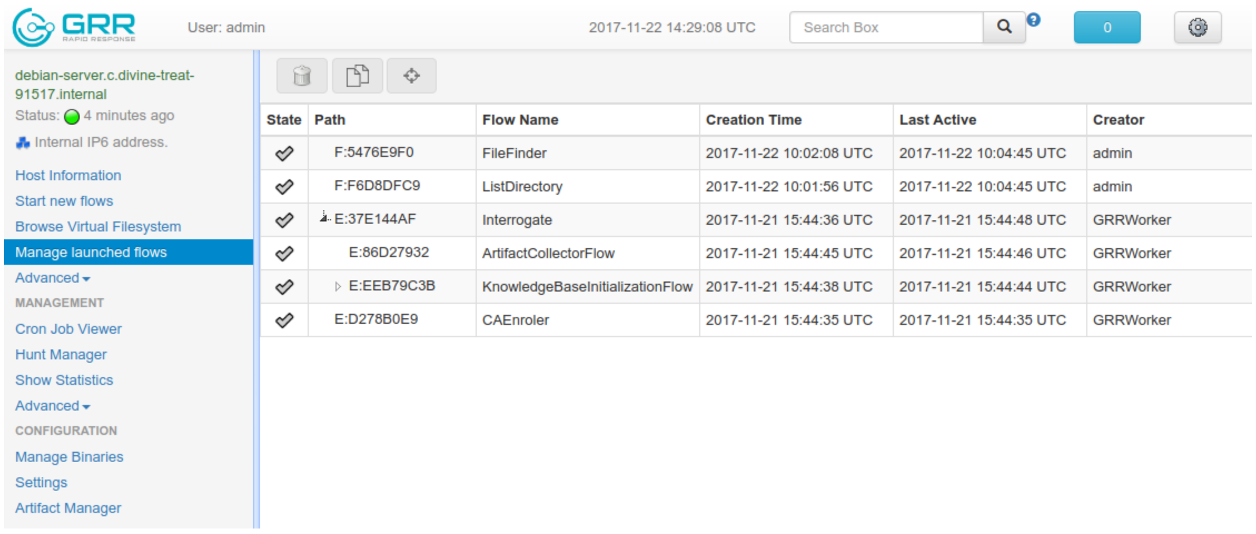
When trying to deal with thousands of clients at the same time, this would translates into the server hoarding many unneeded resources.

To solve the resource hogging problem, Flows were created. Flows are the server-side code entities that call client actions. These calls are done asynchronously. That is, they are requested and their results become available later on. Flows are like a state machine, where transition between states happens when the results of client actions return to the server. So here's what happens when the GRR server launches a typical Flow.

1. The GRR server executes the initial Flow state.
2. This state asks for one or more client actions to be performed on the client.
3. The server clears all the resources this Flow has requested and waits for responses from the client to come back.
4. When responses are received, the server fetches all the needed resources again and runs the Flow state where it expects these responses. If more client actions are requested by this state it goes back to step 2. Otherwise...
5. The results of this Flow are stored and the flow state is updated.

Flows have a second very interesting property. For flows that make use of some of the most primitive client actions, because all of the logic is encapsulated on the server side and the client doesn't have any state at all, they naturally survive reboots while processing is taking place.

In the GRR UI, while having selected a client in the GUI, clicking on the *Manage launched flows* link on the left panel takes you to a view that shows all the Flows that have been requested on this client.



State	Path	Flow Name	Creation Time	Last Active	Creator
✓	F:5476E9F0	FileFinder	2017-11-22 10:02:08 UTC	2017-11-22 10:04:45 UTC	admin
✓	F:F6D8DFC9	ListDirectory	2017-11-22 10:01:56 UTC	2017-11-22 10:04:45 UTC	admin
✓	E:37E144AF	Interrogate	2017-11-21 15:44:36 UTC	2017-11-21 15:44:48 UTC	GRRWorker
✓	E:86D27932	ArtifactCollectorFlow	2017-11-21 15:44:45 UTC	2017-11-21 15:44:46 UTC	GRRWorker
✓	E:EEB79C3B	KnowledgeBaseInitializationFlow	2017-11-21 15:44:38 UTC	2017-11-21 15:44:44 UTC	GRRWorker
✓	E:D278B0E9	CAEnroler	2017-11-21 15:44:35 UTC	2017-11-21 15:44:35 UTC	GRRWorker

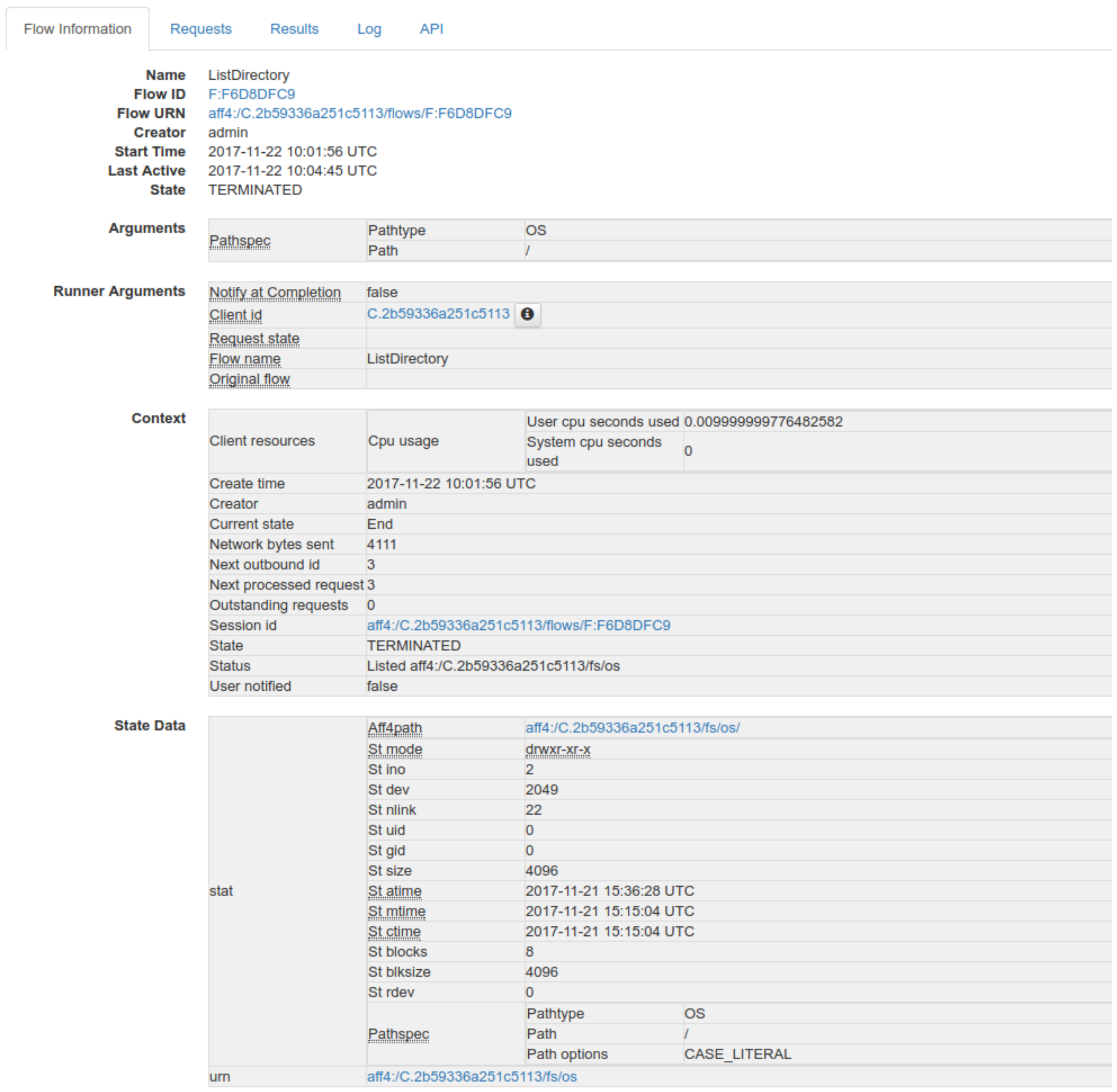
The table view shows the current state of the flow, what's the flow identifier (*Path*), the name of the Flow launched, the date when it was launched, when it was last active and who created it.

As you can see, a couple of Flows have been launched in the shown example:

1. *CAEnroler*. This is the first flow ever to launch for any client. It is the enroling Flow which gets the client set up server side.
2. *Interrogate*. After enroling, a client sends some information about the machine it's running in such as the hostname, MAC address or users available on the system. This is the flow that fetches this information and if you remember the *Host Information* option, most information is contained there.

   This flow has also a few subflows shown in the table - flows can call other flows to collect more information from the client.
3. *ListDirectory*. A Flow that lists the contents of a directory. This is what happened when the refresh button was pressed on the GUI.
4. *FileFinder*. This flow is used to download and list files on the client machine.

Let's see the *ListDirectory* flow in detail. You can click on any flow to get detailed information.



Name	ListDirectory
Flow ID	F:F6D8DFC9
Flow URN	aff4:/C.2b59336a251c5113/flows/F:F6D8DFC9
Creator	admin
Start Time	2017-11-22 10:01:56 UTC
Last Active	2017-11-22 10:04:45 UTC
State	TERMINATED

Arguments

Pathspec	Pathtype	OS
	Path	/

Runner Arguments

Notify at Completion	false
Client id	C.2b59336a251c5113
Request state	
Flow name	ListDirectory
Original flow	

Context

Client resources	Cpu usage	User cpu seconds used	0.009999999776482582
		System cpu seconds used	0
Create time	2017-11-22 10:01:56 UTC		
Creator	admin		
Current state	End		
Network bytes sent	4111		
Next outbound id	3		
Next processed request	3		
Outstanding requests	0		
Session id	aff4:/C.2b59336a251c5113/flows/F:F6D8DFC9		
State	TERMINATED		
Status	Listed aff4:/C.2b59336a251c5113/fs/os		
User notified	false		

State Data

stat	Aff4path	aff4:/C.2b59336a251c5113/fs/os/	
	St mode	drwxr-xr-x	
	St ino	2	
	St dev	2049	
	St nlink	22	
	St uid	0	
	St gid	0	
	St size	4096	
	St atime	2017-11-21 15:36:28 UTC	
	St mtime	2017-11-21 15:15:04 UTC	
	St ctime	2017-11-21 15:15:04 UTC	
	St blocks	8	
	St blksize	4096	
	St rdev	0	
	Pathspec	Pathtype	OS
		Path	/
		Path options	CASE_LITERAL
urn	aff4:/C.2b59336a251c5113/fs/os		

There's a lot of information here. Some bits worth mentioning are:

- Flow *state*, which tells us whether it finished correctly (oddly named **TERMINATED**) or not (**ERROR**), or if it's still running (**RUNNING**).
- *Arguments* show the arguments this flow was started with.
- In *Context* there is information about resource usage of the flow and metadata about how many messages were sent back and forth to the client.
- *State Data* shows internal server side state of this flow. This information is mostly useful for debugging.

The other tabs shown on the flow details view are

- *Requests*. Show what pending requests this flow is waiting for while it's running. Mostly for debugging.

- *Results*. The results returned by this flow. More information in Working with Results.
- *Log*. The log for this flow.
- *Api*. This tab shows how the flow can be started automatically using the GRR API. See the Chapter on Automation.

Starting Flows

To start a new Flow simply click on the *Start new flows* option on the left panel while having a client selected. The main panel will populate with the holy trinity of panels. The tree view shows all the Flows organized by category.

For example, in order to start a *FileFinder* flow, expand the *FileSystem* category and select the corresponding item. The flow view will populate with a form with all the user-configurable parameters for this flow. What's more, because each parameter has a well-defined type, GRR shows you widgets to select a value for each of them.

The FileFinder flow accepts a range parameters:

1. *Paths*. This is a list of textual paths that you want to look at.
2. *Pathtype*. Which VFS handler you want to use for the path. Available options are:
   - **OS**. Uses the OS "open" facility. These are the most straightforward for a first user. Examples of *os* paths are `C:/Windows` on Windows or `/etc/init.d/` on Linux/OSX.
   - **TSK**. Use Sleuthkit. Because Sleuthkit is invoked a path to the device is needed along the actual directory path. Examples of *tsk* paths are `\\?\Volume{19b4a721-6e90-12d3-fa01-806e6f6e6963}\Windows` for Windows or `/dev/sda1/init.d/` on Linux (But GRR is smart enough to figure out what you want if you use `C:\Windows` or `/init.d/` instead even though there is some guessing involved).
   - **REGISTRY**. Windows-related. You can open the live Windows registry as if it was a virtual filesystem. So you can specify a path such as `HKEY_LOCAL_MACHINE/Select/Current`.
   - **MEMORY** and **TMPFILE** are internal and should not be used in most cases.
3. *Condition*. The *FileFinder* can filter files based on condition like file size or file contents. The different conditions should be self explanatory. Multiple conditions can be stacked, the file will only be processed if it fulfills them all.
4. *Action*. Once a file passes all the conditions, the action decides what should be done with it. Options are **STAT**, **HASH** and **DOWNLOAD**. Stat basically just indicates if a file exists, this is mostly used to list directories (path `C:\Windows\*` and action STAT). Hash returns a list of hashes of the file and Download collects the file from the client and stores it on the server.

For this example, a good set of arguments would be a directory listing, something like path `C:\Windows\*` or `/tmp/*` and action **STAT**. Once you've filled in each required field, click on *Launch* and if all parameters validated, the Flow will run. Now you can go to the *Manage launched flows* view to find it running or track it.

> **Important** Not all flows might be available on every platform. When trying to run a flow that's not available in the given platform an error will show up.

Available flows

The easiest ways to see the current flows is to check in the AdminUI under StartFlow. These have useful documentation.

Note that by default only BASIC flows are shown in the Admin UI. By clicking the settings (gear icon) in the top right, you can enable ADVANCED flows. With this set you will see many of the underlying flows which are sometimes useful, but require a deeper understanding of GRR.

Specifying File Paths

Providing file names to flows is a core part of GRR, and many flows have been consolidated into the File Finder flow, which uses a glob+interpolation syntax.

File Path Examples

All executables or dlls in each user's download directory:

```
%%users.homedir%%\Downloads\*.{exe,dll}
```

All .evtx files found up to three directories under "C:\Windows\System32\winevt":

```
%%environ_systemroot%%\System32\winevt\**.evtx
```

"findme.txt" files in user homedirs, up to 10 directories deep:

```
%%users.homedir%%/**10/findme.txt
```

File Paths: backslash or forward slash?

Either forward "/home/me" or backslash "C:\Users\me" path specifications are allowed for any target OS. They will be converted to a common format internally. We recommend using whatever is normal for the target OS: (backslash for Windows, fwdslash for OS X and Linux).

File Path Interpolation

GRR supports path interpolation from values in the artifact Knowledge Base. Interpolated values are enclosed with %%, and may expand to multiple elements. e.g.

```
%%users.homedir%%
```

Might expand to the following paths on Windows:

```
C:\Users\alice
C:\Users\bob
C:\Users\eve
```

and on OS X:

```
/Users/alice
/Users/bob
/Users/eve
```

and on Linux:

```
/home/alice
/usr/local/home/bob
/home/local/eve
```

A full list of possible interpolation values can be found by typing %% in the gui. The canonical reference is the proto/knowledge_base.proto file, which also contains docstrings for each type.

Path Globbing

Curly braces work similarly to bash, e.g:

```
{one,two}.{txt,doc}
```

Will match:

```
one.txt
two.txt
one.doc
two.doc
```

Recursive searching of a directory is performed with **. The default search depth is 3 directories. So:

```
/root/**.doc
```

Will match:

```
/root/blah.doc
/root/1/something.doc
/root/1/2/other.doc
/root/1/2/3/another.doc
```

More depth can be specified by adding a number to the **, e.g. this performs the same search 10 levels deep:

```
/root/**10.doc
```

> **IMPORTANT** Note that the FileFinder transfers all data to the server and does the matching server side. This might lead to terrible performance when used with deep recursive directory searches. For a faster alternative that has the drawback of leaking the path you are searching for to the potentially compromised client, use the *ClientFileFinder* flow which does the matching right on the client.

Grep Syntax

A number of GRR flows (such as the File Finder) accept Grep specifications, which are a powerful way to search file and memory contents. There are two types of grep syntax: literal and regex.

Literal Matches

Use this when you have a simple string to match, or want to match a byte string. Here's a simple string example (note no quotes required):

```
allyourbase
```

And a byte string example:

```
MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00
```

To minimise the potential for errors we recommend using python to create byte strings for you where possible, e.g. the above byte string was created in ipython like this:

```
In [1]: content = open("test.exe","rb").read(12)
In [2]: content
Out[2]: 'MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00'
```

Regex Matches

Use this when you need more complex matching. The format is a regular python regex (see http://docs.python.org/2/library/re.html) with the following switches applied automatically:

```
re.IGNORECASE | re.DOTALL | re.MULTILINE
```

An example regex is below. The entire match is reported, () groups are not broken out separately. Also note that 10 bytes before and after will be added to any matches by default - use the Advanced menu to change this behavior:

```
Accepted [^ ]+ for [^ ]+ from [0-9.]+ port [0-9]+ ssh
```

Specifying Windows Registry Paths

When specifying registry paths, GRR uses the following hive names (these can also be found by looking at the registry folder under "Browse Virtual Filesystem"):

```
HKEY_CLASSES_ROOT
HKEY_CURRENT_CONFIG
HKEY_CURRENT_USER
HKEY_DYN_DATA
HKEY_LOCAL_MACHINE
HKEY_PERFORMANCE_DATA
HKEY_USERS
```

The Registry Finder flow uses the same path globbing and interpolation system as described in Specifying File Paths. Examples:

```
HKEY_USERS\%%users.sid%%\Software\Microsoft\Windows\CurrentVersion\Run\*
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce\*
```

RegistryFinder will retrieve the values of any keys specified and report them in the registry data field. Default values will be retrieved and reported in registry data of the parent key. E.g. for this registry structure:

```
HKEY_LOCAL_MACHINE\Software\test:
  (Default) = "defaultdata"
  subkey = "subkeydata"
```

Collecting this:

```
HKEY_LOCAL_MACHINE\Software\test\*
```

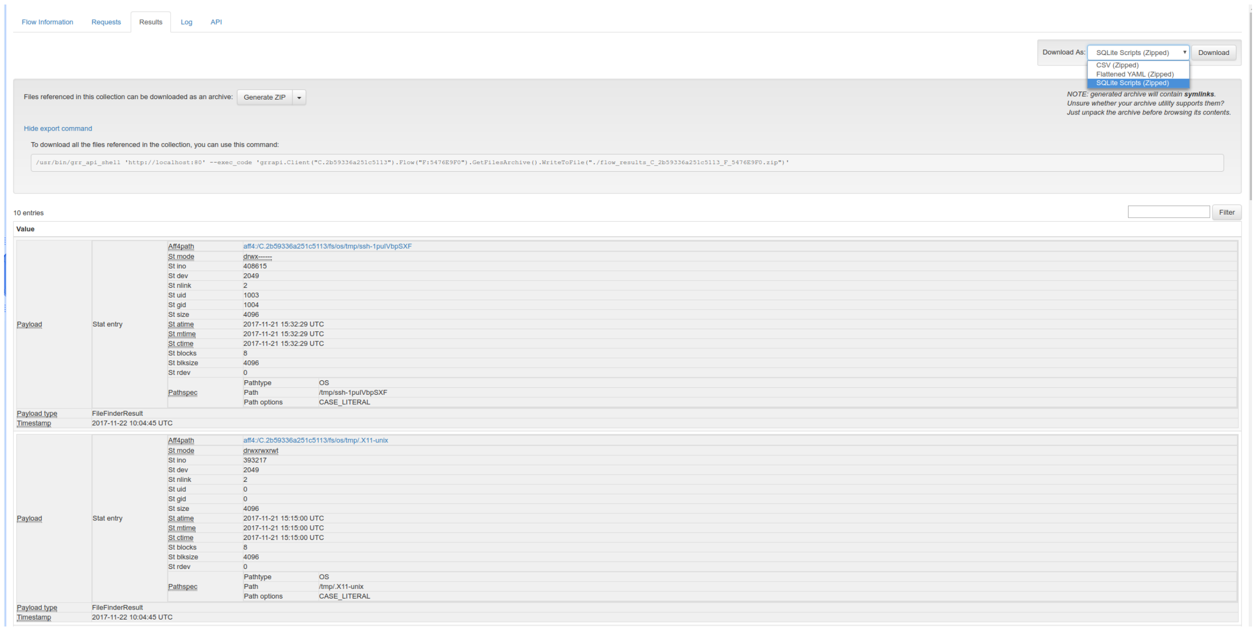
Will give results like:

```
Path:            /HKEY_LOCAL_MACHINE/SOFTWARE/test
Registry data:   defaultdata

Path:            /HKEY_LOCAL_MACHINE/SOFTWARE/test/subkey
Registry data:   subkeydata
```

Working with Flow results

All flows in GRR produce a list of (0 or more) results. For example in the image below, the results for a FileFinder flow are shown. The type of the items in the list is *FileFinderResult* which are essentially Stat entries that also reference collected files in case the file finder was also downloading the files from the client to the GRR server.



Files referenced in this collection can be downloaded as an archive:

NOTE: generated archive will contain ***symlinks****.*
Unsure whether your archive utility supports them?
Just unpack the archive before browsing its contents.

To download all the files referenced in the collection, you can use this command:

```
/usr/bin/grr_api_shell 'http://localhost:80' --exec_code 'grrapi.Client("C.2b59336a251c5113").Flow("F:5476E9F0").GetFilesArchive().WriteToFile("./flow_results_C_2b59336a251c5113_F_5476E9F0.zip")'
```

10 entries

Value				
Payload	Stat entry	Aff4path	aff4:/C.2b59336a251c5113/fs/os/tmp/ssh-1pulVbpSXF	
		St mode	drwx------	
		St ino	408615	
		St dev	2049	
		St nlink	2	
		St uid	1003	
		St gid	1004	
		St size	4096	
		St atime	2017-11-21 15:32:29 UTC	
		St mtime	2017-11-21 15:32:29 UTC	
		St ctime	2017-11-21 15:32:29 UTC	
		St blocks	8	
		St blksize	4096	
		St rdev	0	
		Pathspec	Pathtype	OS
			Path	/tmp/ssh-1pulVbpSXF
			Path options	CASE_LITERAL
Payload type	FileFinderResult			
Timestamp	2017-11-22 10:04:45 UTC			
Payload	Stat entry	Aff4path	aff4:/C.2b59336a251c5113/fs/os/tmp/.X11-unix	
		St mode	drwxrwxrwt	
		St ino	393217	
		St dev	2049	
		St nlink	2	
		St uid	0	
		St gid	0	
		St size	4096	
		St atime	2017-11-21 15:15:00 UTC	
		St mtime	2017-11-21 15:15:00 UTC	
		St ctime	2017-11-21 15:15:00 UTC	
		St blocks	8	
		St blksize	4096	
		St rdev	0	
		Pathspec	Pathtype	OS
			Path	/tmp/.X11-unix
			Path options	CASE_LITERAL
Payload type	FileFinderResult			
Timestamp	2017-11-22 10:04:45 UTC			

GRR only provides rudimentary analysis functionality for results. Results can be filtered using the filter input on the right which does a full text match for results (including all metadata fields) but in many cases it's better to export the results and use external analysis tools instead.

GRR offers to download the results in three different formats (the `Download As:` dropdown on the right:

- CSV
- YAML
- SQlite

NOTE All those formats will flatten the data, i.e., you might lose some information. In most cases this does not make a difference but it's good to be aware that this happens.

Files

In the case where the results contain items that reference files (StatEntry for example), another option will be shown that allows the download of the referenced files as a ZIP or TAR archive. Additionally, GRR shows an export command that can be pasted into a GRR API shell and will download the same ZIP file programmatically using the GRR API.

2.5.6 Hunts

What are "Hunts" and how to use them

Hunting is one of the key features of GRR. Anything you can do on a single client, should be able to be done on thousands of clients just as easily.

A hunt specifies a Flow, the Flow parameters, and a set of rules for which machines to run the Flow on.

Creating a Hunt

Manual Hunt Creation

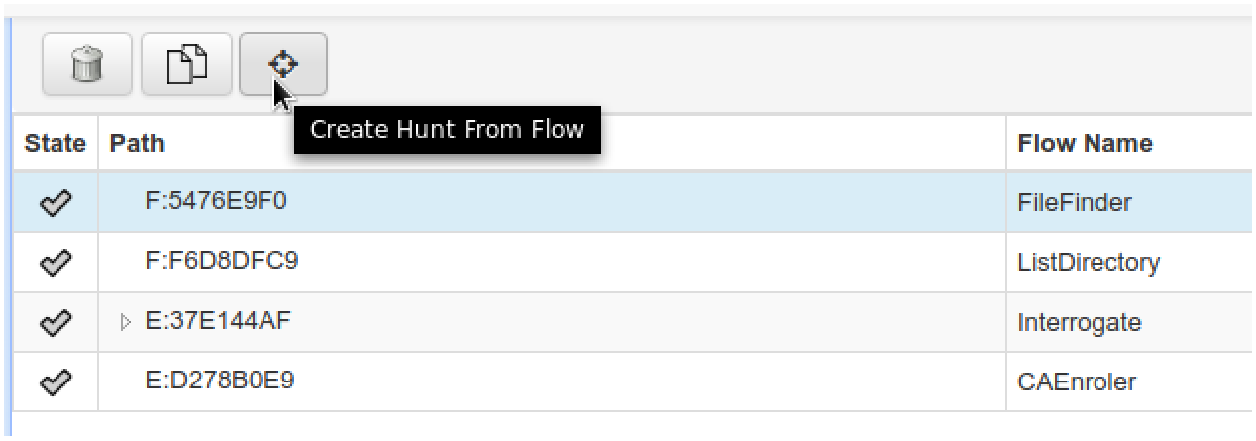
You can create a new Hunt in the Hunt Manager section of the UI. To create a Hunt:

1. Click the + button
2. Select a flow you want to run and fill out all parameters as you would do for a flow running on a single client.
3. The second page of the Hunt Wizard lets you set hunt parameters. They are:
   - *Hunt Description* A friendly name for this hunt to show in the UI.
   - *Client Limit* A limit on the number of clients this hunt should run on. This can be changed later so it makes sense to start the hunt on a few clients first (we like 100) to see how it goes and only later remove the limit.
   - *Crash Limit* If more clients than indicated in this parameter return an error, the hunt is automatically paused.
   - *Expiry Time* Hunts are never really done since new clients might appear at any time and they should run the hunt too. This time indicates the lifetime of a hunt after which it is considered done.
   - *Client Rate* Number of clients to schedule the hunt on per minute. The default of 20 we have found to be safe, avoiding overloading the server for intensive hunts with lots of message passing (e.g. multiple flows, lots of results). A value of 0 disables rate limiting and clients will be scheduled as fast as possible. Use this with care: light hunts you need to run quickly.
   - Some more advanced options are available behind the advanced link but they should not need adjustment in most cases.
4. Set any output plugins, such as receiving an email for each result.
5. Set Hunt Rules. Most commonly this is used to select a specific platform target, e.g. Windows. See the Rules section.
6. Click Run

Unless approvals are required, you can start the hunt immediately.

Creating a Hunt from a Flow

Manual hunt creation is a bit dangerous because mistakes happen and they can propagate to many machines in short time. A better way of creating a hunt is to copy an existing and tested flow. In order to do so, create a flow on a client with the parameters you want to hunt for and make sure that it works. Once you are convinced that this flow is what you want to run on the fleet, use the “Create Hunt from Flow” button in the Manage launched flows view:



State	Path	Flow Name
✓	F:5476E9F0	FileFinder
✓	F:F6D8DFC9	ListDirectory
✓	E:37E144AF	Interrogate
✓	E:D278B0E9	CAEnroler

This will copy all flow parameters to a new hunt and you can continue to set all Hunt parameters starting from Step 3 above.

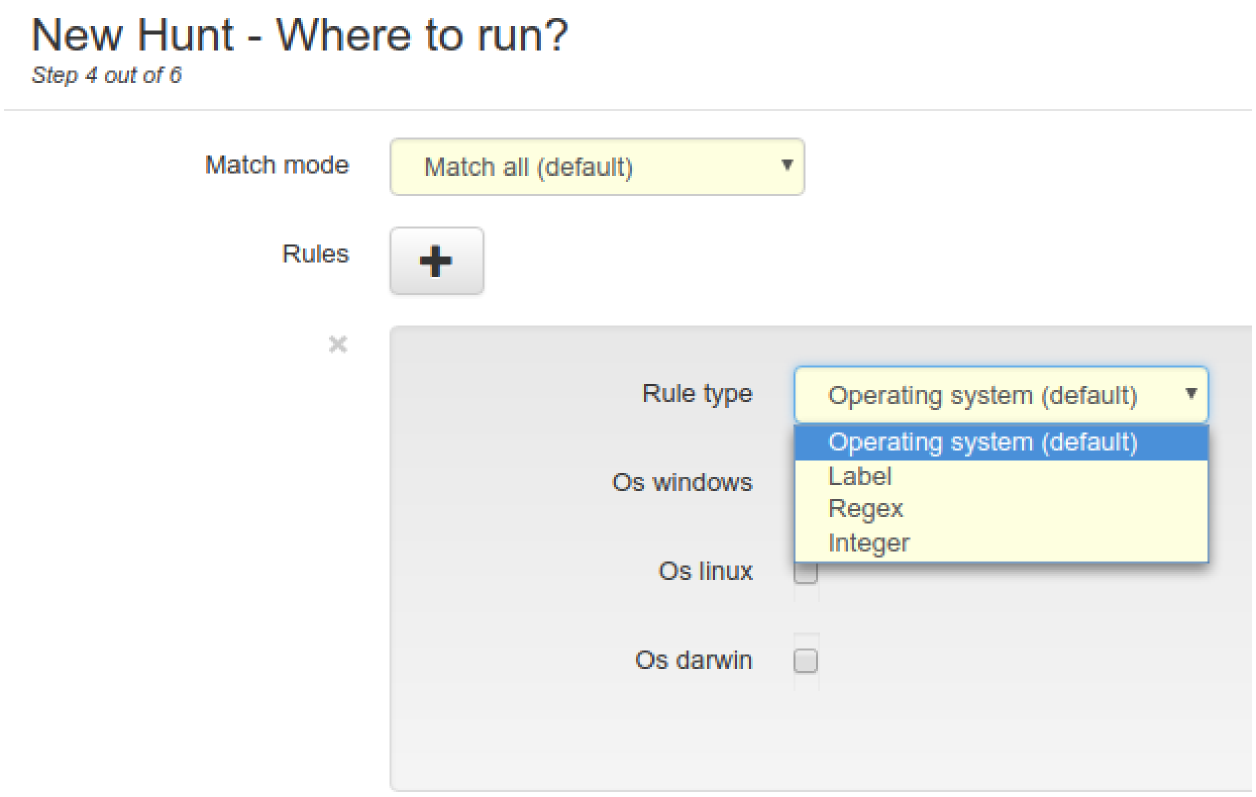
Hunt Rules

Hunt rules are used to define a subset of the clients to run a hunt on. The most common rules are

- Limiting the hunt to a single operating system and
- Limiting the hunt to clients that have a certain label attached

but GRR offers also regex and integer matching on all client attributes so more sophisiticated hunt rules can be created.

The rule section of the New Hunt Wizard can be seen in the following image:


New Hunt - Where to run?
Step 4 out of 6

Match mode: Match all (default)

Rules

Rule type: Operating system (default)

GRR allows to specify more than one rule for hunts. By default, those rules are evaluated using a logical AND, meaning a client will only run the hunt if it satisfies all the conditions specified by every rule. This behavior can be changed with the dropdown on top, changing

- Match all (logical AND)

to

- Match any (logical OR)

Hunt Limits, DOs and DONTs

Creating Hunts

As already outlined in the section about creating hunts, hunts might negatively impact both the client machines they run on and also the GRR system by causing extensive load. In order to prevent this, we recommend to always start hunts either by

- copying an existing and tested flow or
- copying an existing hunt, applying only minor modifications.

We realize that sometimes unexpected problems can still arise and therefore GRR applies some limits to hunts.

Default Hunt Limits

There are two sets of limits in place for hunts. GRR enforces both, limits on individual clients and limits on the average resource usage for the hunt.

Individual Client Limits

Analoguous to what happens with flows, GRR enforces limits on the resource usage for each client participating in a hunt. Since the impact of a hunt is potentially much larger than for a single flow, the limits for hunts are lower than the flow limits. The current defaults are:

- 600 cpu seconds per client
- 100 MB of network traffic per client

Limits on Average Resource Usage

Once a hunt has processed 1000 clients, the average resource usage is also checked and enforced by GRR. The defaults for those limits are:

- 1000 results on average per client
- 60 cpu seconds on average per client
- 10 MB of network traffic on average per client

All the limits in this section can be overridden in the advanced settings when scheduling hunts. Use at your own risk.

Hunt Controls

Create a new hunt

Use this button to create a new hunt.

Start a Hunt

Use this button to start a newly created hunt. New hunts are created in the PAUSED state, so you'll need to do this to run them. Hunts that reach their client limit will also be set to PAUSED, use this button to restart them after you have removed the client limit (see modify below).

Stop a Hunt

Stopping a hunt will prevent new clients from being scheduled and interrupt in-progress flows the next time they change state. This is a hard stop, so in-progress results will be lost, but results already reported are unaffected. Once a hunt is stopped, there is no way to start it again.

Modify a Hunt

The modify button allows you to change the hunt client limit and the hunt expiry time. Typically you use this to remove (set to 0) or increase a client limit to let the hunt run on more machines. Modifying an existing hunt doesn't require re-approval. Hunts can only be modified in the PAUSED state.

Copy a Hunt

The copy button creates a new hunt with the same set of parameters as the currently selected one. If you find a mistake in a scheduled hunt and have to stop it, this button can be used to quickly reschedule a fixed version of the hunt.

Delete a Hunt

Use this to remove an unwanted hunt. For accountability reasons hunts can only be deleted if they haven't run on any clients.

Show automated hunts

Use this button to display all hunts, including those created by cronjobs. These are hidden by default to reduce UI clutter.

Working with Hunt results

Hunt results in GRR work almost the same as Flow results. The only difference is that each item is annotated with the client id of the client that produced it. Everything else (filtering, exporting, generating archives with collected files) works exactly the same as with flow results as described in the flow results section.

Troubleshooting ("Why is my hunt doing nothing?")

- There are caches involved in the frontend server, you may need to wait a couple of minutes before the first client picks up the flow.

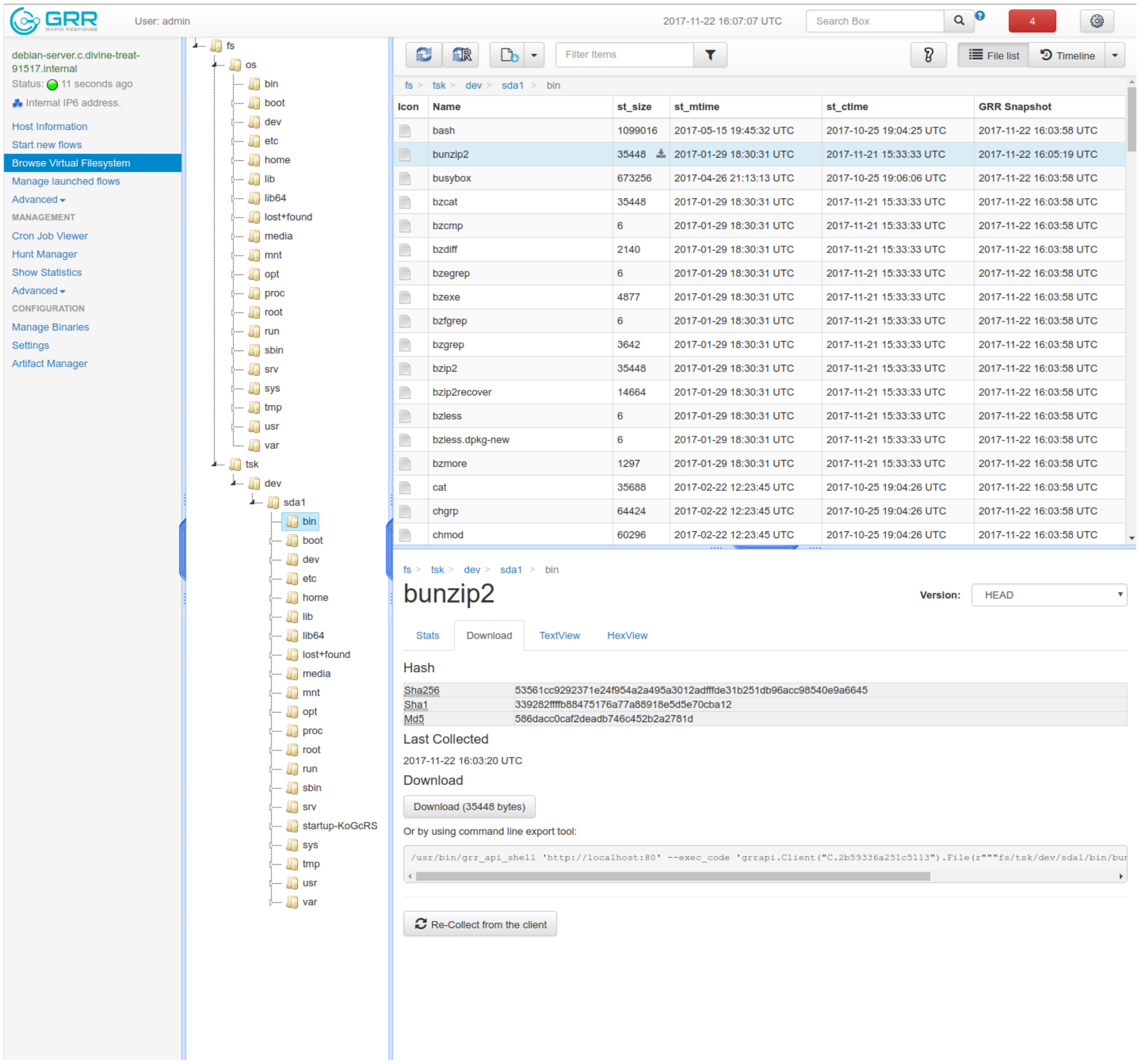
- Clients only check if there is hunt work to do when doing a foreman check. The frequency of these checks are specified in the `Client.foreman_check_frequency` parameter. This defaults to every 30 minutes.
- Even when a client issues a foreman check, the flows may not immediately start. Instead, the process is asynchronous, so the check tells the server to check its hunt rules to see if there are things for the client to do. If there are, it schedules them, but the client may not do its regular poll and pick up that flow until `Client.poll_max period` (10 minutes by default).
- When you run a hunt you can specify a "Client Rate" as specified in creating hunts. If this is set low (but not 0), you can expect a slow hunt.
- When running a hunt under high server load, clients seem appear complete in batches. This results in the completion graph appearing "stepped". The clients are finishing normally, but their results are being processed and logged in batches by the Hunt. When the system is under load, this hunt processing takes some time to complete resulting in the *steps*.

2.5.7 Virtual File System

Virtual File System

Every time GRR collects forensics information (files, Registry data, ...) from a client, it stores it server side in the data store. This information is presented to the analyst in a unified view we call the Virtual Filesystem view (menu item *Browse Virtual Filesystem*).

A screenshot is shown below (larger Image).



Icon	Name	st_size	st_mtime	st_ctime	GRR Snapshot
	bash	1099016	2017-05-15 19:45:32 UTC	2017-10-25 19:04:25 UTC	2017-11-22 16:03:58 UTC
	bunzip2	35448	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:05:19 UTC
	busybox	673256	2017-04-26 21:13:13 UTC	2017-10-25 19:06:06 UTC	2017-11-22 16:03:58 UTC
	bzcat	35448	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzcmp	6	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzdiff	2140	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzegrep	6	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzexe	4877	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzfgrep	6	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzgrep	3642	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzip2	35448	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzip2recover	14664	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzless	6	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzless.dpkg-new	6	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	bzmore	1297	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:03:58 UTC
	cat	35688	2017-02-22 12:23:45 UTC	2017-10-25 19:04:26 UTC	2017-11-22 16:03:58 UTC
	chgrp	64424	2017-02-22 12:23:45 UTC	2017-10-25 19:04:26 UTC	2017-11-22 16:03:58 UTC
	chmod	60296	2017-02-22 12:23:45 UTC	2017-10-25 19:04:26 UTC	2017-11-22 16:03:58 UTC

bunzip2

Hash

Sha256	53561cc9292371e24f954a2a495a3012adfffde31b251db96acc98540e9a6645
Sha1	339282ffffb88475176a77a88918e5d5e70cba12
Md5	586dacc0caf2deadb746c452b2a2781d

Last Collected

2017-11-22 16:03:20 UTC

Download

Or by using command line export tool:

```
/usr/bin/grr_api_shell 'http://localhost:80' --exec_code 'grrapi.Client("C.2b59336a251c5113").File(r"""fs/tsk/dev/sda1/bin/bun
```

VFS Tree

On the left there is a tree navigator. The most important top level items are:

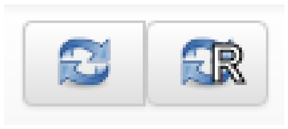
- *fs* This gives a view of the client's filesystem.
- *fs/os* Under fs/os, GRR maps all the files it collected on a client *using the OS apis*, i.e., standard file manipulation system calls. Under fs/os, GRR shows everything that is mounted under / for Linux and Mac clients, for Windows clients, under fs/os there are entries like C: for each disk present on the client system.
- *fs/tsk* GRR also uses The Sleuth Kit (TSK) to open disks and images in raw mode and parsing the file systems on them. Every file collected in that way will be presented under fs/tsk. The first level under fs/tsk will be a raw disk, for example sda1 for Linux or \\?\Volume{5f62dc05-4cfc-58fd-b82c-579426ac01b8} for Windows.
- *registry* Only present for Windows machines, this branch gives a view into the live registry on the client machine. The first level after /registry are hives, HKEY_LOCAL_MACHINE or HKEY_USERS for example.

Note how in the screenshot, the folders under /fs/tsk and /fs/os actually reference the same folder on disk ("/"). However, there is an additional folder startup-KoGcRS that is only visible when using TSK, it gets filtered by the operating system API.

Navigating the VFS

Using the tree navigator, an analyst can look at all the files / registry values *that GRR has already downloaded from the client*. This is important to note, the view only shows what the server has stored for the client machine, not the current state. Sometimes this means that directories or Registry keys show up empty but not because there are no keys / values but just because the corresponding information has never been collected.

In order to populate the VFS, there are two buttons that trigger collection of more information:



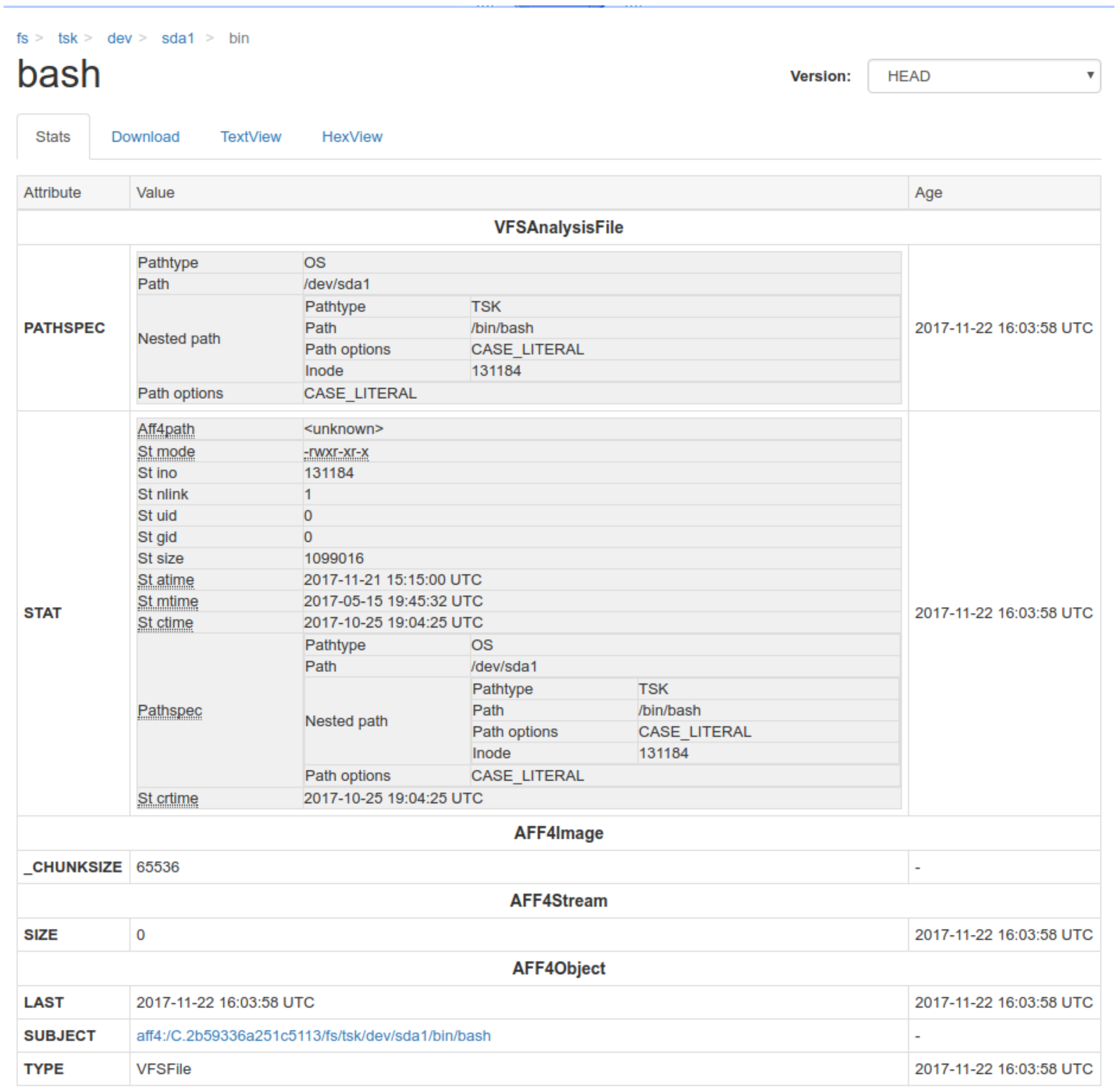
The left button triggers a refresh of the current branch of the VFS with data collected from the client - similar to listing a directory. GRR automatically figures out which kind of data is currently displayed (OS based files, TSK based files, Registry information) and schedules the corresponding collection action.

The button marked with an R schedules a recursive refresh.

> **Note** that recursive refrehses with a large depth might impact the client and GRR system.

Looking at Files

In the file detail view, GRR always shows Stat information about the file:



fs > tsk > dev > sda1 > bin

bash

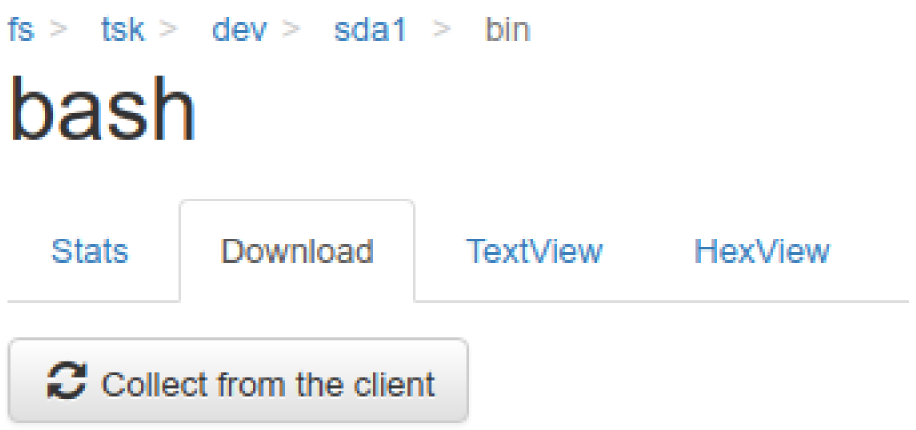
Version: HEAD

Stats | Download | TextView | HexView

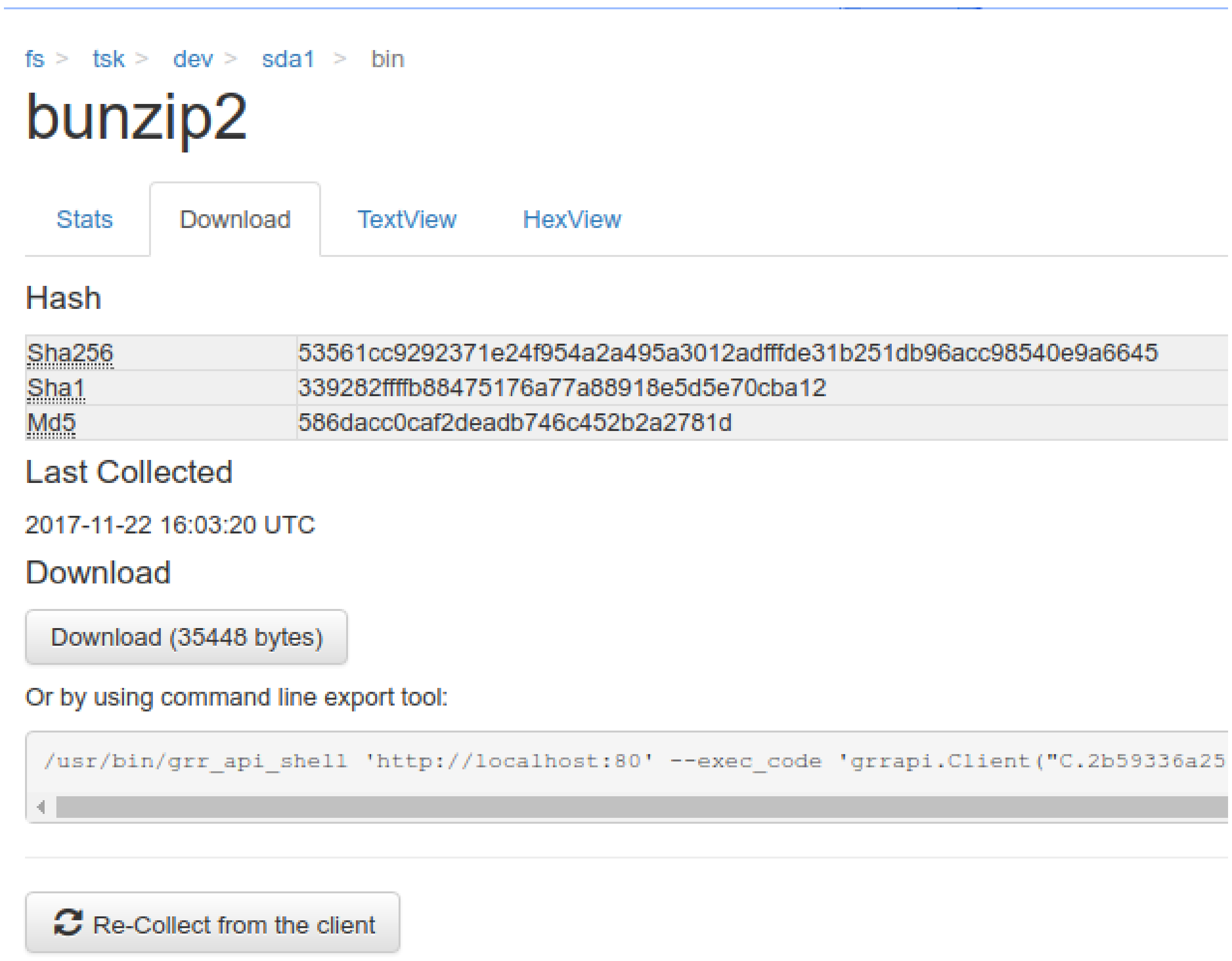
Attribute	Value	Age
VFSAnalysisFile		
PATHSPEC	Pathtype: OS; Path: /dev/sda1; Nested path: (Pathtype: TSK; Path: /bin/bash; Path options: CASE_LITERAL; Inode: 131184); Path options: CASE_LITERAL	2017-11-22 16:03:58 UTC
STAT	Aff4path: <unknown>; St mode: -rwxr-xr-x; St ino: 131184; St nlink: 1; St uid: 0; St gid: 0; St size: 1099016; St atime: 2017-11-21 15:15:00 UTC; St mtime: 2017-05-15 19:45:32 UTC; St ctime: 2017-10-25 19:04:25 UTC; Pathspec: (Pathtype: OS; Path: /dev/sda1; Nested path: (Pathtype: TSK; Path: /bin/bash; Path options: CASE_LITERAL; Inode: 131184); Path options: CASE_LITERAL); St crtime: 2017-10-25 19:04:25 UTC	2017-11-22 16:03:58 UTC
AFF4Image		
_CHUNKSIZE	65536	-
AFF4Stream		
SIZE	0	2017-11-22 16:03:58 UTC
AFF4Object		
LAST	2017-11-22 16:03:58 UTC	2017-11-22 16:03:58 UTC
SUBJECT	aff4:/C.2b59336a251c5113/fs/tsk/dev/sda1/bin/bash	-
TYPE	VFSFile	2017-11-22 16:03:58 UTC

Note that there is a version dropdown on top of this page and all data is timestamped. Whenever GRR collects information from a client machine, it just creates a new version of the stored object in the database, old information is never overwritten. This is why it's always possible to go back in the GRR DB and look at previously collected data - both metadata and contents - by simply selecting an earlier version.

Often, GRR knows about the file but has not collected any contents yet. In the download tab there is an option to collect the file from the client:



Again, GRR will collect the file the correct way (OS or TSK) automatically. Once the file is available, the same download view will have options to download the file to the analysts machine. Basic text and hex views are also provided.



bunzip2

Hash

Sha256	53561cc9292371e24f954a2a495a3012adfffde31b251db96acc98540e9a6645
Sha1	339282ffffb88475176a77a88918e5d5e70cba12
Md5	586dacc0caf2deadb746c452b2a2781d

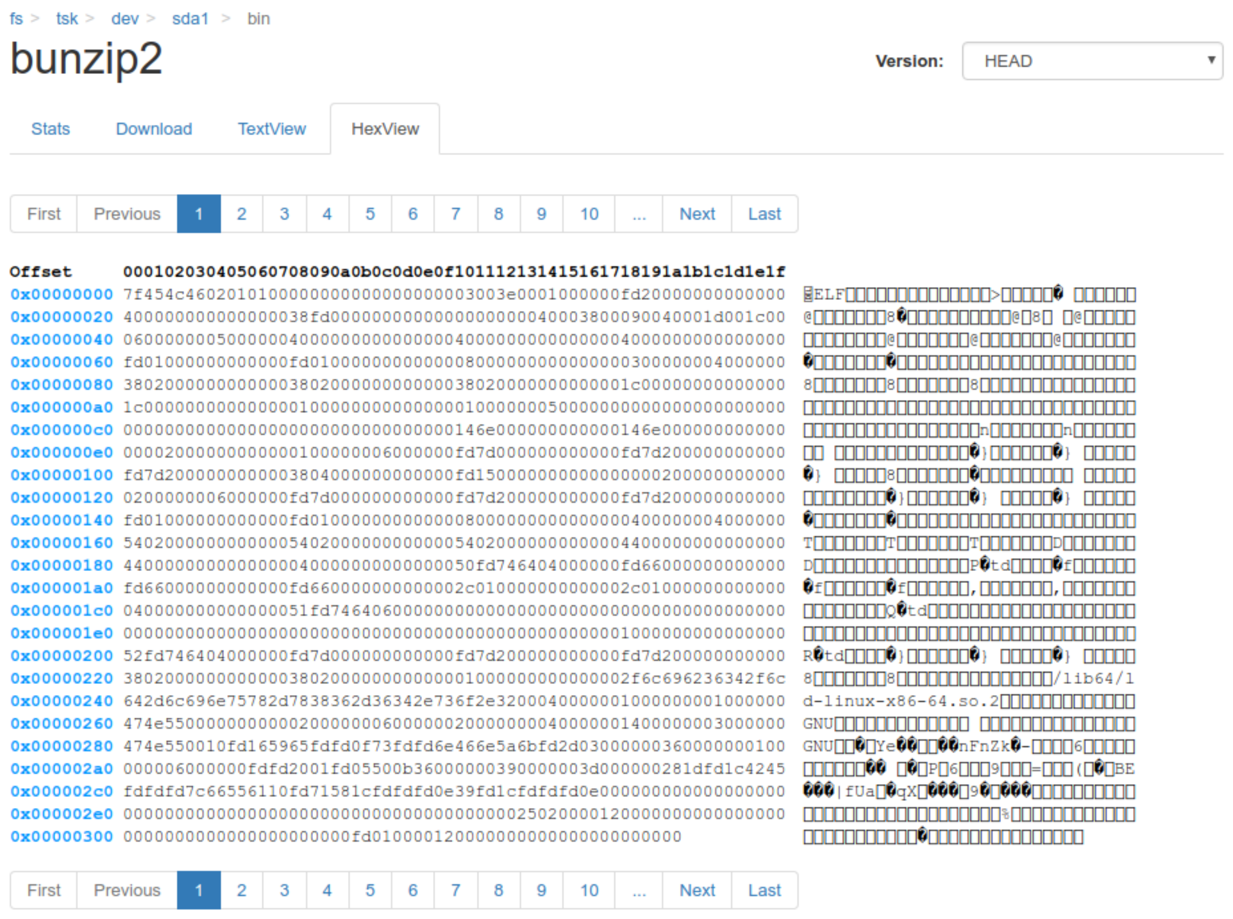
Last Collected

2017-11-22 16:03:20 UTC

Download

Or by using command line export tool:

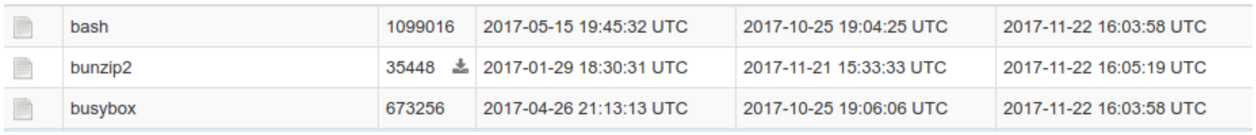
```
/usr/bin/grr_api_shell 'http://localhost:80' --exec_code 'grrapi.Client("C.2b59336a25
```



bunzip2

```
Offset     000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
0x00000000 7f454c46020101000000000000000000003003e0001000000fd20000000000000
0x00000020 4000000000000000038fd00000000000000000000040003800090040001d001c00
0x00000040 0600000005000000400000000000000040000000000000004000000000000000
0x00000060 fd01000000000000fd0100000000000008000000000000000300000004000000
0x00000080 38020000000000003802000000000000380200000000000001c00000000000000
0x000000a0 1c00000000000000010000000000000001000000050000000000000000000000
0x000000c0 000000000000000000000000000000000146e0000000000000146e00000000000
0x000000e0 00002000000000000001000000060000000fd7d000000000000fd7d2000000000000
0x00000100 fd7d2000000000000380400000000000000fd150000000000000000002000000000000
0x00000120 02000000060000000fd7d000000000000fd7d200000000000000fd7d2000000000000
0x00000140 fd01000000000000fd0100000000000008000000000000000400000004000000
0x00000160 540200000000000005402000000000000054020000000000000440000000000000
0x00000180 4400000000000000004000000000000000050fd7464040000000fd660000000000000
0x000001a0 fd6600000000000000fd660000000000000002c01000000000000002c01000000000000
0x000001c0 04000000000000000051fd7464060000000000000000000000000000000000000000
0x000001e0 0000000000000000000000000000000000000000000000001000000000000000
0x00000200 52fd746404000000fd7d000000000000fd7d200000000000000fd7d2000000000000
0x00000220 380200000000000003802000000000000001000000000000002f6c696236342f6c
0x00000240 642d6c696e75782d7838362d36342e736f2e32000400000010000000001000000
0x00000260 474e55000000000000200000006000000020000000040000001400000003000000
0x00000280 474e550010fd165965fdfd0f73fdfd6e466e5a6bfd2d030000000360000000100
0x000002a0 000006000000fdfd2001fd05500b36000000039000000003d000000281dfd1c4245
0x000002c0 fdfdfd7c66556110fd71581cfdfdfd0e39fd1cfdfdfd0e000000000000000000
0x000002e0 0000000000000000000000000000000000000025020000120000000000000000
0x00000300 0000000000000000000fd010000120000000000000000000000
```

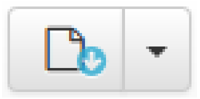
In the tree view, files with actual contents stored in GRR will have a small downloaded icon next to the size information. In this screenshot, bunzip2 is the only file with data collected:



	bash	1099016	2017-05-15 19:45:32 UTC	2017-10-25 19:04:25 UTC	2017-11-22 16:03:58 UTC
	bunzip2	35448	2017-01-29 18:30:31 UTC	2017-11-21 15:33:33 UTC	2017-11-22 16:05:19 UTC
	busybox	673256	2017-04-26 21:13:13 UTC	2017-10-25 19:06:06 UTC	2017-11-22 16:03:58 UTC

Bulk Downloading of Files

GRR offers the option to conviniently download all files collected from a client for offline analysis. The button



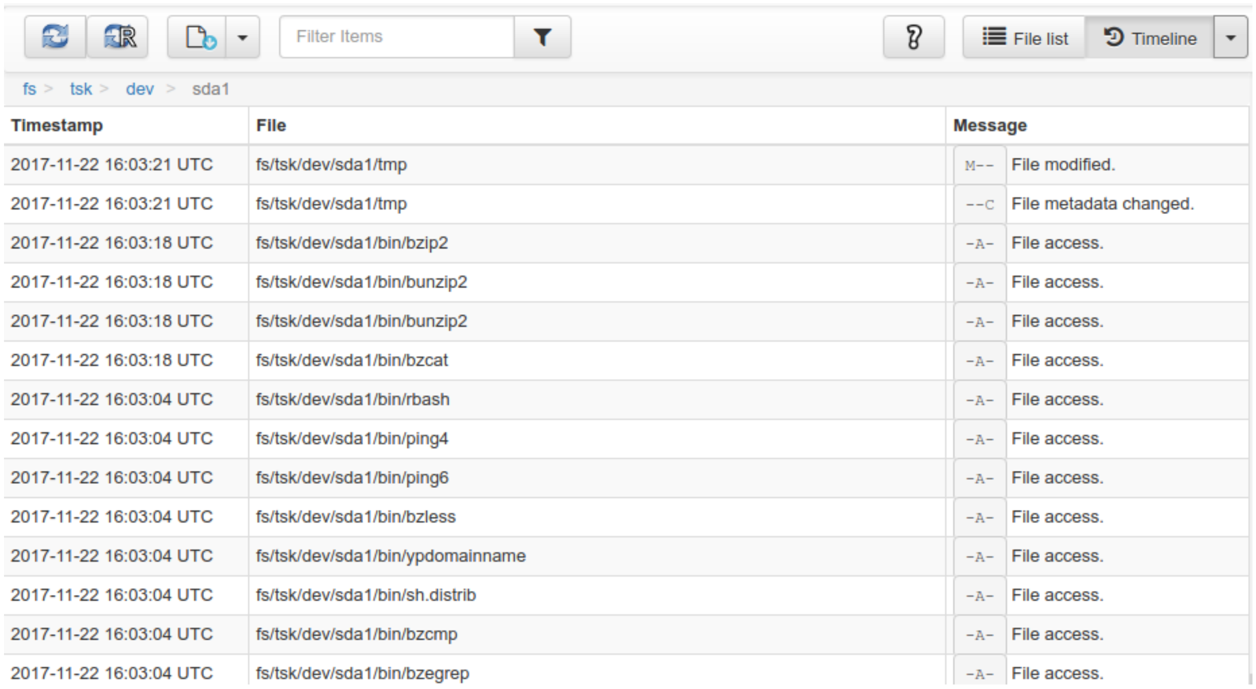
can be used to

- Download all files with data in the current directory and below or
- Download all files with data ever collected from that client

as a zip archive. The drop down choses between the two options.

Creating Timelines

The GRR VFS can be switched to a timeline view where all files *GRR knows about* are shown sorted by their various timestamps:



Timestamp	File	Message
2017-11-22 16:03:21 UTC	fs/tsk/dev/sda1/tmp	M-- File modified.
2017-11-22 16:03:21 UTC	fs/tsk/dev/sda1/tmp	--C File metadata changed.
2017-11-22 16:03:18 UTC	fs/tsk/dev/sda1/bin/bzip2	-A- File access.
2017-11-22 16:03:18 UTC	fs/tsk/dev/sda1/bin/bunzip2	-A- File access.
2017-11-22 16:03:18 UTC	fs/tsk/dev/sda1/bin/bunzip2	-A- File access.
2017-11-22 16:03:18 UTC	fs/tsk/dev/sda1/bin/bzcat	-A- File access.
2017-11-22 16:03:04 UTC	fs/tsk/dev/sda1/bin/rbash	-A- File access.
2017-11-22 16:03:04 UTC	fs/tsk/dev/sda1/bin/ping4	-A- File access.
2017-11-22 16:03:04 UTC	fs/tsk/dev/sda1/bin/ping6	-A- File access.
2017-11-22 16:03:04 UTC	fs/tsk/dev/sda1/bin/bzless	-A- File access.
2017-11-22 16:03:04 UTC	fs/tsk/dev/sda1/bin/ypdomainname	-A- File access.
2017-11-22 16:03:04 UTC	fs/tsk/dev/sda1/bin/sh.distrib	-A- File access.
2017-11-22 16:03:04 UTC	fs/tsk/dev/sda1/bin/bzcmp	-A- File access.
2017-11-22 16:03:04 UTC	fs/tsk/dev/sda1/bin/bzegrep	-A- File access.

Use the button on the top right in the VFS view to switch between Timeline and File List View. The drop down menu can be used to download the generated timeline as CSV.

Note that

- the timeline is always generated by recursively descending into directories starting from the currently selected one.
- the timeline only contains files that GRR has downloaded metadata before. This timelining mechanism is fully server side, if you haven't collected the metadata yet, the recursive refresh button will do this for you.

2.5.8 Artifacts

Artifacts

Outline

During a security investigation responders need to quickly retrieve common pieces of information that include items such as logs, configured services, cron jobs, patch state, user accounts, and much more. These pieces of information are known as forensic artifacts, and their location and format vary drastically across systems.

We have built a framework to describe forensic artifacts that allows them to be collected and customised quickly using GRR. This collection was initially contained inside the GRR repository, but we have now moved it out to a separate repository to make access simple for other tools.

Goals

The goals of the GRR artifacts implementation are:

- Describe artifacts with enough precision that they can be collected automatically without user input.
- Cover modern versions of Mac, Windows, and Linux and common software products of interest for forensics.
- Provide a standard variable interpolation scheme that allows artifacts to simply specify concepts like "all user home directories", `%TEMP%`, `%SYSTEMROOT%` etc.
- Allow grouping across operating systems and products e.g. "Chrome Web History" artifact knows where the web history is for Chrome on Mac, Windows and Linux.
- Allow grouping of artifacts into high level concepts like *persistence mechanisms*, and investigation specific meta-artifacts.
- To create simple, shareable, non-GRR-specific human-readable definitions that allow people unfamiliar with the system to create new artifacts. i.e. not XML or a domain specific language.
- The ability to write new artifacts, upload them to GRR and be able to collect them immediately.

Collecting Artifacts

Artifacts can be collected using the artifact collector flow. Multiple artifacts can be collected at once.

Using artifacts in hunts is particularly powerful as an artifact like `JavaCacheFiles` can be scheduled in a single hunt across all three operating systems, and the artifact itself determines the correct paths to be downloaded for each operating system.

Defining Artifacts

Basics

GRR artifacts are defined in YAML. The main artifact repository is hosted on GitHub. You can browse existing artifact definitions or read the exhaustive syntax overview.

Knowledgebase

We use a standard set of machine information collected from the host for variable interpolation. This collection of data is called the *knowledgebase* and is referenced with a `%%variable%%` syntax.

The artifact defines where the data lives. Once it is retrieved by GRR a parser can optionally be applied to turn the collected information into a more useful format, such as parsing a browser history file to produce URLs.

Uploading new definitions

New artifacts should be added to the forensic artifacts repository.

The changes can be imported into GRR by running `make` in the `grr/artifacts` directory. This will delete the existing artifacts, checkout the latest version of the artifact repository and add all of the YAML definitions into GRR's directory. Running `python setup.py build` will have the same effect. The new artifacts will be available once the server is restarted.

Artifacts can also be uploaded via the Artifact Manager GUI and used immediately without the need for a restart.

Local definitions

Artifacts that are specific to your environment or need to remain private can be added to the `grr/artifacts/local` directory. This directory will remain untouched when you update the main artifacts repository. You can also use this directory to test new artifacts before they are added to the main public repository.

Flow templates

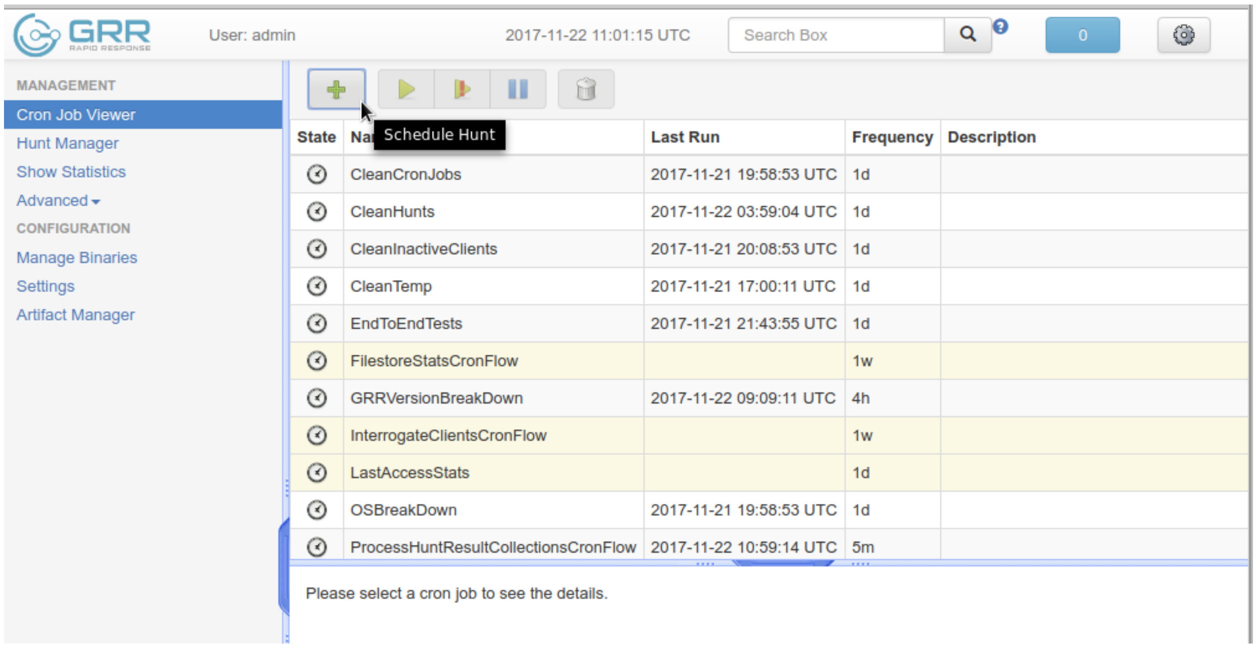
We currently support using the artifact format to call GRR-specific functionality, such as invoking a GRR client action, listing processes or running a rekall plugin. Such "artifacts" are called *flow templates* and since they are GRR-specific they remain in the GRR repository in the `grr/artifacts/flow_templates` directory.

This is a temporary working name. We intend to rework this functionality into a more general, powerful and configurable way to call GRR from YAML.

2.5.9 Cron Jobs in GRR

You can see a list of GRR cron jobs in "Cron Job Viewer" pane. GRR has a few system cron jobs that run periodically. These jobs perform periodical cleanup and maintenance tasks.

You can also use GRR's "Cron Job Viewer" to schedule periodical hunts.



State	Name	Last Run	Frequency	Description
	CleanCronJobs	2017-11-21 19:58:53 UTC	1d	
	CleanHunts	2017-11-22 03:59:04 UTC	1d	
	CleanInactiveClients	2017-11-21 20:08:53 UTC	1d	
	CleanTemp	2017-11-21 17:00:11 UTC	1d	
	EndToEndTests	2017-11-21 21:43:55 UTC	1d	
	FilestoreStatsCronFlow		1w	
	GRRVersionBreakDown	2017-11-22 09:09:11 UTC	4h	
	InterrogateClientsCronFlow		1w	
	LastAccessStats		1d	
	OSBreakDown	2017-11-21 19:58:53 UTC	1d	
	ProcessHuntResultCollectionsCronFlow	2017-11-22 10:59:14 UTC	5m	

Please select a cron job to see the details.

2.5.10 Automation using the GRR API

GRR AdminUI server also serves as an API endpoint. We provide the *grr_api_client* Python library which can be used for easy GRR automation. See the API client documentation for more details.

There's also an API client library in PowerShell working on Windows, Linux and macOS for GRR automation and scripting, developed by Swisscom.

2.5.11 Emergency Pushing of Code and Binaries

GRR offers functionality to deploy custom code or binaries to the whole fleet quickly. It goes without saying that this functinality can lead to problems and should therefore be used with care - we recommend using this as a last resort means in case things have gone terribly bad and not relying on this functionality for normal operations.

Binaries and python code can be pushed from the server to the clients to enable new functionality. The code that is pushed from the server must be signed by the corresponding private key for `Client.executable_signing_public_key` for python and binaries. These signatures will be checked by the client to ensure they match before the code is used.

What is actually sent to the client is the code or binary wrapped in a protobuf which will contain a hash, a signature and some other configuration data.

To sign code requires use of config_updater utility. In a secure environment the signing may occur on a different box from the server, but the examples below show the basic example.

Deploying Arbitrary Python Code.

To execute an arbitrary python blob, you need to create a file with python code that has the following attributes:

- Code in the file must work when executed by exec() in the context of a running GRR client.
- Any return data that you want sent back to the server can be stored encoded as a string in a variable called “magic_return_str”. Alternatively, GRR collects all output from print statements and puts them into the magic_return_string.

E.g. as a simple example. The following code modifies the clients poll_max setting and pings test.com.

```
import commands
status, output = commands.getstatusoutput("ping -c 3 test.com")
config_lib.CONFIG.Set("Client.poll_max", 100)
config_lib.CONFIG.Write()
magic_return_str = "poll_max successfully set. ping output %s" % output
```

This file then needs to be signed and converted into the protobuf format required, and then needs to be uploaded to the data store. You can do this using the following command line.

```
grr_config_updater upload_python --file=myfile.py --platform=windows
```

The uploaded files live by convention in aff4:/config/python_hacks and are viewable in the Manage Binaries section of the Admin UI.

The ExecutePythonHack Flow is provided for executing the file on a client. This is visible in the Admin UI under Administrative flows if Advanced Mode is enabled.

Note: Specifying arguments to a PythonHack is possible as well through the py_args argument, this can be useful for making the hack more generic.

Deploying Executables.

The GRR Agent provides an ExecuteBinaryCommand Client Action which allows us to send a binary and set of command line arguments to be executed. Again, the binary must be signed using the executable signing key (config option PrivateKeys.executable_signing_private_key).

To sign an exe for execution use the config updater script.

```
db@host:$ grr_config_updater upload_exe --file=/tmp/bazinga.exe --platform=windows
Using configuration <ConfigFileParser filename="/etc/grr/grr-server.conf">
Uploaded successfully to /config/executables/windows/installers/bazinga.exe
db@host:$
```

This file can then be executed with the LaunchBinary flow which is in the Administrative flows if Advanced Mode is enabled.

2.5.12 Importing the NSRL

The National Software Registry List (NSRL) is a collection of known software managed by NIST. It is commonly used in forensics to reduce the scope of analysis of already known software. This is typically done by whitelisting anything on the NSRL by hash.

GRR has the ability to import the NSRL. This function prepopulates the GRR datastore with all known hashes and reduces the need for GRR to collect these from the client systems. This can be done by downloading the latest quarterly release of the NSRL from NIST.

1. Download NSRL from NIST
2. Expand the zipped file containing hashes
3. Run "import_nsrl_hashes.py" with the appropriate configuration options

```
~/grr/tools# python import_nsrl_hashes.py --config /etc/grr/grr-server.yaml --
↪filename /media/<path to expanded NSRL>/NSRLFile.txt
Imported 5000 hashes
Imported 10000 hashes
Imported 15000 hashes
Imported 20000 hashes
Imported 25000 hashes
```

2.5.13 Glossary

AFF4

AFF4 is the data model used for storage in GRR, with some minor extensions. You can read about the usage in the GRR paper linked above and there is additional documentation on Forensics Wiki.

Agent

A platform-specific program that is installed on machines that one might want to investigate. It communicates with the GRR server and can perform client actions at the server's request.

Client

A system that has an agent installed. Also used to refer to the specific instance of an agent running in that system.

Client Action

A client action is an action that a client can perform on behalf of the server. It is the base unit of work on the client. Client actions are initiated by the server through flows. Example client actions are `ListDirectory`, `EnumerateFilesystems`, `Uninstall`.

Collection

A collection is a logical set of objects stored in the AFF4 database. Generally these are a list of URNs containing a grouping of data such as artifacts or events from a client.

Datastore

The backend is where all AFF4 and scheduler data is stored. It is provided as an abstraction to allow for replacement of the datastore without significant rewrite. The datastore supports read, write, querying and filtering.

Flow

A logical collection of server or client actions which achieve a given objective. A flow is the core unit of work in the GRR server. For example a `BrowserHistory` flow contains all the logic to download, extract and display browser history from a client. Flows can call other flows to get their job done. E.g. A `CollectBrowserHistory` flow might call `ListDirectory` and `GetFile` to do it's work. A flow is implemented as a class that inherits from `GRRFlow`.

Frontend server

Server-side component that sends and receives messages back and forth from clients.

Hunt

A hunt is a mechanism for managing the execution of a flow on a large number of machines. A hunt is normally used when you are searching for a specific piece of data across a fleet of machines. Hunts allow for monitoring and reporting of status.

Message

Transfer unit in GRR that transports information from a flow to a client and vice versa.

Worker

Once receiving a message from a client a worker will wake up the flow that requested its results and execute it.

2.6 Maintaining and tuning GRR deployment

2.6.1 Changing GRR server configuration

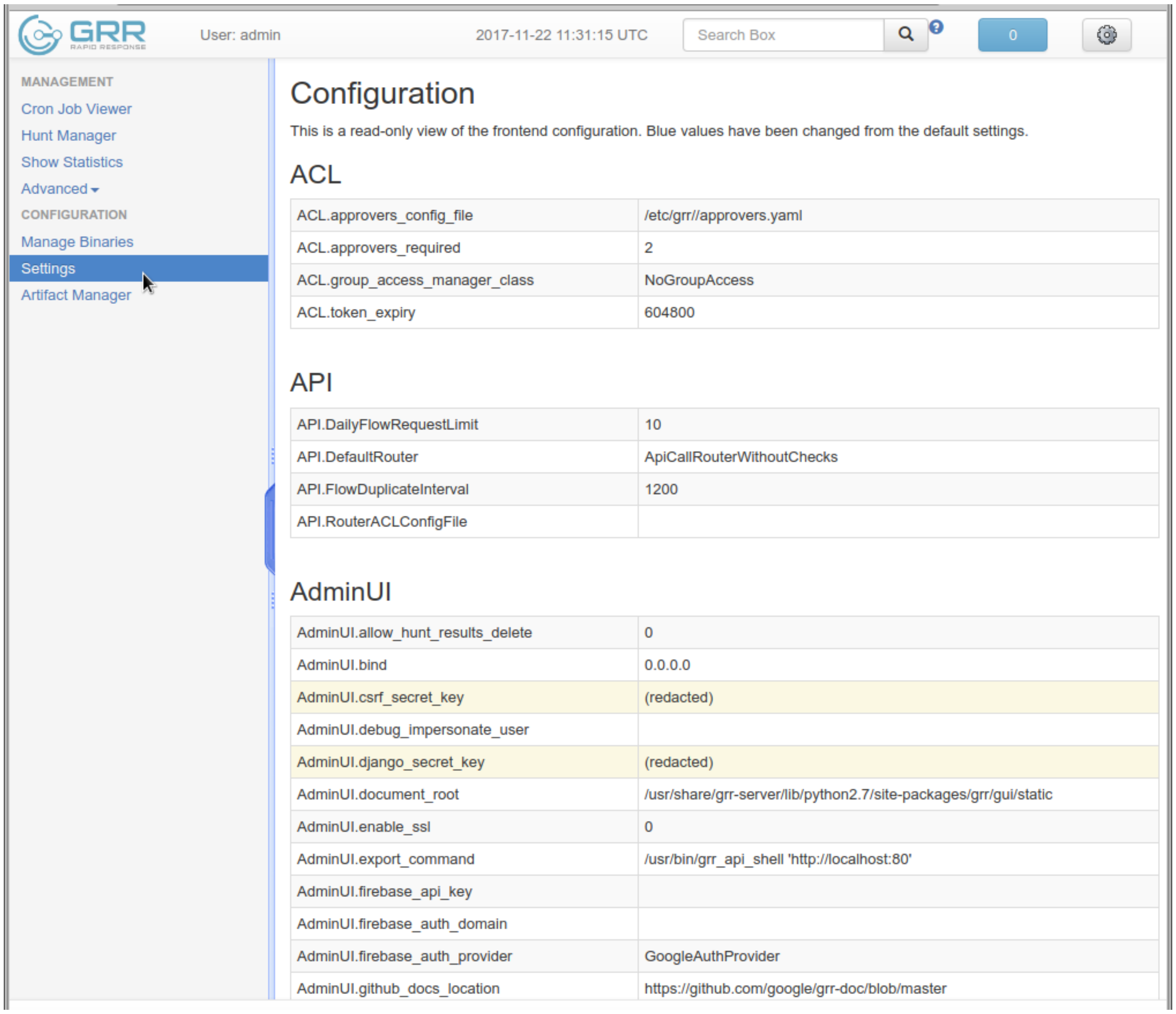
Configuration file organization

When GRR is installed on a system, a system wide, distribution specific, configuration file is also installed (by default `/etc/grr/grr_server.yaml`). This specifies the basic configuration of the GRR service (i.e. various distribution specific locations). However, the configuration typically needs to be updated with site specific parameters (for example new crypto keys, the `Client.control_urls` setting to allow the client to connect to the server, etc.)

In order to avoid overwriting the user customized configuration files when GRR is updated in future, we write site specific configuration files to a location specified by the `Config.writeback` config option (by default `/etc/grr/grr.local.yaml`). This local file only contains the parameters which are changed from the defaults, or the system configuration file.

> **Note**
>
> The system configuration file is never modified, all updated configurations are always written to the write-back location. Do not edit the system configuration file as changes in this file will be lost when GRR is upgraded.

You can see all available configuration options and their current values in the "Settings" pane in GRR's web UI.



Configuration

This is a read-only view of the frontend configuration. Blue values have been changed from the default settings.

ACL

ACL.approvers_config_file	/etc/grr//approvers.yaml
ACL.approvers_required	2
ACL.group_access_manager_class	NoGroupAccess
ACL.token_expiry	604800

API

API.DailyFlowRequestLimit	10
API.DefaultRouter	ApiCallRouterWithoutChecks
API.FlowDuplicateInterval	1200
API.RouterACLConfigFile	

AdminUI

AdminUI.allow_hunt_results_delete	0
AdminUI.bind	0.0.0.0
AdminUI.csrf_secret_key	(redacted)
AdminUI.debug_impersonate_user	
AdminUI.django_secret_key	(redacted)
AdminUI.document_root	/usr/share/grr-server/lib/python2.7/site-packages/grr/gui/static
AdminUI.enable_ssl	0
AdminUI.export_command	/usr/bin/grr_api_shell 'http://localhost:80'
AdminUI.firebase_api_key	
AdminUI.firebase_auth_domain	
AdminUI.firebase_auth_provider	GoogleAuthProvider
AdminUI.github_docs_location	https://github.com/google/grr-doc/blob/master

Configuration Contexts

The next important concept to understand is that of configuration contexts. GRR consists of many components, which share most of their configuration. Often, however, we want to specify slight differences between each component. If we were to write a separate configuration file say for building each client version (Currently, 3 operating systems, and 2 architectures) as well as for each component (e.g. worker, web UI, frontend etc), we would end up with an unmaintainable mess of many configuration files. It would make more sense to put most of the common configuration parameters in the same file, and only specify the different ones, depending on the component which is running.

GRR introduces the concept of running context. The context is a list of attributes about the currently running component. When the code accesses the configuration system, the configuration system considers the currently running context and returns the appropriate value of the required parameter.

For example, consider the parameter `Logging.path` which specifies the location of the log file. In a full GRR installation, we wish the GRR Admin UI to log to `/var/log/grr/adminui.log` while the frontend should log to `/var/log/grr/frontend.log`. We can therefore specify in the YAML config file:

```
Logging.path: /var/log/grr/grr_server.log

AdminUI Context:
  Logging.path: /var/log/grr/adminui.log

Frontend Context:
  Logging.path: /var/log/grr/httpserver.log

Client Context:
  Logging.path: /var/log/grr/grr_client.log

  Platform:Windows:
    Logging.path: c:\\windows\\system32\\grr.log
```

When the *AdminUI* program starts up, it populates its context with `AdminUI Context`. This forces the configuration system to return the value specific to this context. Note that other components, such as the worker will set a context of *Worker Context*, which will not match any of the above overrides. Therefore the worker will simply log to the default of `/var/log/grr/grr_server.log`.

Note that it is possible to have multiple contexts defined at the same time. So for example, GRR client running under windows will have the contexts, `Client Context` and `Platform:Windows`, while a GRR client running under Linux will have the context `Client Context` and `Platform:Linux`. When several possibilities can apply, the option which matches the most context parameters will be selected. So in the above example, linux and osx clients will have a context like `Client Context` and will select `/var/log/grr/grr_client.log`. On the other hand, a windows client will also match the `Platform:Windows` context, and will therefore select `c:\windows\system32\grr.log`.

The following is a non-exhaustive list of available contexts:

- *AdminUI Context*, *Worker Context*, *Frontend Context* etc. Are defined when running one of the main programs.
- *Test Context* is defined when running unit tests.
- *Arch:i386* and *Arch:amd64* are set when running a 32 or 64 bit GRR client binary.
- *Installer Context* is defined when the windows installer is active (i.e. during installation)
- *Platform:Windows*, *Platform:Linux*, *Platform:Darwin* are set when running GRR client under these platforms.

The full list can be seen here.

Parameter Expansion

The GRR configuration file format allows for expansion of configuration parameters inside other parameters. For example consider the following configuration file:

```
Client.name: GRR

Nanny.service_name: "%(Client.name)service.exe"
Nanny.service_key_hive: HKEY_LOCAL_MACHINE
Nanny.service_key: Software\\%(Client.name)
Nanny.child_command_line: |
  %(child_binary) --config "%(Client.install_path)\\%(Client.binary_name).yaml"
```

The expansion sequence `%(parameter_name)` substitutes or expands the parameter into the string. This allows us to build values automatically based on other values. For example above, the `Nanny.service_name` will be `GRRservice.exe` and the `service_key` will be set to `Software\GRR`

Expansion is very useful, but sometimes it gets in the way. For example, if we need to pass literal % escape sequences. Consider the Logging.format parameter which is actually a python format string:

```
Logging.format: \%(levelname\)s \%(module\)s:\%(lineno\)s] \%(message\)s
Logging.format: %{%(levelname)s %(module)s:%(lineno)s] %(message)s}
```

- This form escapes GRR's special escaping sequence by preceding both opening and closing sequences with the backslash character.
- This variation uses the literal expansion sequence *%{}* to declare the entire line as a literal string and prevent expansion.

Filtering

The configuration system may be extended to provide additional functionality accessible from the configuration file. For example consider the following configuration file:

```
Logging.path: "%(HOME|env)/grr.log"
```

This expansion sequence uses the *env* filter which expands a value from the environment. In this case the environment variable *HOME* will be expanded into the `Logging.path` parameter to place the log file in the user's home directory.

There are a number of additional interesting filters. The *file* filter allows including other files from the filesystem into the configuration file. For example, some people prefer to keep their certificates in separate files rather than paste them into the config file:

```
CA.certificate: "%(/etc/certificates/ca.pem|file)"
```

It is even possible to nest expansion sequences. For example this retrieves the client's private key from a location which depends on the client name:

```
Client.private_key: "%(/etc/grr/client/%(Client.name)/private_key.pem|file)"
```

2.6.2 Key management

Cryptographic Keys

Communication Security.

GRR communication happens using signed and encrypted protobuf messages. We use 2048 bit RSA keys to protect symmetric AES128 encryption. The security of the system does not rely on SSL transport for communication security. This enables easy replacement of the comms protocol with non-http mechanisms such as UDP packets.

The communications use a CA and server public key pair generated on server install. The CA public key is deployed to the client so that it can ensure it is communicating with the correct server. If these keys are not kept secure, anyone with MITM capability can intercept communications and take control of your clients. Additionally, if you lose these keys, you lose the ability to communicate with your clients.

Code Signing and CA Keys.

In addition to the CA and Server key pairs, GRR maintains a set of code signing signing keys. By default GRR aims to provide only read-only actions, this means that GRR is unlikely to modify evidence, and cannot trivially be used to take control of systems running the agent.

Note that read only access many not give direct code exec, but may well provide it indirectly via read access to important keys and passwords on disk or in memory!

However, there are a number of use cases where it makes sense to have GRR execute arbitrary code as explained in the section *Deploying Custom Code*.

As part of the GRR design, we decided that administrative control of the GRR server shouldn't trivially lead to code execution on the clients. As such we embed a strict whitelist of commands that can be executed on the client and we have a separate set of keys for code signing. For a binary to be run, the code has to be signed by the specific key and the client will confirm this signature before execution.

This mechanism helps give the separation of control required in some deployments. For example, the Incident Response team need to analyze hosts to get their job done, but deployment of new code to the platfrom is only done when blessed by the administrators and rolled out as part of standard change control. The signing mechanism allows Incident Response to react fast with new code if necessary, but only with the blessing of the Signing Key held by the platform administrator.

In the default install, the driver and code signing private keys are not passphrase protected. In a secure environment we strongly recommended generating and storing these keys off the GRR server and doing offline signing every time this functionality is required, or at a minimum setting passphrases which are required on every use. We recommend encrypting the keys in the config with PEM encryption, config_updater will then ask for the passphrase when they are used. An alternative is to keep a separate offline config that contains the private keys.

Key Generation

As state above, GRR requires multiple key pairs. These are used to:

- Sign the client certificates for enrollment.
- Sign and decrypt messages from the client.
- Sign code and binaries sent to the client.

These keys can be generated using the config_updater script normally installed in the path as grr_config_updater using the generate_keys command.

```
db@host:$ sudo grr_config_updater generate_keys
Generating executable signing key
..............+++
.....+++
Generating driver signing key
..................................................................+++
.............................................................+++
Generating CA keys
Generating Server keys
Generating Django Secret key (used for xsrf protection etc)
db@host:$
```

Adding a Passphrase

To Encrypt and add a password to the code & driver signing certificates

- Copy the keys (PrivateKeys.executable_signing_private_key & PrivateKeys.driver_signing_private_key) from the current GRR configuration file, most likely

  ```
  - /etc/grr/server.local.yaml
  ```

- Save these two keys as new text files temporarily. You'll need to convert the key text to the normal format by removing the leading whitespace and blank lines:

```
cat <original.exe.private.key.txt> | sed 's/^  //g' | sed '/^$/d' > <clean.exe.
↪private.key>
cat <original.driver.private.key.txt> | sed 's/^  //g' | sed '/^$/d' > <clean.driver.
↪private.key>
```

- Encrypt key and add a password

```
openssl rsa -des3 -in <clean.exe.private.key.txt> -out <exe.private.secure.key>
openssl rsa -des3 -in <clean.driver.private.key.txt> -out <driver.private.secure.key>
```

- Securely wipe all temporary files with cleartext keys.
- Replace the keys in the GRR config with the new encrypted keys (or store them offline). Ensure the server is restarted to load the updated configuration.

```
 PrivateKeys.executable_signing_private_key: '-----BEGIN RSA PRIVATE KEY-----

   Proc-Type: 4,ENCRYPTED

   DEK-Info: DES-EDE3-CBC,8EDA740783B7563C


   <start key text after *two* blank lines...>

   <KEY...>

   -----END RSA PRIVATE KEY-----'
```

Note In the YAML encoding,there **must** be an extra line between the encrypted PEM header and the encoded key. The key is double-spaces and indented two spaced exactly like all other keys in configuration file.

Alternatively, you can also keep your new, protected keys in files on the server and load them in the configuration using the file filter like this:

```
    PrivateKeys.executable_signing_private_key: %(<path_to_keyfile>|file)
```

Rotating the keys

Rotating Client Keys

The keys used by the clients to talk to the server do not have to be / cannot be rotated since the client id depends directly on the private key. Just removing the private key results in the client generating a new one but the client will also get a new client id.

Client keys used for signing executables and drivers can be changed in the config as described in the documentation about keys but

- Clients need to be repacked and redeployed so they can use the new keys.
- Old clients on the fleet will not be able to use the new keys anymore so there will be some issues during transition.

Rotating Server Keys

Issuing a new server key can be done using the

```
grr_config_updater rotate_server_key
```

command. This will update the server configuration with a new private key that is signed by the same CA. Existing clients will not lose connectivity by changing those keys even though old clients might need to be restarted before they will accept the new server credentials.

Note Changing the server key will invoke GRRs certificate revocation process. All clients that have seen the new server certificate will from that point on refuse to accept the previous key so it's not possible to revert to the old key.

2.6.3 User management

Users in GRR

Concept

GRR has a concept of users of the system. The GUI supports authentication and this verfication of user identity is used in all auditing functions (so for example GRR can properly record which user accessed which client, and who executed flows on clients).

A GRR user may be marked as "admin". This is only important if the approval-based workflow is turned on, since only "admin" users can approve hunts.

To add the user joe as an admin:

```
db@host:~$ sudo grr_config_updater add_user joe
Using configuration <ConfigFileParser filename="/etc/grr/grr-server.conf">
Please enter password for user 'joe':
Updating user joe

Username: joe
Labels:
Password: set
```

To list all users:

```
db@host:~$ sudo grr_config_updater show_user
Using configuration <ConfigFileParser filename="/etc/grr/grr-server.conf">

Username: test
Labels:
Password: set

Username: admin
Labels: admin
Password: set
```

To update a user (useful for setting labels or for changing passwords):

```
db@host:~$ sudo grr_config_updater update_user joe --add_labels admin,user
Using configuration <ConfigFileParser filename="/etc/grr/grr-server.conf">
Updating user joe

Username: joe
Labels: admin,user
Password: set
```

Authentication

The AdminUI uses HTTP Basic Auth authentication by default, based on the passwords within the user objects stored in the data store, but we **don't expect you to use this in production** (see Securing Access for more details).

There is so much diversity and customization in enterprise authentication schemes that there isn't a good way to provide a solution that works for a majority of users. Large enterprises most likely already have internal webapps that use authentication, this is just one more. Most people have found the easiest approach is to sit Apache (or similar) in front of the GRR Admin UI as a reverse proxy and use an existing SSO plugin that already works for that platform. Alternatively, with more work you can handle auth inside GRR by writing a Webauth Manager (`AdminUI.webauth_manager` config option) that uses an SSO or SAML based authentication mechanism.

Running GRR UI behind Apache

Running apache as a reverse proxy in front of the GRR admin UI is a good way to provide SSL protection for the UI traffic and also integrate with corporate single sign on (if available), for authentication.

To set this up:

- Buy an SSL certificate, or generate a self-signed one if you're only testing.
- Place the public key into "/etc/ssl/certs/" and ensure it's world readable

```
chmod 644 /etc/ssl/certs/grr_ssl_certificate_filename.crt
```

- Place the private key into "/etc/ssl/private" and ensure it is **NOT** world readable

```
chmod 400 /etc/ssl/private/grr_ssl_certificate_filename.key
```

- Install apache2 and required modules

```
apt-get install apache2
a2enmod proxy
a2enmod ssl
a2enmod proxy_http
```

- Disable any default apache files currently enabled (probably 000-default.conf, but check for others that may interfere with GRR)

```
a2dissite 000-default
```

- Redirect port 80 HTTP to 443 HTTPS
- Create the file "/etc/apache2/sites-available/redirect.conf" and copy the text below into it.

```
    <VirtualHost *:80>
        Redirect "/" "https://<your grr adminUI url here>"
    </VirtualHost>
```

- Create the file "/etc/apache2/sites-available/grr_reverse_proxy.conf" and copy the text below into it.

```
<VirtualHost *:443>
SSLEngine On
SSLCertificateFile /etc/ssl/certs/grr_ssl_certificate_filename.crt
SSLCertificateKeyFile /etc/ssl/private/grr_ssl_certificate_filename.key
ProxyPass / http://127.0.0.1:8000/
ProxyPassReverse / http://127.0.0.1:8000/
</VirtualHost>
```

- Enable the new apache files

```
a2ensite redirect.conf
a2ensite grr_reverse_proxy.conf
```

- Restart apache

```
service apache2 restart
```

NOTE: This reverse proxy will only proxy the AdminUI. It will have no impact on the agent communications on port 8080. It is advised to restrict access to the AdminUI at the network level.

Limiting access to GRR UI/API with API routers

GRR has a notion of *API call routers*. Every API request that GRR server handles is processed in a following fashion:

1. A username of a user making the request is matched against rules in the router configuration file (as defined in the `API.RouterACLConfigFile` config option).
    - If router configuration file is not set, a default router is used. Default router is specified in the `API.DefaultRouter` configuration option.
    - If one of the rules in the `API.RouterACLConfigFile` matches, then a router from the matching rule is used.
2. The API router is used to process the request.

GRR has a few predefined API routers:

- *DisabledApiCallRouter* - a router that will return an error for all possible requests.
- *ApiCallRouterWithoutChecks (default)* - a router that performs no access checks and just gives blanket access to the system.
- *ApiCallRouterWithApprovalChecks* - a router that enables GRR approvals-based workflow for better auditing. See Approval-based auditing for more details.
- *ApiCallRobotRouter* - a router that provides limited access to the system that is suitable for automation. Normally used together with *ApiCallRouterWithApprovalChecks* so that scripts and robots can perform actions without getting approvals.

Various configuration scenarios

All GRR users should have unrestricted access *(default)*

In GRR server configuration file:

```
API.DefaultRouter: ApiCallRouterWithoutChecks
```

All GRR users should follow audit-based workflow

In GRR server configuration file:

```
API.DefaultRouter: ApiCallRouterWithApprovalChecks
```

GRR users should follow audit-based workflow, but user "john" should have blanket access.

In GRR server configuration file:

```
API.RouterACLConfigFile: /etc/grr_api_acls.yaml
API.DefaultRouter: ApiCallRouterWithApprovalChecks
```

In /etc/grr_api_acls.yaml:

```
router: "ApiCallRouterWithoutChecks"
users:
  - "john"
```

NOTE: for example, you can set up user 'john' as a robot user, so that his credentials are used by scripts talking to the GRR API.

2.6.4 Email configuration

This section assumes you have already installed an MTA, such as Postfix or nullmailer. After you have successfully tested your mail transfer agent, please proceed to the steps outlined below.

To configure GRR to send emails for reports or other purposes:

Ensure email settings are correct by running back through the configuration script if needed (or by checking /etc/grr/server.local.yaml):

```
grr_config_updater initialize
```

Edit /etc/grr/server.local.yaml to include the following at the end of the file:

```
Worker.smtp_server: <server>
Worker.smpt_port: <port>
```

and, if needed,

```
Worker.smtp_starttls: True
Worker.smtp_user: <user>
Worker.smtp_password: <password>
```

After configuration is complete, restart the GRR worker(s). You can test this configuration by running *OnlineNotification" flow on any of the online clients (in the "Administrative" category). This flow sends an email to a given email address as soon as the client goes online. If the client is already online, the email will be sent next time it checks in (normally within 10 minutes).

2.6.5 Approval-based workflow

GRR has support for an approval-based access control system. To turn it on, set the *API.DefaultRouter* configuration option in the server configuration (see API routers documentation for more details):

```
API.DefaultRouter: ApiCallRouterWithApprovalChecks
```

Approval-based workflow places certain restrictions on what users can do without getting an explicit approval from another user.

Actions that don't require an approval

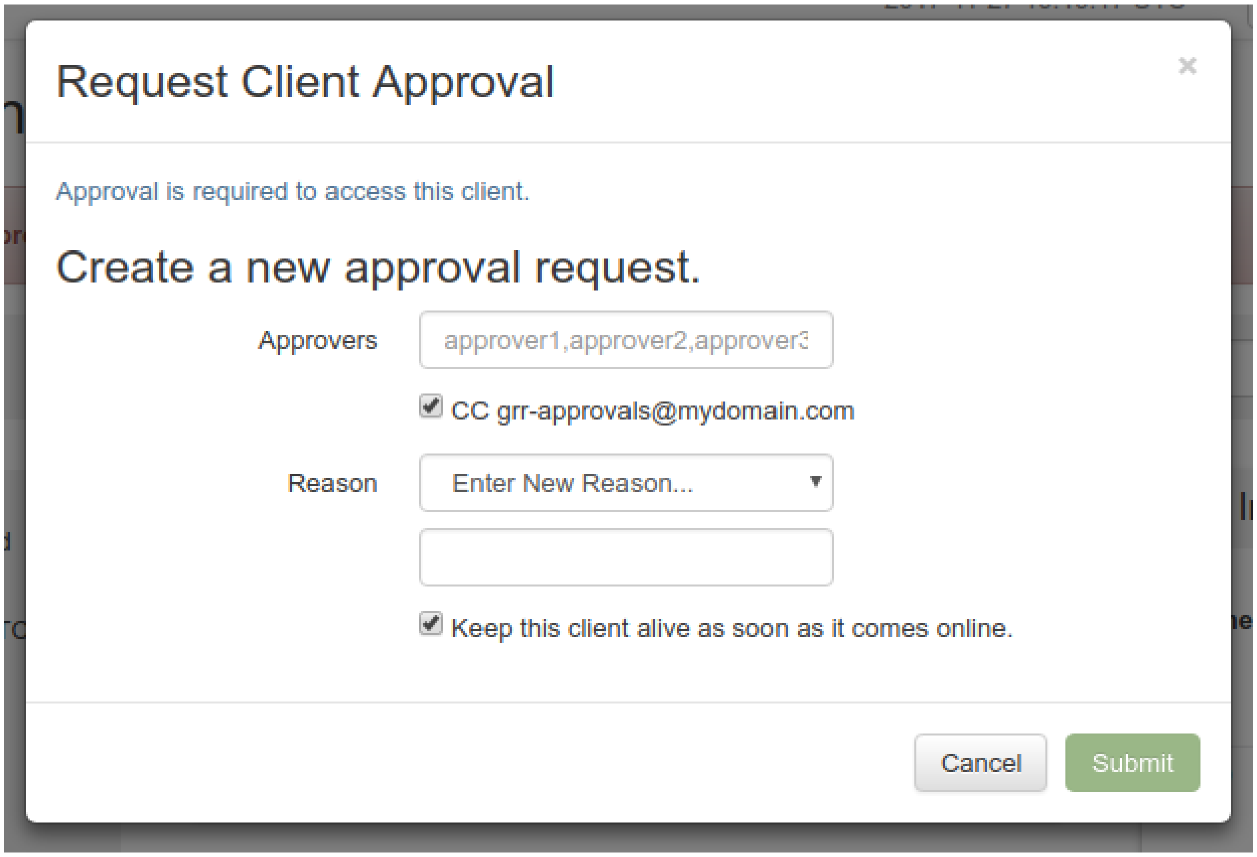
1. Searching for clients.
2. Getting information about a client host (including the history).
3. Looking at the list of hunts.
4. Looking at hunt results, logs and other related data.
5. Looking at cron jobs.

Actions that do require an approval

1. Browsing client's Virtual File System.
2. Inspecting client's flows.
3. Starting new flows on a client.
4. Starting a new hunt (**NOTE**: this is a special case - starting a hunt requires an approval from an admin user).
5. Downloading files collected by a hunt.
6. Starting a new cron job.

Requesting an approval

Whenever you try to perform an action that requires an approval in the GRR web UI, you'll see a popup dialog similar to this one:


Request Client Approval

Approval is required to access this client.

Create a new approval request.

Approvers: approver1,approver2,approver3

[✓] CC grr-approvals@mydomain.com

Reason: Enter New Reason...

[✓] Keep this client alive as soon as it comes online.

You need to fill-in a comma-separated list of approvers. Each approver will get a notification about a pending approval request in their GRR web UI.

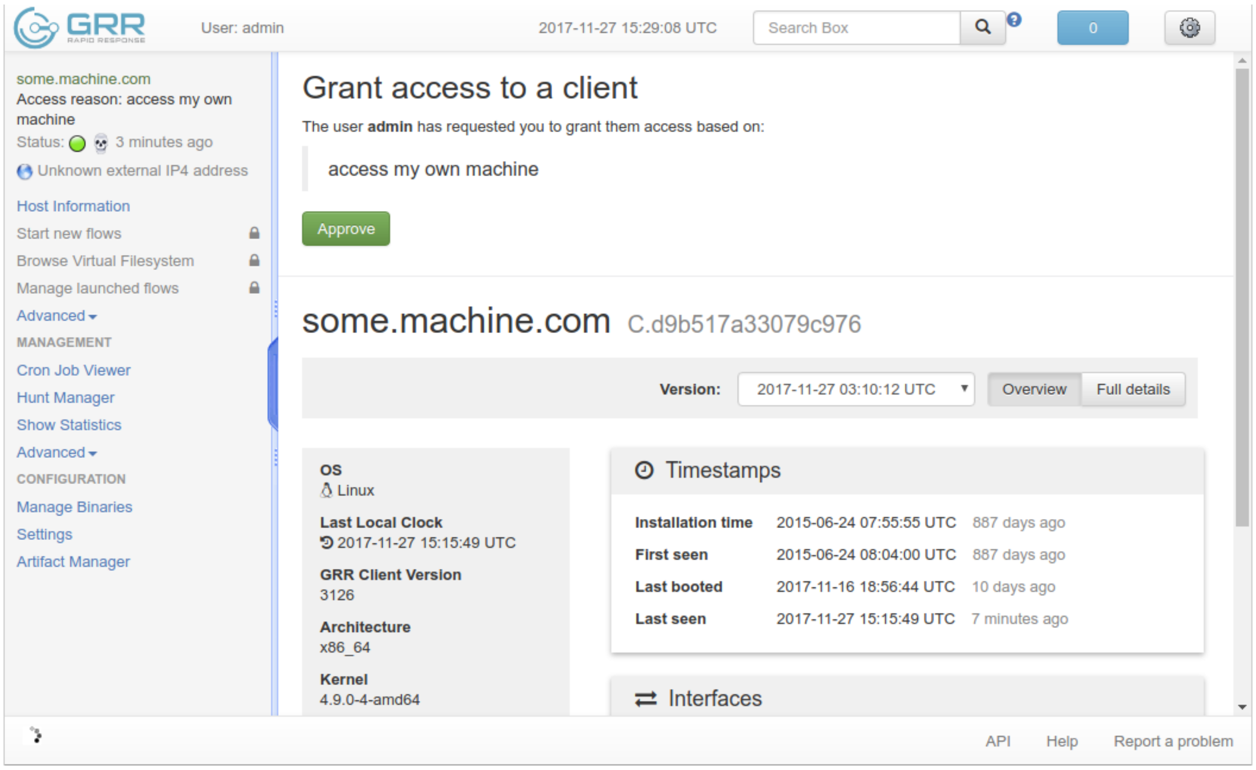
You can also select *CC* option to send an email to a predefined email address. This helps if you have multiple possible approvers and they're all subscribed to the same mailing list. *CC* option will only work if you have a proper Email Configuration in your GRR server config.

For every approval you also need to enter a reason you request one. This is done for auditing purposes. Also, when you access a client that you have an approval for, you'll see the reason displayed in GRR web UI's top left corner.

Granting an approval

Whenever somebody requests an approval, you'll get a notification in the GRR web UI: the notification button in the top right corner will turn red and display a number of pending notifications.

When you click on a *Please grant access to* <...> notification, you'll see an approval request review page similar to this one:


User: admin
2017-11-27 15:29:08 UTC

some.machine.com
Access reason: access my own machine
Status: 3 minutes ago
Unknown external IP4 address

Grant access to a client

The user **admin** has requested you to grant them access based on:

access my own machine

some.machine.com C.d9b517a33079c976

Version: 2017-11-27 03:10:12 UTC

OS
Linux
Last Local Clock
2017-11-27 15:15:49 UTC
GRR Client Version
3126
Architecture
x86_64
Kernel
4.9.0-4-amd64

Timestamps

Installation time	2015-06-24 07:55:55 UTC	887 days ago
First seen	2015-06-24 08:04:00 UTC	887 days ago
Last booted	2017-11-16 18:56:44 UTC	10 days ago
Last seen	2017-11-27 15:15:49 UTC	7 minutes ago

You can see the approval request reason on the review page and also some details about the object that the requestor is trying to access (a client, a hunt or a cron job). If you find the approval request justified, you click on the "Approve" button. The requestor will receive a notification then stating that the access was granted.

2.6.6 Repacking GRR clients

The client can be customized for deployment. There are two key ways of doing this:

1. Repack the released client with a new configuration.
2. Rebuild the client from scratch (advanced users); see Building custom client templates.

Doing a rebuild allows full reconfiguration, changing names and everything else. A repack on the other hand limits what you can change. Each approach is described below.

Repacking the Client with a New Configuration.

Changing basic configuration parameters can be done by editing the server config file (/etc/grr/server.local.yaml) to override default values, and then using the config_updater to repack the binaries. This allows for changing basic configuration parameters such as the URL the client reports back to.

Once the config has been edited, you can repack all clients with the new config and upload them to the datastore using

```
$ grr_config_updater repack_clients
```

Repacking works by taking the template zip files, injecting relevant configuration files, and renaming files inside the zip to match requested names. This template is then turned into something that can be deployed on the system by using the debian package builder on linux, creating a self extracting zip on Windows, or creating an installer package on OSX.

After running the repack you should have binaries available in the UI under manage binaries → installers and also on the filesystem under `grr/executables`.

Repacking Clients: Signing installer binaries

You can also use the grr_client_build tool to repack individual templates and control more aspects of the repacking, such as signing. For signing to work you need to follow these instructions:

Setting up for RPM signing

Linux RPMs are signed following a similar process to windows. A template is built inside the vagrant VM and the host does the repacking and signing.

To get set up for signing, first you need to create a gpg key that you will use for signing (here's a decent HOWTO).

Then make sure you have rpmsign and rpm utilities installed on your host system:

```
sudo apt-get install rpm
```

Tell GRR where your public key is:

```
sudo ln -s /path/to/mykey.pub /etc/alternatives/grr_rpm_signing_key
```

Set this config variable to whatever you used as your key name:

```
ClientBuilder.rpm_gpg_name: my key name
```

That's it, you can follow the normal build process.

Setting up for Windows EXE signing

Windows licensing means we can't just simply provide a build vm via vagrant as we've done for linux. So there's more hoops to jump through here, but it's as automated as possible.

Building Templates

First you need to make sdists from the GRR source (which requires a protobuf compiler) and get them to the windows build machine. We do this on linux with this script which uses google cloud storage to copy the files to the windows machine (note you'll need to use a different cloud storage bucket name):

```
BUCKET=mybucketname scripts/make_sdist_for_templates.sh
```

On your Windows/VM with git and the Google cloud SDK installed, run this as admin:

```
mkdir C:\grr_src
git clone https://github.com/google/grr.git C:\grr_src
C:\grr_src\vagrant\windows\install_for_build.bat
```

Then as a regular user you can download the sdists and build the templates from that:

```
C:\Python27-x64\python.exe C:\grr_src\vagrant\windows\build_windows_templates.py --
↪grr_src=C:\grr_src --cloud_storage_sdist_bucket=mybucketname --cloud_storage_output_
↪bucket=mybucketname
```

Download the built templates and components from cloud storage to your linux vm ready for repacking. Put them under `grr/executables/windows/templates`.

Setting Up For Windows EXE Signing

To make automation easier we now sign the windows installer executable on linux using osslsigncode. To set up for signing, install osslsigncode:

```
sudo apt-get install libcurl4-openssl-dev
wget http://downloads.sourceforge.net/project/osslsigncode/osslsigncode/osslsigncode-
↪1.7.1.tar.gz
tar zxvf osslsigncode-1.7.1.tar.gz
cd osslsigncode-1.7.1/
./configure
make
sudo make install
```

Get your signing key into the .pvk and .spc format, example commands below (will vary based on who you buy the signing cert from):

```
openssl pkcs12 -in authenticode.pfx -nocerts -nodes -out key.pem
openssl rsa -in key.pem -outform PVK -pvk-strong -out authenticode.pvk
openssl pkcs12 -in authenticode.pfx -nokeys -nodes -out cert.pem
cat Thawte_Primary_Root_CA_Cross.cer >> cert.pem
openssl crl2pkcs7 -nocrl -certfile cert.pem -outform DER -out authenticode.spc
shred -u key.pem
```

Link to wherever your key lives. This allows you to keep it on removable media and have different people use different keys with the same grr config.

```
sudo ln -s /path/to/authenticode.pvk /etc/alternatives/grr_windows_signing_key
sudo ln -s /path/to/authenticode.spc /etc/alternatives/grr_windows_signing_cert
```

Repacking Clients - Follow-up

After doing the above, add the –sign parameter to the repack command:

```
grr_client_build repack --template path/to/grr-response-templates/templates/grr_3.1.0.
↪2_amd64.xar.zip --output_dir=/tmp/test --sign
```

To repack and sign multiple templates at once, see the next section.

Repacking Clients With Custom Labels: Multi-Organization Deployments

Each client can have a label "baked in" at build time that allows it to be identified and hunted separately. This is especially useful when you want to deploy across a large number of separate organisations. You achieve this by creating a config file that contains the unique configuration you want applied to the template. A minimal config that just applies a label would contain:

```
Client.labels: [mylabel]
```

and be written into repack_configs/mylabel.yaml.

Then you can call repack_multiple to repack all templates (or whichever templates you choose) with this configuration and any others in the repack_configs directory. An installer will be built for each config:

```
grr_client_build repack_multiple --templates /path/to/grr-response-templates/
↪templates/*.zip --repack_configs /path/to/repack_configs/*.yaml --output_dir=/grr_
↪installers
```

To sign the installers (RPM and EXE), add –sign.

Client Configuration.

Configuration of the client is done during the packing/repacking of the client. The process looks like:

1. For the client we are packing, find the correct context and platform, e.g. `Platform:  Windows Client Context`
2. Extract the relevant configuration parameters for that context from the server configuration file, and write them to a client specific configuration file e.g. `GRR.exe.yaml`
3. Pack that configuration file into the binary to be deployed.

When the client runs, it determines the configuration in the following manner based on `--config` and `--secondary_configs` arguments that are given to it:

1. Read the config file packed with the installer, default: `c:\windows\system32\GRR\GRR.exe.yaml`
2. GRR.exe.yaml reads the Config.writeback value, default: `reg://HKEY_LOCAL_MACHINE/Software/GRR`
3. Read in the values at that registry key and override any values from the yaml file with those values.

Most parameters are able to be modified by changing parameters and then restarting GRR. However, some configuration options, such as `Client.name` affect the name of the actual binary itself and therefore can only be changed with a repack on the server.

Updating a configuration variable in the client can be done in multiple ways:

1. Change the configuration on the server, repack the clients and redeploy/update them.
2. Edit the yaml configuration file on the machine running the client and restart the client.

3. Update where Config.writeback points to with new values, e.g. by editing the registry key.
4. Issue an UpdateConfig flow from the server (not visible in the UI), to achieve 3.

As an example, to reduce how often the client polls the server to every 300 seconds, you can update the registry as per below, and then restart the service:

```
C:\Windows\System32\>reg add HKLM\Software\GRR /v Client.poll_max /d 300

The operation completed successfully.
C:\Windows\System32\>
```

2.6.7 Low-level maintenance with grr_console

GRR is shipped with a *grr_console* binary. This is effectively an IPython shell that gets initialized with a few basic GRR objects and offers you direct access to internal GRR classes.

In cases when there's no proper tool or UI or API for performing a certain task, grr_console may be useful. However, given that it exposes undocumented GRR internals and gives you complete unrestricted and unaudited access to GRR datastore, it's important to remember that:

1. It's very easy to shoot yourself in the foot and make the system unusable by running a wrong piece of code in *grr_console*.
2. GRR is a project that's being actively developed. Class names, import paths, interfaces are changing all the time. Even if you have a snippet of code that works in *grr_console* today, it may stop working in a month.

2.6.8 Choosing GRR datastore

When installing GRR, you can choose between *SQLiteDataStore* and *MySQLAdvancedDataStore*.

NOTE: Database choice is something that's normally done when GRR is installed. There's no documented way of migrating an existing deployment from one datastore to another.

SQLiteDataStore *(for small/demo deployments only)*

SQLiteDataStore is based on SQLite, meaning zero-effort setup and maintenance. However, its performance characteristics are not comparable with MySQL, it's unlikely to handle a big load well. With SQLiteDataStore you're also limited to running all GRR components on a single machine, because they all will have to access database file directly.

MySQLAdvancedDataStore *(for production use)*

MySQLAdvancedDataStore requires a bit more effort to set up, as you need to create a database user for GRR and configure GRR with appropriate credentials. However, MySQLAdvancedDataStore is the datastore that can handle significant load and the only one that makes scalable GRR deployment possible: you can run individual GRR components (workers, frontends, admin UI) on separate machines and make them all connect to the same database running elsewhere.

2.6.9 Scaling GRR within a single server

For 3.1.0 and later, use a systemd drop-in override to control how many copies of each component you run on each machine. This can initially be done using:

```
sudo systemctl edit grr-server
```

which creates "/etc/systemd/system/grr-server.service.d/override.conf". You'll want to turn this into a template file and control via puppet or similar. An example override that just runs 3 workers looks like:

```
[Service]
ExecReload=
ExecReload=/bin/systemctl --no-block reload grr-server@worker.service grr-
↪server@worker2.service grr-server@worker3.service
ExecStart=
ExecStart=/bin/systemctl --no-block start grr-server@worker.service grr-
↪server@worker2.service grr-server@worker3.service
ExecStop=
ExecStop=/bin/systemctl --no-block stop grr-server@worker.service grr-server@worker2.
↪service grr-server@worker3.service
```

When starting multiple copies of the UI and the web frontend you also need to tell GRR which ports it should be using. So if you want 10 http frontends on a machine you would configure your systemd drop-in to start 10 copies and then set Frontend.port_max so that you have a range of 10 ports from Frontend.bind_port. (I.E. set Frontend.bind_port to 8080 and Frontend.port_max to 8089) You can then configure your load balancer to distribute across that port range. AdminUI.port_max works the same way for the UI.

2.6.10 Large Scale Deployment

The GRR server components should be distributed across multiple machines in any deployment where you expect to have more than a few hundred clients, or even smaller deployments if you plan on doing intensive hunting. The performance needs of the various components are discussed below, and some real-world example deployment configurations are described in the FAQ.

You should install the GRR package on all machines and use configuration management (chef, puppet etc.) to:

- Distribute the same grr-server.yaml to each machine
- Control how many of each component to run on each machine (see next section for details)

2.6.11 Component Performance Needs

- **Worker**: you will probably want to run more than one worker. In a large deployment where you are running numerous hunts it makes sense to run 20+ workers. As long as the datastore scales, the more workers you have the faster things get done. We previously had a config setting that forked worker processes off, but this turned out to play badly with datastore connection pools, the stats store, and monitoring ports so it was removed.
- **HTTP frontend**: The frontend http server can be a significant bottleneck. By default we ship with a simple http server, but this is single process, written in python which means it may have thread lock issues. To get better performance you will need to run multiple instances of the HTTP frontend behind a reverse HTTP proxy (i.e. Apache or Nginx). Assuming your datastore handles it, these should scale linearly.
- **Web UI**: The admin UI component is usually under light load, but you can run as many as you want for redundancy (you'll need to run them behind Apache or Nginx to load-balance the traffic). The more concurrent GRR users you have, the more instances you need. This is also the API server, so if you intend to use the API heavily run more.

2.6.12 Building custom client templates

After installing the GRR server components, you can build client templates with

```
grr_client_build build --output mytemplates
```

and repack them with

```
grr_client_build repack --template mytemplates/*.zip --output_dir mytemplates
```

The first step needs to be done on the client target platform while the repacking step happens on your GRR server machine that knows about the proper configuration (i.e., crypto credentials for the clients to use).

We have fully automated client builds for Linux, OSX and Windows. The scripts we use are mostly in the Vagrant directory on GitHub. Duplicating what we do there might give you custom clients with little effort.

2.6.13 Troubleshooting

The upstart/init.d scripts show no output

When I run an init.d script e.g. "/etc/init.d/grr-http-server start" it does not show me any output.

Make sure that the "START" parameter in the corresponding default file, e.g. "/etc/default/grr-http-server", has been changed to "yes".

I cannot start any/some of the GRR services using the init.d scripts

When it run an init.d script e.g. "/etc/init.d/grr-http-server start" it indicates it started the service although when I check with "/etc/init.d/grr-http-server status" it says it is not running.

You can troubleshoot by running the services in the foreground, e.g. to run the HTTP Front-end server in the foreground:

```
sudo grr_server --start_http_server --verbose
```

Any/some of the GRR services are not running correctly

Check if the logs contain an indication of what is going wrong.

Troubleshoot by running the services in the foreground, e.g. to run the UI in the foreground:

```
sudo grr_server --verbose --component frontend
```

Cannot open libtsk3.so.3

error while loading shared libraries: libtsk3.so.3: cannot open shared object file: No such file or directory

The libtsk3 library cannot be found in the ld cache. Check if the path to libtsk3.so.3 is in /etc/ld.so.conf (or equivalent) and update the cache:

```
sudo ldconfig
```

Cron Job view reports an error

Delete and recreate all the cronjobs using GRR console:

```
aff4.FACTORY.Delete("aff4:/cron", token=data_store.default_token)
from grr.server.aff4_objects import cronjobs
cronjobs.ScheduleSystemCronFlows(token=data_store.default_token)
```

Missing Rekall Profiles

If you get errors like:

```
Error loading profile: Could not load profile nt/GUID/ABC123...
Needed profile nt/GUID/ABC123... not found!
```

when using rekall, you're missing a profile (see the Rekall FAQ and blogpost for some background about what this means).

The simplest way to get this fixed is to add it into Rekall's list of GUIDs, which is of great benefit to the whole memory forensics community. You can do this yourself via a pull request on rekall-profiles, or simply email the GUID to rekall-discuss@googlegroups.com. Once it's in the public rekall server, the GRR server will download and use it automatically next time you run a rekall flow that requires that profile. If your GRR server doesn't have internet access you'll need to run the GetMissingProfiles function from the GRR console on a machine that has internet access and can access the GRR database, like this:

```
from grr.server import rekall_profile_server
rekall_profile_server.GRRRekallProfileServer().GetMissingProfiles()
```

2.7 Developing GRR

2.7.1 Setting Up a Development Environment

Navigate to the root GRR directory (or clone the repository if you have not done it yet):

```
git clone https://github.com/google/grr
cd grr/
```

Virtual environment

We strongly recommend setting up a Python virtual environment for the development in order to prevent your everyday environment from being corrupted with a work-in-progress code. If you are feeling adventurous you might omit all the steps related to the virtual environment.

Make sure that you have `virtualenv` installed. It should be available in repositories of all popular Linux distributions. For example, to install it on Ubuntu-based distros simply run:

```
sudo apt install virtualenv
```

To create a virtual environment you just execute the `virtualenv $DIR` command where `$DIR` is the directory where you want it to be placed. The rest of the manual will assume that the environment is created in `~/.virtualenv/GRR` like this:

```
virtualenv ~/.virtualenv/GRR
```

After creating the environment you have to activate it:

```
source ~/.virtualenv/GRR/bin/activate
```

It is also advised to make sure that we are running a recent version of `pip`:

```
pip install --upgrade pip
```

For more information about creating and managing your virtual environments refer to the `virtualenv` documentation.

Node.js environment

Because GRR offers a user interface component it also needs some JavaScript code to be built with Node.js. Fortunately, this can be done with `pip` inside our virtual environment so that your system remains uncluttered.

To install Node.js simply do:

```
pip install nodeenv
nodeenv -p --prebuilt
```

Because the `nodeenv` command modifies our virtual environment, we also need to reinitialize it with:

```
source ~/.virtalenv/GRR
```

Installing GRR packages

GRR is split into multiple packages. For the development we recommend installing all components. Assuming that you are in the root GRR directory run the following commands:

```
pip install -e .
pip install -e ./grr/config/grr-response-server
pip install -e ./grr/config/grr-response-client
pip install -e ./grr/config/grr-response-test
```

The `-e` (or `--editable`) flag passed to `pip` makes sure that the packages are installed in a "development" mode and any changes you make in your working directory are directly reflected in your virtual environment, no reinstalling is required.

Note: Some prerequisites might be necessary as described in installing from released pip packages.

Testing

Now you are ready to start the GRR development. To make sure that everything is set-up correctly follow the testing guide.

2.7.2 Running the tests

For running tests GRR uses the pytest framework.

Prerequisites

Make sure you have correctly set-up the development environment, especially the part about installing the `grr-response-test` package.

This setup is sufficient for running most of the tests. However, GRR also has a UI component written in Angular. For testing that part we use Selenium and ChromeDriver which need to be installed first. You can skip this part if you do not need to execute UI tests but we strongly recommend to run them as well.

On Debian-based distros simply install `chromium-driver` package:

```
sudo apt install chromium-driver
```

If there is no `chromium-driver` available in your repositories you may try installing Chromium browser and then downloading latest `chromium-driver` binary from the official website. After downloading unpack it somewhere and add it to your `$PATH` or just move it to `/usr/bin`.

Running the whole test suite

To run all the tests, navigate to the root GRR directory and simply execute:

```
pytest
```

This will automatically discover and execute all test cases.

Running tests in parallel

To use pytest to run tests in parallel, install the pytest-xdist plugin

```
pip install pytest-xdist
```

and run

```
pytest -n <number of cores to use>
```

Running the tests selectively

Running all the tests is reasonable when you want to test everything before publishing your code, but it is not feasible during development. Usually you just want to execute only tests in a particular directory, like so:

```
pytest grr/server/aff4_objects
```

Or just a particular file:

```
pytest grr/server/aff4_objects/filestore_test.py
```

Or just a particular test class:

```
pytest grr/server/aff4_objects/filestore_test.py::HashFileStoreTest
```

Or even just a single test method:

```
pytest grr/server/aff4_objects/filestore_test.py::HashFileStoreTest::testListHashes
```

Ignoring test cases

Some kind of tests are particularly slow to run. For example, all UI tests are based on running a real web browser instance and simulating its actions which is painfully sluggish and can be very annoying if we want to test non-UI code.

In this case we can skip all tests in particular directory using the `--ignore` flag, e.g.:

```
pytest --ignore=grr/gui/selenium_tests
```

This will run all the tests except the Selenium ones.

Benchmarks

Benchmarks are not really testing the code correctness so there is no point in running them every time we want to publish our code. This is why the test suite will not run them by default. However, sometimes they can be useful for testing the performance and sanity of our system.

In order to run the test suite including the benchmarks, pass the `--benchmark` option to the test runner:

```
pytest --benchmark
```

Debugging

If our tests are failing and we need to fix our code the Python Debugger can come in handy.

If you run pytest with `--pdb` flag then upon a failure the program execution will be halted and you will be dropped into the PDB shell where you can investigate the problem in the environment it occurred.

If you set breakpoints in your code manually using `pdb.set_trace()` you will notice a weird behaviour when running your tests. This is because pytest intercepts the standard input and output writes, breaking the PDB shell. To deal with this behaviour simply run tests with `-s` flag - it will prevent pytest from doing that.

More information

The functionalities outlined in this guide are just a tip of the pytest capabilities. For more information consult pytest's man page, check `pytest --help` or visit the pytest homepage.

2.7.3 Contributing Code

GRR is a somewhat complex piece of code. While this complexity is necessary to provide the scale and performance characteristics we need, it makes it more difficult to get involved and understand the code base. So just be aware that making significant changes in the core will be difficult if you don't have a software engineering background, or solid experience with python that includes use of things like pdb and profiling.

That said, we want your contribution! You don't need to be a veteran pythonista to contribute artifacts or parsers. But whatever your experience, we strongly recommend starting somewhere simple before embarking on core changes and reading our documentation. In particular we recommend these as good starting points:

- Build a standalone console script to perform the actions you want. A standalone console script won't benefit from being able to be run as a Collector or a Hunt, but it will get you familiar with the API, and an experienced developer can convert it into a fully fledged flow afterwards.
- Add to Artifacts or an existing flow. Many flows could do with work to detect more things or support additional platforms.

- Add a new parser to parse a new filetype, e.g. if you have a different Anti-virus or HIDS log you want to parse.

Contributor License Agreement

GRR is an opensource project released under the Apache License and you should feel free to use it in any way compatible with this. However, in order to accept changes into the GRR mainline repository we must ask that keep a signed a Google Contributor License Agreement on file.

Style

The Chromium and Plaso projects have some good general advice about code contributions that is worth reading. In particular, make sure you're communicating via the dev list before you get too far into a feature or bug, it's possible we're writing something similar or have already fixed the bug.

Code needs to conform to the Google Python Style Guide. Note that despite what the guide says, we use two-space indentation, and MixedCase instead of lower_underscore for function names since this is the internal standard. Two-space indentation also applies to CSS.

Setup

We use the github fork and pull review process to review all contributions.

First, fork the GRR repository by following the github instructions.

Now that you have a github.com/your-username/grr repository:

```
# Make a new branch for the bug/feature
$ git checkout -b my_shiny_feature

# Make your changes, add any new files
$ git add newmodule.py newmodule_test.py
```

When you're ready for review, sync your branch with upstream:

```
$ git fetch upstream
$ git merge upstream/master

# Fix any conflicts and commit your changes
$ git commit -a
$ git push origin HEAD
```

Use the GitHub Web UI to create and send the pull request. We'll review the change.

```
# Make review changes
$ git commit -a
$ git push origin HEAD
```

Once we're done with review we'll commit the pull request.

2.7.4 Implementation details

GRR Messages

On the wire, the client and server interchange messages. We term the messages sent from server to the client *requests*, while messages sent from the client to the server are *responses*. Requests sent to the client ask the client to perform some action, for example *ListDirectory*. We term these actions the *client action*. A single request may elicit multiple responses.

For example, the request message *ListDirectory* will elicit a response for each file in the directory (potentially thousands). Requests and responses are tracked using an incremental *request_id* and *response_id*.

In order to indicate when all responses have been sent for a particular request, the client sends a special STATUS message as the last response. In the event of errors, the message contains details of the error (including backtrace). If the action completed successfully, an OK status is returned.

Messages are encoded as GrrMessage protobufs:

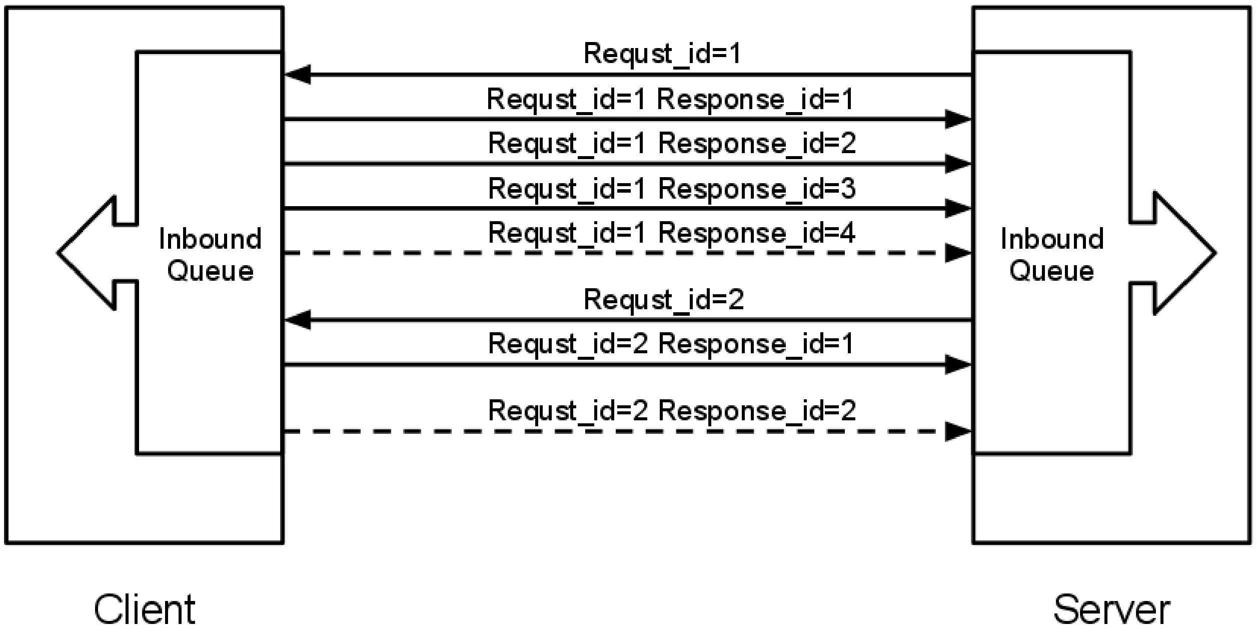

Requst_id=1
Requst_id=1 Response_id=1
Requst_id=1 Response_id=2
Requst_id=1 Response_id=3
Requst_id=1 Response_id=4
Requst_id=2
Requst_id=2 Response_id=1
Requst_id=2 Response_id=2
Inbound Queue
Inbound Queue
Client
Server

Figure 2 illustrates a typical sequence of messages. Request 1 was sent from the server to the client, and elicited 3 responses, in addition to a status message.

When the server sends the client messages, the messages are tagged in the data store with a lease time. If the client does not reply for these requests within the lease time, the requests become available for lease again. This is designed for the case of the client rebooting or losing connectivity part way through running the action. In this case, the request is re-transmitted and the action is run again.

AFF4 Data Model

AFF4 was first published in 2008 as an extensible, modern forensic storage format. The AFF4 data model allows the representation of arbitrary objects and the association of these with semantic meaning. The AFF4 data model is at the heart of GRR and is essential for understanding how GRR store, analyzes and represents forensic artifacts.

AFF4 is an object oriented model. This means that all entities are just different types of *AFF4 objects*. An AFF4 object is simply an entity, addressable by a globally unique name, which has attributes attached to it as well as behaviors.

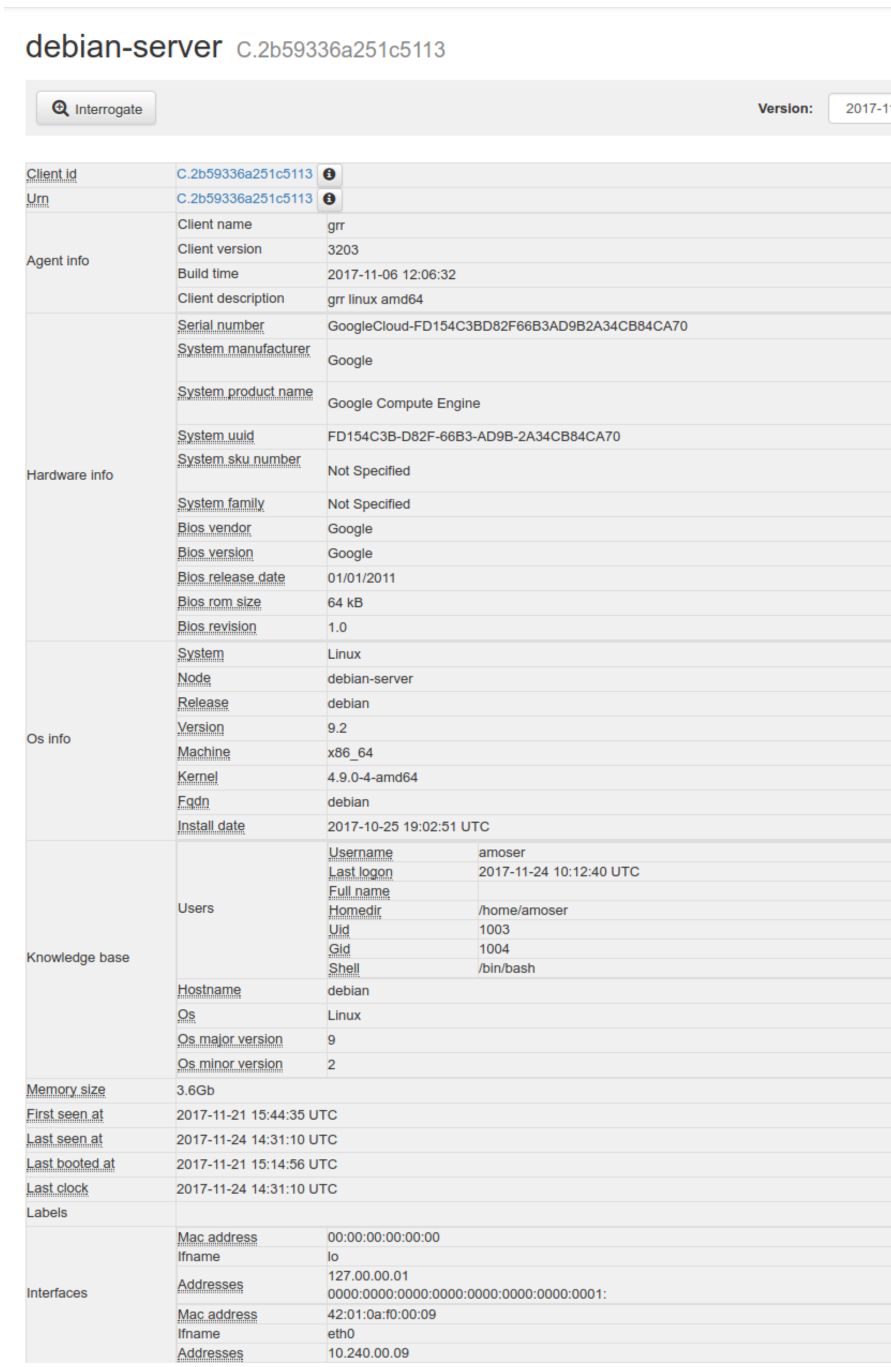
Each AFF4 object has a unique urn by which it can be addressed. AFF4 objects also have optional attributes which are defined in the object's Schema. For example consider the following definition of an AFF4 object representing a GRR Client:

```
class VFSGRRClient(aff4.AFF4Object):
  """A Remote client."""

  class SchemaCls(aff4.AFF4Object.SchemaCls):
    """The schema for the client."""
    CERT = aff4.Attribute("metadata:cert", RDFX509Cert,
                          "The PEM encoded cert of the client.")

    # Information about the host.
    HOSTNAME = aff4.Attribute("metadata:hostname", aff4.RDFString,
                              "Hostname of the host.", "Host",
                              index=client_index)
```

- An AFF4 object is simply a class which extends the AFF4Object base class.
- Each AFF4 object contains a Schema - in this case the Schema extends the base AFF4Object schema - this means this object can contains the attributes on the base class in addition to these attributes. Attributes do not need to be set.
- Attributes have both a name ("metadata:cert") as well as a type ("RDFX509Cert"). In this example, the VFS-GRRClient object will contain a CERT attribute which will be an instance of the type RDFX509Cert.
- An attribute can also be marked as ready for indexing. This means that whenever this attribute is updated, the corresponding index is also updated.



debian-server C.2b59336a251c5113

Interrogate

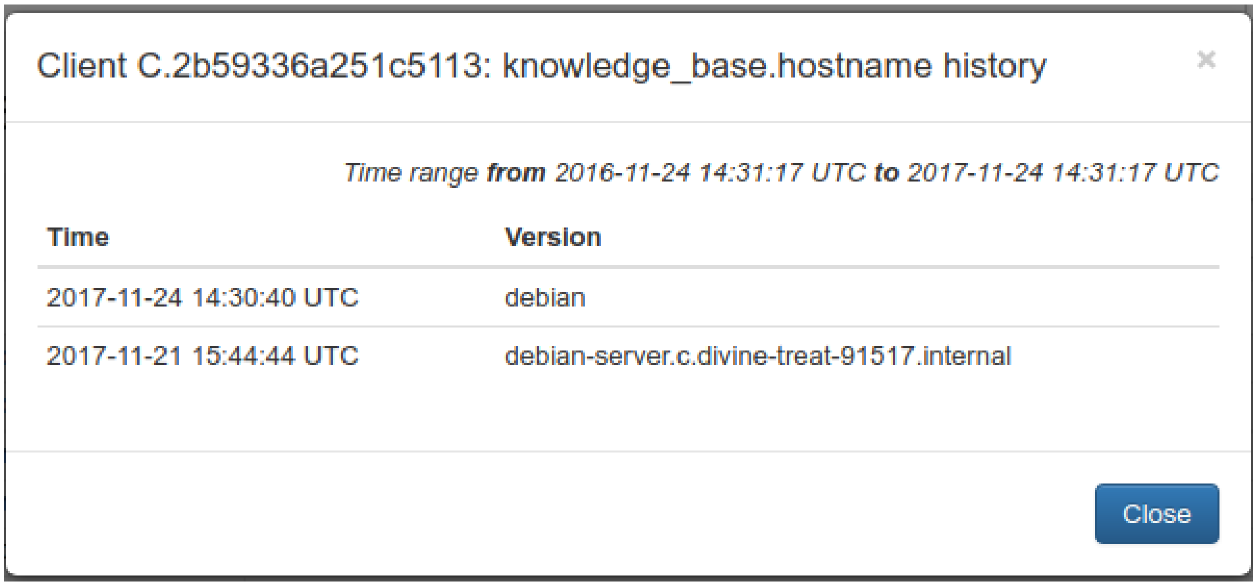
Version: 2017-1

Client id	C.2b59336a251c5113		
Urn	C.2b59336a251c5113		
Agent info	Client name	grr	
	Client version	3203	
	Build time	2017-11-06 12:06:32	
	Client description	grr linux amd64	
Hardware info	Serial number	GoogleCloud-FD154C3BD82F66B3AD9B2A34CB84CA70	
	System manufacturer	Google	
	System product name	Google Compute Engine	
	System uuid	FD154C3B-D82F-66B3-AD9B-2A34CB84CA70	
	System sku number	Not Specified	
	System family	Not Specified	
	Bios vendor	Google	
	Bios version	Google	
	Bios release date	01/01/2011	
	Bios rom size	64 kB	
	Bios revision	1.0	
Os info	System	Linux	
	Node	debian-server	
	Release	debian	
	Version	9.2	
	Machine	x86_64	
	Kernel	4.9.0-4-amd64	
	Fqdn	debian	
	Install date	2017-10-25 19:02:51 UTC	
Knowledge base	Users	Username	amoser
		Last logon	2017-11-24 10:12:40 UTC
		Full name	
		Homedir	/home/amoser
		Uid	1003
		Gid	1004
		Shell	/bin/bash
	Hostname	debian	
	Os	Linux	
	Os major version	9	
	Os minor version	2	
Memory size	3.6Gb		
First seen at	2017-11-21 15:44:35 UTC		
Last seen at	2017-11-24 14:31:10 UTC		
Last booted at	2017-11-21 15:14:56 UTC		
Last clock	2017-11-24 14:31:10 UTC		
Labels			
Interfaces	Mac address	00:00:00:00:00:00	
	Ifname	lo	
	Addresses	127.00.00.01 0000:0000:0000:0000:0000:0000:0000:0001:	
	Mac address	42:01:0a:f0:00:09	
	Ifname	eth0	
	Addresses	10.240.00.09	

Client with AFF4 attributes in the UI (larger image):

The figure above illustrates an AFF4 Object of type VFSGRRClient. It has a URN of "aff4:/C.2b59336a251c5113". The figure also lists all the attributes attached to this object.

Every attribute also has an *age* indicating the time when the attribute was created. Since GRR deals with fluid, constantly changing systems, each fact about the system must be tagged with the point in time where that fact was known. For example, at a future time, the hostname may change. In that case, we will have several versions for the HOSTNAME attribute, each correct for that point in time. We consider the entire object to have a new version when a versioned attribute changes.



Client C.2b59336a251c5113: knowledge_base.hostname history

Time range ***from*** *2016-11-24 14:31:17 UTC* ***to*** *2017-11-24 14:31:17 UTC*

Time	Version
2017-11-24 14:30:40 UTC	debian
2017-11-21 15:44:44 UTC	debian-server.c.divine-treat-91517.internal

AFF4 objects take care of their own serialization and unserialization and the data store technology is abstracted. Usually AFF4 objects are managed using the aff4 FACTORY:

```
In [8]: pslist = aff4.FACTORY.Open("aff4:/C.d74adcb3bef6a388/devices\
   /memory/pslist", mode="r", age=aff4.ALL_TIMES)

In [9]: pslist
Out[9]: <AFF4MemoryStream@7F2664442250 = aff4:/C.d74adcb3bef6a388/devices/memory/
↪pslist>

In [10]: print pslist.read(500)
 Offset(V) Offset(P)  Name                 PID    PPID   Thds   Hnds   Time
---------- ---------- -------------------- ------ ------ ------ ------ ---------------
↪----
0xfffffa8001530b30 0x6f787b30 System                  4      0     97    520  2012-
↪05-14 18:21:33
0xfffffa80027119d0 0x6e5119d0 smss.exe              256      4      3     33  2012-
↪05-14 18:21:34
0xfffffa8002ce3060 0x6dee3060 csrss.exe             332    324      9    611  2012-
↪05-14 18:22:25
0xfffffa8002c3

In [11]: s = pslist.Get(pslist.Schema.SIZE)

In [12]: print type(s)
<class 'grr.lib.aff4.RDFInteger'>

In [13]: print s
4938

In [14]: print s.age
2012-05-21 14:48:20

In [15]: for s in pslist.GetValuesForAttribute(pslist.Schema.SIZE):
   ....:     print s, s.age
4938 2012-05-21 14:48:20
4832 2012-05-21 14:20:30
4938 2012-05-21 13:53:05
```

- We have asked the aff4 factory to open the AFF4 object located at the unique location of *aff4:/C.d74adcb3bef6a388/devices/memory/pslist* for reading. The factory will now go to the data store, and retrieve all the attributes which comprise this object. We also indicate that we wish to examine all versions of all attributes on this object.
- We receive back an AFF4 object of type *AFF4MemoryStream*. This is a stream (i.e. it contains data) which stores all its content in memory.
- Since it is a stream, it also implements the stream interface (i.e. supports reading and seeking). Reading this stream gives back the results from running Volatility's pslist against the memory of the client.
- The SIZE attribute is attached to the stream and indicates how much data is contained in the stream. Using the Get() interface we retrieve the most recent one.
- The attribute is strongly typed, and it is an instance of an RDFInteger.
- The RDFInteger is able to stringify itself sensibly.
- All attributes carry the timestamp when they were created. The last time the SIZE attribute was updated was when the object was written to last.
- We can now retrieve all versions of this attribute - The pslist flow was run on this client 3 times at different dates. Each time the data is different.

Client Path Specifications

One of the nice things about the GRR client is the ability to nest file readers. For example, we can read files inside an image using the sleuthkit and also directly through the API. We can read registry keys using REGFI from raw registry files as well as using the API. The way this is implemented is using a pathspec.

Pathspecs

The GRR client has a number of drivers to virtualize access to different objects, creating a Virtual File System (VFS) abstraction. These are called *VFS Handlers* and they provide typical file-like operations (e.g. read, seek, tell and stat). It is possible to recursively apply different drivers in the correct order to arrive at a certain file like object. In order to specify how drivers should be applied we use *Path Specifications* or pathspecs.

Each VFS handler is constructed from a previous handler and a pathspec. The pathspec is just a collection of arguments which make sense to the specific VFS handler. The type of the handler is carried by the pathtype parameter:

- pathtype: OSImplemented by the grr.client.vfs_handlers.file module is a VFS Handler for accessing files through the normal operating system APIs.
- pathtype: TSKImplemented by the grr.client.vfs_handlers.sleuthkit module is a VFS Handler for accessing files through the sleuthkit. This Handle depends on being passed a raw file like object, which is interpreted as the raw device.

A pathspec is a list of components. Each component specifies a way to derive a new python file-like object from the previous file-like object. For example, image we have the following pathspec:

```
path:   /dev/sda1
pathtype: OS
nested_path: {
   path: /windows/notepad.exe
   pathtype: TSK
}
```

This opens the raw device /dev/sda1 using the OS driver. The TSK driver is then given the previous file like object and the nested pathspec instructing it to open the /windows/notepad.exe file after parsing the filesystem in the previous step.

This can get more involved, for example:

```
path:   /dev/sda1
pathtype: OS
nested_path: {
   path: /windows/system32/config/system
   pathtype: TSK
   nested_path: {
      path: SOFTWARE/MICROSOFT/WINDOWS/
      pathtype: REGISTRY
  }
}
```

Which means to use TSK to open the raw registry file and then REGFI to read the key from it (note that is needed because you generally cant read the registry file while the system is running).

Pathspec transformations

The pathspec tells the client exactly how to open the required file, by nesting drivers on the client. Generally, however, the server has no prior knowledge of files on the client, therefore the client needs to transform the server request to the pathspec that makes sense for the client. The following are the transformations which are applied to the pathspec by the client.

File Case Correction and path separator correction

Some filesystems are not case sensitive (e.g. NTFS). However they do preserve file cases. This means that the same pathspecs with different case filename will access the same file on disk. This file however, does have a well defined and unchanging casing. The client can correct file casing, e.g.:

```
path: c:\documents and settings\
pathtype: OS
```

Is corrected to the normalized form:

```
path: /c/Documents and Settings/
pathtype: OS
```

Filesystem mount point conversions

Sometimes the server requires to read a particular file from the raw disk using TSK. However, the server generally does not know where the file physically exists without finding out the mounted devices and their mount points. This mapping can only be done on the client at request runtime. When the top level pathtype is TSK, the client knows that the server intends to read the file through the raw interface, and therefore converts the pathspec to the correct form using the mount points information. For example:

```
path: /home/user/hello.txt
pathtype: TSK
```

Is converted to:

```
path: /dev/sda2
pathtype: OS
nested_path: {
      path: /user/hello.txt
      pathtype: TSK
}
```

UUIDs versus "classical" device nodes

External disks can easily get re-ordered at start time, so that path specifiers containing /dev/sd? etc. may not be valid anymore after the last reboot. For that reason the client will typically replace /dev/sda2 or similar strings with /dev/disk/by-uuid/[UUID] on Linux or other constructions (e.g. pathtype: uuid) for all clients.

Life of a client pathspec request

How are the pathspecs sent to the client and how are they related to the aff4 system. The figure below illustrates a typical request - in this case to list a directory:

1. A ListDirectory Flow is called with a pathspec of:

   ```
   path: c:\docume~1\bob\
   pathtype: OS
   ```

2. The flow sends a request for the client action ListDirectory with the provided pathspec.
3. Client calls VFSOpen(pathspec) which opens the file, and corrects the pathspec to:

   ```
   path: c:\Documents and Settings\Bob\
   pathtype: OS
   ```

4. Client returns StatResponse for this directory with the corrected pathspec.
5. The client AFF4 object maps the pathspec to an AFF4 hierarchy in the AFF4 space. The server flow converts from client pathspec to the aff4 URN for this object using the PathspecToURN() API. In this case a mapping is created for files read through the OS apis under **/fs/os/**. Note that the AFF4 URN created contains the case corrected - expanded pathspec:

   ```
   urn = GRRClient.PathspecToURN(pathspec)
   urn = aff4:/C.12345/fs/os/c/Documents and Settings/Bob
   ```

6. The server now creates this object, and stores the corrected pathspec as a STAT AFF4 attribute.

Client pathspec conversions can be expensive so the next time the server uses this AFF4 object for a client request, the server can simply return the client the corrected pathspec. The corrected pathspec has the LITERAL option enabled which prevents the client from applying any corrections.

Foreman

The Foreman is a client scheduling service. At a regular intervals (defaults to every 30 minutes) the client will report in asking if there are Foreman actions for it. At the time of this check in, the Foreman will be queried to decide if there are any jobs that match the host, if there are, appropriate flows will be created for the client. This mechanism is generally used by Hunts to schedule flows on a large number of clients.

The foreman maintains a list of rules, if the rule matches a client when it checks in, the specified flow will execute on the client. The rules work against AFF4 attributes allowing for things like “All XP Machines” or “All OSX machines installed after 01.01.2011”. For more information see the section about hunt rules.

The foreman check-in request is a special request made by the client that communicates with a Well Known Flow (W:Foreman). When the server sees this request it does the following:

1. Determines how long since this client did a Foreman check-in.
2. Determines the set of rules that are non-expired and haven’t previously been checked by the client.
3. Matches those rules against the client’s attributes to determine if there is a match.
4. If there is a match, run the associated flow.

The reason for the separate Foreman check-in request is that the rule matching can be expensive when you have a lot of clients, so having these less frequent saves a lot of processing.

Authorization and Auditing

GRR contains support for a full authorization and audit API (even for console users) and is implemented in an abstraction called a Security Manager. This Security Manager shipped with GRR, does not make use of these APIs and is open by default. However, a deployment may build their own Security Manager which implements the authorization semantics they require.

This infrastructure is noticeable throughout much of the code, as access to any data within the system requires the presence of a “token”. The token contains the user information and additionally information about the authorization of the action. This passing of the token may seem superfluous with the current implementation, but enables developers to create extensive audit capabilities and interesting modes of authorization.

By default, GRR should use data_store.default_token if one is not provided. To ease use this variable is automatically populated by the console if –client is used.

Token generation is done using the access_control.ACLToken.

```
token = access_control.ACLToken()
fd = aff4.FACTORY.Open("aff4:/C.12345/", token=token)
```

2.8 Release notes

Each release of GRR brings some significant changes as the project’s code is moving quickly. This section tries to identify the key changes and issues to watch for when upgrading from one version to another.

2.8.1 Server

3.1.0.2 to 3.2.0.1 (Sep 5 2017)

Starting with 3.2.0.1, we plan on releasing more frequently, since we have automated a large part of the release process. The recommended way of installing this version of GRR is with the server debian package, which includes client templates for all supported platforms. PIP packages, and a Docker image for the release will be made available at a later date.

- **IMPORTANT** Before upgrading, make sure to back up your data store. This release has changes that may be incompatible with older versions of the data store. Also make copies of your configs - for reference, and in case you want to roll back the upgrade.

- **WARN** Rekall is disabled by default. 'Rekall.enabled: True' line should be added to GRR configuration file in order for Rekall to work. There are known stability issues in Rekall and winpmem driver. Disabling it by default makes its 'in dev/experimental' status more explicit.
- **WARN** Components are deprecated. The GRR client is now monolithic, there's no need to upload or update components anymore. NOTE: The new GRR server will only be able to run Rekall-based flows (AnalyzeClient-Memory, MemoryCollector) with the latest GRR client, since it expects the client not to use components.
- **WARN** Rekall flows have been moved to 'DEBUG' category. They won't be visible unless users enable 'Mode: DEBUG' in the settings menu (accessible via the settings button in top right corner of GRR AdminUI).
- **WARN** A number of backwards-incompatible datastore schema changes were made. The nature of these changes is such that easy data migration between releases is not possible. Data loss of flows and hunts is possible. Please back up your current system if this data is critical.
- **WARN** A number of deprecated configuration options were removed. If GRR fails to start after the upgrade, please check the server logs for 'Unknown configuration option' messages.
- **WARN** Global Flows are no longer supported.
- **INFO** Bugfixes, additionals tests and code health work in all parts of the system.
- **INFO** HTTP API now covers 100% of GRR functionality. Python API client library significantly extended. API call routers implemented, allowing for flexible authorization.
- **INFO** 'Download as' functionality added to flows and hunts. Supported formats are CSV, YAML and SQLite.
- **INFO** Build process for clients and server is significantly improved. Client templates for all supported platforms, as well as a release-ready server deb, get built on every github commit.

0.3.0-7 to 3.1.0.2

Note that we shifted our version string left to reduce confusion, see the documentation on GRR versions.

- **WARN** Quite a few config options have changed. We recommend moving any customizations you have into your server.local.yaml, and using the main config file supplied with the new version. Some important specific ones are called out here.
- **WARN** Config option Client.control_urls has been replaced with Client.server_urls. You will need to make sure your polling URL is set in the new variable.
- **WARN** We have completely rewritten how memory collection and analysis works inside GRR. Rekall is now responsible for all driver loading, memory collection, and memory analysis. It also replaces the functionality of the Memory Collector flow. This means you **must** upgrade your clients to be able to do memory collection and analysis.
- **WARN** Repacking of old client templates no longer works due to changes necessary for repacking different versions. You will need to use the new client templates that include a build.yaml file.
- **WARN** Config option Cron.enabled_system_jobs (a whitelist) was replaced with Cron.disabled_system_jobs (a blacklist) to make adding new cronjobs easier. You will need to remove Cron.enabled_system_jobs and if you made customizations here, translate any jobs you want to remain disabled to the new list.
- **WARN** We changed the format for the Knowledge Base protocol buffer to remove a duplication of the users component which was storing users information twice and complicating code. This means that all clients need a fresh interrogate run to populate the machine information. Until the machines run interrogate the `%%users.homedir%%` and similar expansions won't work.
- **WARN** The compiler we use to build the Mac client template has become more aggressive with optimizations and now emits code optimized to specific processor versions. The mac client will not run on a processor older

than our build machine - an early 2011 Macbook with the Intel Sandy Bridge architecture. If you need to run GRR on, for example, Core 2 duo machines, you will need a custom client built on one of those machines.

- **WARN** We no longer support running the client on Windows XP, it's difficult to make it work and we have no use for an XP client ourselves. See here for our OS support statement.
- **INFO** Strict context checking was added for config files in this commit, which exposed a number of minor config typo bugs. If you get InvalidContextError on upgrade, you need to update your config to fix up the typos. The config/contexts.py file is the canonical reference.

Upgrade procedure:

1. Make backups of important data such as your configs and your database.
2. Upgrade to xenial
3. Clear out any old installations: `sudo rm -rf /usr/local/bin/grr_*; sudo rm -rf /usr/local/lib/python2.7/dist-packages/grr/`
4. Install deb package
5. Backup then remove your /etc/grr/grr-server.yaml. Make sure any customizations you made are stored in `/etc/grr/server.local.yaml`.
6. Run `grr_config_updater initialize`

0.3.0-6 to 0.3.0-7

- **INFO** We moved to using the open source binplist, rather than bundling it with GRR. If you are getting "ImportError: cannot import name binplist", make sure you have installed the binplist package and deleted grr/parsers/binplist* to clean up any .pyc files that might have been created.
- **INFO** If you have troubles with the rekall efilter library failing to import, try deleting all of your rekall-related .pyc's and make sure you're installing the version in requirements.txt. See this bug for full details.

0.3.0-5 to 0.3.0-6

This version bump was to keep in step with the client, which got a new version because of a memory collection bug.

- **WARN** The artifact format changed in a way that is not backwards compatible. You'll need to delete any artifacts you have in the artifact manager, either through the GUI or on the console with `aff4.FACTORY.Delete("aff4:/artifact_store")` before you upgrade. If you forget you should be able to use the console or you can roll back to this commit before the artifact change.
- **WARN** Keyword search is now based on an index that is built at weekly interrogate time, which means decomissioned clients won't appear in search results. To populate all of your historical clients into the search index, use the code snippet on this commit.

0.3.0 to 0.3.0-5

- **WARN** Rekall made some fundamental changes that mean rekall in the 0.3.0-5 server won't work with rekall in 3.0.0.3 clients, so a client upgrade to 3.0.0.5 is required. Non-rekall GRR components should continue to work.
- **WARN** The enroller component was removed, and is now handled by the worker. You will need to stop those jobs and delete the relevant upstart/init scripts.
- **INFO** We now check that all config options are defined in the code and the server won't start if they aren't. When you upgrade it's likely there is some old config that will need cleaning up. See the graveyard below for advice.

0.2.9 to 0.3.0

- **WARN** After install of new deb you will need to restart the service manually.
- **INFO** To get the new clients you will want to repack the new version with `sudo grr_config_updater repack_clients`. This should give you new Windows, Linux and OSX clients.
- **INFO** Client themselves will not automatically be upgraded but should continue to work.

0.2-8 to 0.2-9

- **WARN** After install of new deb you will need to restart the service manually.
- **WARN** Users have changed from being managed in the config file to being managed in the datastore. This means your users will not work. We haven't migrated them automatically, please re-add with `sudo grr_config_updater add_user <user>`
- **INFO** To get the new clients you will want to repack the new version with `sudo grr_config_updater repack_clients`. This should give you new Windows, Linux and OSX clients.
- **INFO** Client themselves will not automatically be upgraded but should continue to work.

2.8.2 Client

3.2.0.1 -

Starting from 3.2.0.1, we the same version string for the client and server, since they get released at the same time.

3.0.7.1 to 3.1.0.0

Note that we skipped some numbers to make versioning simpler and reduce confusion, see the documentation on GRR versions.

- **WARN** We changed rekall to be a independently updatable component in the client, which is a backwards incompatible change. You must upgrade your clients to 3.1.0.0 if you want to use memory capabilities in the 3.1.0 server.
- **WARN** Our previous debian package added the GRR service using both upstart and init.d runlevels. This caused some confusion on systems with ubuntu upstart systems. We detect and remove this problem automatically with the new version, but since it is a config file change you need to specify whether to install the new config or keep the old one, or you will get a config change prompt. New is preferred. Something like `sudo apt-get -o Dpkg::Options::="--force-confnew" grr`.

3.0.0.7 to 3.0.7.1

- **INFO** 3.0.7.1 is the first version of GRR that will work on OS X El Capitan. The new OS X System Integrity Protection meant we had to shift our install location from `/usr/lib/` to `/usr/local/lib`.
- **INFO** We changed our version numbering scheme for the client at this point to give us the ability to indicate multiple client versions that work with a server version. So 3.0.7.* clients will all work with server 0.3.0-7.

3.0.0.6 to 3.0.0.7

- **WARN** Linux and OS X clients prior to 3.0.0.7 were using `/etc/grr/client.local.yaml` as the local writeback location. For 3.0.0.7 this was changed to `/etc/%(Client.name).local.yaml` where the default is `/etc/grr.local.yaml`. If you wish to preserve the same client IDs you need to use platform management tools to copy the old config into the new location for all clients before you upgrade. If you don't do this the clients will just re-enrol and get new client IDs automatically.

3.0.0.5 to 3.0.0.6

- **INFO** 3.0.0.5 had a bug that broke memory collection, fixed in this commit. We also wrote a temporary server-side workaround, so upgrading isn't mandatory. 3.0.0.5 clients should still work fine.

3.0.0.3 to 3.0.0.5

(We skipped a version number, there's no 3.0.0.4)

3.0.0.2 to 3.0.0.3

- **WARN** A change to OpenSSL required us to sign our CSRs generated during the enrollment process. This wasn't necessary previously and provided no benefit for GRR so we had gained some speed by not doing it. Since new OpenSSL required it, we started signing the CSRs, but it meant that the 3.0.0.3 server will reject any 3.0.0.2 clients that haven't already enrolled (i.e. they will see a HTTP 406). Old 3.0.0.2 clients that have already enrolled and new 3.0.0.3 clients will work fine. This basically just means that you need to push out new clients at the same time as you upgrade the server.

2.8.3 Config Variable Graveyard

Sometimes config variables get renamed, sometimes removed. When this happens we'll try to record it here, so users know if local settings should be migrated/ignored etc.

You can verify your config with this (root is required to read the writeback config)

```
sudo PYTHONPATH=. python ./run_tests.py --test=BuildConfigTests.testAllConfigs
```

- AdminUI.team_name: replaced by Email.signature
- ClientBuilder.build_src_dir: unused, effectively duplicated ClientBuilder.source
- ClientBuilder.executables_path: ClientBuilder.executables_dir
- Client.config: unused. Now built from Client.config_hive and Client.config_key
- Client.config_file_name: unused
- Client.location: replaced by Client.control_urls
- Client.package_maker_organization: replaced by ClientBuilder.package_maker_organization
- Client.tempdir: replaced by Client.grr_tempdir and Client.tempdir_roots
- Email.default_domain: essentially duplicated Logging.domain, use that instead.
- Frontend.processes: unused
- Nanny.nanny_binary: replaced by Nanny.binary

- NannyWindows.* : replaced by Nanny.
- PyInstaller.build_root_dir: unused, effectively duplicated ClientBuilder.build_root_dir.
- Users.authentication: unused, user auth is now based on aff4:/users objects. Use config_updater to modify them.
- Worker.task_limit: unused
- Worker.worker_process_count: unused
- Cron.enabled_system_jobs (a whitelist) was replaced with Cron.disabled_system_jobs (a blacklist). Cron.enabled_system_jobs should be removed. Any custom jobs you want to stay disabled should be added to Cron.enabled_system_jobs.
- Client.control_urls: renamed to Client.server_urls.

2.9 Frequently Asked Questions

2.9.1 Who wrote GRR and Why?

GRR started at Google as a 20% project and gained in popularity until it became fully-supported and open sourced. The primary motivation was that we felt the state of the art for incident response was going in the wrong direction, and wasn't going to meet our cross platform, scalability, obfuscation or flexibility goals for an incident response agent.

Additionally, we believe that for things to progress in security, everyone has to up their game and improve their capabilities. We hope that by open sourcing GRR, we can foster development of new ways of doing things and thinking about the problem, get it into the hands of real organizations at reasonable cost, and generally push the state of the art forward.

We are getting significant code contributions from outside of Google, and have close relationships with a number companies running large-scale deployments.

2.9.2 Why is the project called GRR?

When using other tools, we found ourselves making the sound "grrr" a lot, and it just kind of stuck. GRR is a recursive acronym, in the tradition of GNU and it stands for GRR Rapid Response. Not GRR Response Rig or Google Rapid Response which it is sometimes mistaken for.

2.9.3 Is GRR production ready?

As of Aug 2015 GRR is running at large scale both inside and outside of Google. The largest opensource deployment we know of is roughly 30k machines, there's another company at around 15k, and quite a few around 2k. Google's deployment is bigger than all of those put together, although there are some differences in the codebase (see below).

2.9.4 Should I expect to be able to install and just start running GRR?

Yes, for basic use cases.

But if you want to do a large-scale enterprise deployment, it is probably best to think about GRR as a 80% written software project that you could invest in. The question then becomes: instead of investing X million in product Y to buy something, should I instead invest 25% of that in GRR and hire a dev to contribute, build and deploy it? On the one hand that gives you control and in-house support, on the other, it is a real investment of resources.

If you are selling GRR internally (or to yourself) as a free <insert commercial IR product here>, your expectations will be wrong, and you may get disillusioned.

2.9.5 Can the GRR team provide me with assistance in getting it setup?

The core GRR team cares about the open source project, but in the end, our main goals are to build something that works for us. We don't, and won't offer a helpdesk, professionally curated documentation, nor someone you can pay money to help you out if something goes wrong. We aren't providing feeds or consulting services, and have nothing direct to gain from offering the platform. If you need something pre-packaged and polished, GRR probably isn't right for you (at least in the short-medium term). For a large deployment you should expect to fix minor bugs, improve or write documentation, and actively engage with the team to make it successful.

If someone is willing to code, and has invested some time learning we will do what we can to support them. We're happy to spend time on VC or in person helping people get up to speed or running hackathons. However, the time that the developers invest in packaging, building, testing and debugging for open source is mostly our own personal time. So please be reasonable in what and how you ask for assistance. We're more likely to help if you've contributed documentation or code, or even filed good bug reports.

2.9.6 I'm interested in GRR but I, or my team need some more convincing. Can you help?

The core GRR team has invested a lot in the project, we think its pretty awesome, so the team happy to talk, do phone calls, or chat with other teams that are considering GRR. We've even been known to send house-trained GRR engineers to companies to talk with interested teams. Just contact us directly. You also can corner one of us, or at least someone from the team, or someone who works on GRR at most leading forensics/IR type conferences around the world.

2.9.7 I've heard that there are secret internal versions of GRR that aren't open sourced that may have additional capabilities. Is that true?

GRR was always designed to be open sourced, but with any sufficiently complex "enterprise" product you expect to integrate it with other systems and potentially even with proprietary technology. So its true that some of the core developers time is spent working on internal features that won't be released publicly. The goal is to ensure that everything useful is released, but there are some things that don't make sense. Below are listed some of the key differences that may matter to you:

- **Datastore/Storage**: At Google we run GRR on a Bigtable datastore (see below for more detail), but we have abstracted things such that using a different datastore is very simple. The SQLite and MySQLAdvanced datastores available to open source are actively used at real scale outside of Google.
- **Security and privacy**: The open source version has minimal controls immediately available for user authentication, multi-party authorization, privacy controls, logging, auditing etc. This is because these things are important enough for them to be custom and integrated with internal infrastructure in a large deployment. We open source the bits that make sense, and provide sensible hooks for others to use, but full implementations of these may require some integration work.
- **Machine handling and monitoring**: Much of the infrastructure for running and monitoring a scalable service is often built into the platform itself. As such GRR hasn't invested a lot in built-in service or performance monitoring. We expose a lot of statistics, but only display a small subset in the UI. We expect companies to have package deployment (SCCM/munki/casper/apt/yum etc.), config management (chef/puppet/salt etc.), and server monitoring (nagios/cacti/munin/zabbix/spiceworks etc.).

Differences will be whittled away over time as the core GRR team runs open source GRR deployments themselves. That means you can expect most of these things to become much less of an issue over time.

2.9.8 Should I choose MySQL Advanced or the HTTPDatastore + SQLite?

There are currently two supported datastore options, consider the following points when choosing which to use.

MySQL Advanced:

- You can buy support, and some organisations have considerable in-house experience in making MySQL scale, setting up replication, tweaking performance etc.
- Needs powerful hardware: CPU, RAM, fast disk.
- Backup, replication, recovery are standard processes.

HTTPDatastore + SQLite:

- Our testing shows better performance, but MySQL is being used at real scale too (see "What hardware do I need to run GRR?" below).
- You'll need to setup your own backup and recovery processses. This should be fairly simple since it's just a collection of sqlite files in a directory.
- Slave DBs don't need particularly powerful hardware, master should still be reasonably powerful.

2.9.9 I heard GRR was designed for Bigtable and now Google has a Cloud Bigtable service. Can I use it?

Internally we use Bigtable, but the internal API is very different so the code cannot be used directly. The Cloud Bigtable service uses an extension of the HBase API. We'd like to write a GRR database connector that can use this service, but (as at Aug 2015) the Bigtable service is still in Beta and the python libraries to interact with it are still being developed, so it isn't currently a high priority.

2.9.10 What operating system versions does the client support?

We try to support a wide range of operating systems because we know it's often the old forgotten machines that get owned and need GRR the most. Having said that 'support' is minimal for very old operating systems, we're a small team and we don't have the time to test the client on every possible combination. So the approach is basically to try and keep support for old systems by building compiled dependencies in a way that should work on old systems.

Windows

- Well tested on: 64bit Windows 7+ workstations, 64bit Win2k8+ servers.
- Should probably work: 32bit versions of the above, Windows Vista+ workstations.

OS X

- Well tested on: 64bit 10.8 (Mountain Lion)+ Note that 10.11 (El cap)+ systems require a 3.0.7.1+ client due to the OS X System Integrity Protection changes. And sleuthkit hasn't caught up to filesystem changes from Yosemite onwards. The compiler we use to build the Mac client template has become more aggressive with optimizations and now emits code optimized to specific processor versions. The mac client will not run on a processor older than our build machine - an early 2011 Macbook with the Intel Sandy Bridge architecture. If you need to run GRR on, for example, Core 2 duo machines, you will need a custom client built on one of those machines.

Linux

- Well tested on: 64bit Ubuntu Lucid+, CentOS 5.11+
- Should probably work: 32bit versions of the above, and essentially any system of similar vintage that can install a deb or rpm.

2.9.11 What operating system versions does the server support?

As of 3.1.0.2 we only support 64bit Ubuntu Xenial, since we had to move to systemd and didn't want the complexity of continuing support for upstart and init.d. Older versions of Ubuntu are easy to install on, but you will at least need a newer version of the protobuf library than what comes with the OS, and you will need to copy upstart scripts from the previous GRR release and modify them to work with the new release.

The server should probably work on most versions of linux that support systemd. Some users are running production installs on RHEL and CentOS, it just takes more effort to set up.

We don't provide a 32-bit server version since standing up new 32-bit linux servers is not something rational people do, and there are many places you can get 64-bit virtual servers for ~free. We use the "amd64" notation, but this just means 64-bit, it runs fine on Intel.

2.9.12 What hardware do I need to run GRR?

This is actually a pretty tough question to answer. It depends on the database you choose, the number of clients you have, and how intensively you hunt. Someone who wants to do big collection hunts (such as sucking all the executables out of System32) will need more grunt and storage than someone who mainly wants to check for specific IOCs and investigate single machines.

But to give you some data points we asked some of the GRR users with large production installs about the hardware they are using (as at October 2015) and provide it here below:

32k clients:

- Workers: AWS m4.large running 3 worker processes
- HTTP frontends: 6-10 x AWS m4.large. Sits behind an AWS Elastic Load Balancer.
- Datastore (SQLite): 5 x AWS m4.2xlarge. m4.2xlarge is used when running intensive enterprise hunts. During normal usage, m4.large is fine.
- AdminUI: 1 m3.large

15k clients:

- Workers and HTTP frontends: 10 x 4 core 8GB RAM virtual servers running 1 worker + 1 frontend each
- Datastore (MySQLAdvanced): 16 core 256G ram 8x10k drives. 128G RAM was sufficient, but we had the opportunity to stuff extra RAM in so we did.
- AdminUI: 12 core 24G RAM is where we left the UI since it was spare hardware and we had a lot of active users and the extra RAM was nice for faster downloads of large datasets. It was definitely overkill and the backup was on a 4 core 8GB of RAM VM and worked fine during maintenance stuff.

7k clients:

Run in AWS with c3.large instances in two autoscaling groups.

- Workers: 4 worker processes per server. The weekly interrogate flow tends to scale up the servers to about 10 systems, or 40 workers, and then back down in a couple of hours.
- HTTP frontends and AdminUI: Each server has apache running a reverse proxy for the GRR AdminUI. At idle it uses just a t2.small, but whenever there is any tasking it scales up to 1-3 c3.large instances. Sits behind an AWS Elastic Load Balancer.
- Datastore (MySQLAdvanced): AWS r3.4xlarge RDS server. RDS instance is optimized for 2000 IOPS and we've provisioned 3000.

2.9.13 How do I handle multi-organisational deployments?

Bake labels into clients at build time, and use a "Clients With Label" hunt rule to hunt specific groups of clients separately.

2.9.14 Which cloud should I deploy in? GCE? EC2? Azure?

Google Compute Engine (GCE) of course :) We're working on making cloud deployment easier by dockerizing and building a click-to-deploy for GCE. Our focus will be primarily on making this work on GCE, but moving to a docker deployment with orchestration will simplify deployment on all clouds. The largest cloud deployments of GRR are currently on EC2, and we hope the community will be able to share configuration and HOWTOs for this and other cloud deployments.

2.9.15 Where/how do you do your data analysis?

We mostly do this outside of GRR using an internal system very similar to BigQuery, and this powerful capability is now available to opensource users. GRR data is formatted for BigQuery using a hunt output plugin. There's a cronjob that outputs new results every 5 minutes, so there is very little delay between the server seeing a result and having it available for analysis externally. As at March 2016 an opensource user is working on an ElasticSearch output plugin.

2.9.16 When will feature X be ready?

Generally our roadmap on the main project page matches what we are working on, but we reserve the right to miss those goals, work on something entirely different, or sit around a fire singing kumbaya. Of course, given this is open source, you can add the feature yourself if it matters.

2.9.17 Who is working on GRR?

GRR has around 5 full-time software engineers working on it as their day job, plus additional part time code contributors. The project has long term commitment.

2.9.18 Why aren't you developing directly on open source?

Given we previously had limited code contribution from outside, it was hard to justify the extra effort of jumping out of our internal code review and submission processes. That has now changed, we are syncing far more regularly (often multiple times per week), and we are working on code structure changes that will make it easier for us to develop externally.

2.9.19 Why is GRR so complicated?

GRR **is** complicated. We are talking about a distributed, asynchronous, cross platform, large scale system with a lot of moving parts. Building that is a hard and complicated engineering problem. This is not your average pet python project.

Having said that, the most common action of just collecting something from machines and parsing what you get back has been made significantly easier with the artifacts system. This allows you to specify complex multi-operating system collection tasks with just a few lines of YAML, and collect any of the hundreds of pre-defined forensic artifacts with the click of a button.

2.9.20 What are the commercial competitors to GRR?

Some people have compared GRR functionality to Mandiant's MIR, Encase Enterprise, or F-Response. There is some crossover in functionality with those products, but we don't consider GRR to be a direct competitor. GRR is unlikely to ever be the product for everyone, as most organizations need consultants, support and the whole package that goes with that.

In many ways we have a way to go to match the capabilities and ease of use of some of the commercial products, but we hope we can learn something off each other, we can all get better, and together we can all genuinely improve the security of the ecosystem we all exist in. We're happy to see others use GRR in their commercial consulting practices.

2.9.21 Where is the logout button?

There isn't one. We ship with basic auth which doesn't really handle logout, you need to close the browser. This is OK for testing, but for production we expect you to sit a reverse proxy in front of the UI that handles auth, or write a webauth module for GRR. See the Authentication to the AdminUI section for more details.

2.9.22 How do I change the timezone from UTC?

You can't. Technically it's possible, but it's a lot of work and we don't see much benefit. You should run your GRR server in the UTC timezone.

In the datastore GRR stores all timestamps as microseconds from epoch (UTC). To implement timezone customization we could store a user's local preference for timezone and daylight savings based on a location, and convert all the timestamps in the UI, but there is a lot of complexity involved. We'd need to make sure each user gets the correct time and timezone displayed in all parts of the UI, handle the shifting winds of daylight savings correctly, provide clear input options on forms that accept timestamps, and make sure all the conversions were correct. All of that adds up to lots of potential to display or store an incorrect time, which is not what you want from a forensics tool.

It's common to have GRR users split across different timezones and a client fleet in many timezones. If we went to all of the effort above, all we have really achieved is fragmentation of each user's view of the system. Each user is still going to have to analyse and reason about events on clients in a different timezone to their own. But now we have made collaborative analysis harder: instead of being able to simply copy a timestamp out of the UI into a ticket or handoff timeline analysis notes to another shift in another timezone they will have to convert timestamps back into UTC manually for consistency.

For forensic analysis and timeline building, UTC always makes the most sense. We have added a UTC clock to the top right hand corner of the UI to help reason about the timestamps you see.

2.10 Publications

2.10.1 Papers

A scalable file based data store for forensic analysis, by Flavio Cruz, Andreas Moser, Michael Cohen. DFRWS 2015 Europe.

Hunting in the enterprise: Forensic triage and incident response, by Moser, Andreas, and Michael I. Cohen. Digital Investigation, 2013.

Distributed forensics and incident response in the enterprise, by M.I. Cohen, D. Bilby, G. Caronni. Digital Investigation, 2011.

2.10.2 Presentations

GRR Meetup: GRR Users Meetup: Fleetspeak, API client, Go rewrite @ Mar 2017 GRR Meetup by A. Moser, M. Bushkov, B. Galehouse, M. Lakomy. Video.

GRR Meetup: 3.1.0 Release @ Apr 2016 GRR Meetup by G. Castle and M. Cohen. Video.

GRR Meetup: API Edition @ Nov 2015 GRR Meetup by G. Castle and M. Bushkov. Covers API basics and discusses where we are headed with API dev.

GRR Hunting Best Practices @ Oct 2015 GRR Meetup by G. Castle.

Tactical use of GRR Artifacts, Fuse, Plaso and Timesketch by D. Bilby Demo Video (youtube).

"Intro to GRR" for the Open Source Network Security Monitoring Group @ UIUC Feb 2015. Video by G. Castle

"GRR: Find all the badness, collect all the things" @ Blackhat 2014 Video by G. Castle. Also presented a 20 min version of the same talk for Hacker Hotshots.

OSDFC 2012 GRR Overview, by D. Bilby

2.10.3 Workshops

ICDF2C Prague 2017 Workshop slides by M. Bushkov and B. Galehouse

DFRWS US 2015 Workshop slides and scripts to setup your own workshop infrastructure by G. Castle.

DFRWS EU 2015 Workshop workshop slides by A. Moser

OSDF 2013 workshop presentation by D. Bilby.

2.10.4 Podcasts

GRR Rapid Response on Down the Security Rabbithole Jun 2014 by G. Castle.

2.10.5 Whitepapers

GRR Forensic Artifacts White Paper, by G. Castle

2.10.6 External Presentations

These are presentations about GRR by people outside the core GRR team.

"Human Hunting" @BSidesSF 2015 by Sean Gillespie. Covers how Yahoo is using GRR. Video.

"GRR Rapid Response: Remote Live Forensics for Incident Response" @ Linux Security Summit Aug 2015 by Sean Gillespie.