MalGen *v.* 0.9 Overview

Authors
Collin Bennett, Robert Grossman, David Locke, Jonathan Seidman and Steve Vejcik

Open Data Group
400 Lathrop Avenue, Suite 90
River Forest, IL 60305
Phone: 708-488-8660
Fax: 708-434-0445
www.opendatagroup.com

23 April 2009

Abstract

This document provides an introduction to **`MalGen`** version `0.9`, an open source software package for generating site-entity log files. **`MalGen`** is intended to generate large, distributed data sets suitable for testing and benchmarking software designed to perform parallel processing on large data sets – *e.g.* Hadoop. The synthetic data generated follows certain statistical distributions which we believe presents a usable model for such logs. This document covers the format of the **`MalGen`** data, along with an overview of how the data is generated.

The generated data files consist of `100` byte records, one per line. The length includes the field separators and each field has a fixed-width.

An example of a generated record is:

```
00000000000-04817609343335299481|2008-10-06 03:28:11.812911|04435442449543695831|1|0000000000003498648
```


Introduction

`MalGen` is an open source software package for generating site-entity log files. Site-entity logs consist of events that record a visit by an entity to a particular site at a particular date and time. The assumption is that when visiting a site the entity may become compromised, possibly at a later time. If the entity is compromised, a flag indicating this is set to `1`; otherwise, the flag is `0`. Given a large collection of site-entity logs, an interesting problem is to identify those sites that are the sources of compromises. An associated document describes two benchmarks called **`MalStone A`** and **`MalStone B`** that represent stylized computations for detecting sites that are sources of compromises.

`MalGen` consists of Python and shell scripts that provide the facilities to generate a large data set distributed across multiple nodes in a cluster. **`MalGen`** uses a power law distribution to model the number of events associated with a site. Most sites have few events associated with them, while a few sites have a large number. This is the case, for example, with web sites: most sites have only a few visitors, a few sites have a lot of visitors, and a power law distribution is often used to model this distribution.

Data Format

The data generated by **`MalGen`** has the following pipe-delimitated format:

```
Event ID|Timestamp|Site ID|Flag|Entity ID
```

The following is a description of each field:

- **Event ID** – A sequential identifier for each record. To ensure uniqueness when generating a data set across multiple nodes the script appends a hash of the hostname where the data are generated. This is not a numeric field.
- **Timestamp** – The date and time of the event. This is a uniformly distributed random value over a configurable number days. The default is `365`.
- **Site ID** – Identifier of the site associated with the event.
- **Flag** – Field indicating whether the entity associated with the event is known to be compromised.
- **Entity ID** – Identifier of an entity associated with the event.

With release `0.9`, the overall length (`100` bytes) stays the same as with release `0.8`, but each field is a now a fixed-width. The number of bytes for each field is:

Field	Event ID	Timestamp	Site ID	Flag	Entity ID
Bytes	31	26	19	1	19

Table 1. Field Widths

The delimiter is a single `ASCII` character which requires one byte, giving us a total of `100 ASCII` characters / bytes. See the file `FIXED_WIDTH.txt` in the `docs` directory in the **`MalGen`** package for more details concerning the field widths.

Examples of the generated records are:

```
0079999999600438500244954329305|2009-04-02 10:50:41.606070|0445544244954369755|0|0000000000001032397
00000000000-0481760934333529948|2008-10-06 03:28:11.812911|0443544244954369583|1|0000000000003498648
```

The Data

Site-entity log files are generated by **`MalGen`** in several steps.

First, a set of entities is generated with the assumption that the number of events associated with any given entity is distributed according to a normal distribution. The current code uses a default mean of `27` events per entity per day, with a default standard deviation which is `10`% of the mean.

Second, compromised sites and events associated with them are generated. For each compromised site, a random compromise date is generated. The time for the first such site is constrained to lie within a very short offset of the time period of interest. Subsequent sites are constrained to lie after, but close to, the time beginning with the date of the previous compromise.

For each site, the number of events is randomly generated using a power law distribution and a set of entity IDs is randomly generated from the pool of available entity IDs. The power law distribution is constructed so that most sites are associated with a relatively few number of entity IDs (a few hundred a day), but with a long tail so that there are a small number of sites with a very large number of events. The Entity IDs are sampled until the number of events for each site is complete.

For the compromised sites, a visit by an entity subjects the entity to a probability (*e.g.* `70`%) of being compromised. If an entity is compromised, it is tagged as such after a delay period (*e.g.* one week). If an entity is already compromised when it visits a compromised site again, it is subject to compromise if the date of the current event precedes that of the event that compromised it. In this case, the date-time of compromise is updated accordingly.

After all event histories for the requested number of compromised sites are complete, subsequent sites are assumed to be uncompromised with no possibility of additional compromise. This is the third step and uses the same process as what was described to construct visits for compromised sites.

The Event ID is the unique key. The field is `31` bytes and has two components. The Event ID has a sequence which makes the value unique on the node. The field also contains a hash of the `hostname`, which makes it unique across nodes. Using `31` bytes means that we can generate a maximum of `100` billion records per node and with `n` nodes, a total of `n * 100` billion `100` byte records can be generated. There a few cases which are detailed below which could decrease the number of unique keys which can be generated.

Two additional comments about data generation:

`MalGen` is designed to generate large data sets that span all the nodes in a cluster. To create consistent data in parallel over all the nodes in the cluster, **`MalGen`**: *i*) generates the data describing all compromised sites on one machine in the cluster; *ii*) scatters the required information to all the other nodes in the cluster; *iii*) generates the data for all the uncompromised sites on all the other nodes in the cluster.

By keeping information about sites in memory (*vs.* on disk), **`MalGen`** can improve its performance. With the default statistical parameters, **`MalGen`** generates the compromised data file with `500` million `100` byte records for approximately `120000` sites in about `190` minutes using a Dell 1435 2.0GHz dual-core AMD Opteron 2212 processor and `16` GB of memory. Generating a `100` million record compromised file took approximately `54` minutes on the same machine.

Using MalGen

This section is a formatted version of the `README` file in the `docs` directory of the source package.

Introduction

Generating the data is a two step process. First, the initial run has to take place on a single node. This generates a data file of compromised events and it's associated metadata file. In addition, state information is saved from the Python session as `INITIAL.txt`. This file is necessary for step 2.

The `INITIAL.txt` file must be made available to each node that is going to generate the uncompromised synthetic data. We `scp` the file to each node where this is going to take place and run the data generation in parallel across the nodes.

The second step, for historical reasons, generates multiple data files and their associated metadata and then concatenates them into a single file. The original pieces are then deleted. Originally, this was so that the code could run on older hardware with less RAM. Later versions of the code improved memory handling and this is no longer necessary, but has been left for backward compatibility and for people running **`MalGen`** on older hardware.

Scripts have been created to assist in generating the data. The default values in `env.sh` assume that the release archive was extracted in your home directory. If this is not the case, update the `MY_HOME` value in `env.sh`. For example, if I downloaded the `tar.gz` file from Google Code into `/opt/malgen`, then I would edit `env.sh` to have

```
MY_HOME=/opt/malgen
```

Release archives are named `malgen-v<VER>-googlecode-<YYYMMDD>-r<REV>.tar.gz`. The **`MalGen`** release archive contains:

```
$ tar zxvf malgen-v0.9-googlecode-20090429-r495.tar.gz
$ tree

.
|-- LICENSE
|-- MalGen_v0.9_Overview.pdf
|-- bin
|   |-- cloud
|   |   |-- env.sh
|   |   |-- execs
|   |   |   `-- exec_sector_upload.sh
|   |   |-- fork
|   |   |   |-- fork_hdfs_load.sh
|   |   |   |-- fork_malgen.sh
|   |   |   |-- fork_sector_upload.sh
|   |   |   `-- pushout_initial.sh
|   |   |-- malgen
|   |   |   |-- initial_generator.sh
|   |   |   |-- malgen.py
|   |   |   `-- malgen.sh
|   |   |-- nodes.txt
|   |   `-- test.sh
|   `-- docs
|       |-- FIXED_WIDTH.txt
|       |-- README
|       |-- RELEASE.txt
|       `-- STRICT_OPTION.txt
|
`-- malgen-v0.9-googlecode-20090429-r495.tar.gz

6 directories, 18 files
```

These files need to be on every machine where data will be generated. Also, make sure that they are executable and that you have the necessary permissions. The directory structure must be the same as well. A description of what the scripts do is provided in case you want to modify them for your system or to skip them all together and only use the python code.

You should have `passphraseless ssh` set up to use these scripts. To do this, log into the machine where the scripts will be run from, and type:

```
$ ssh-keygen -t dsa
```

Then add the generated key to the `.ssh/authorized_keys` file in your `${HOME}` directory on each node where data will be generated.

`env.sh` is where the common parameters from the scripts are defined. This will most likely have to be edited to fit your system. You can run `./test.sh` to see what the parameters are set to. This does not make any changes to your system.

Running the Scripts

Step One, Initial Generation

The first step has been scripted as `initial_generator.sh`. Running it creates a data file containing compromised events, its metadata and the `INITIAL.txt` file.

`INITIAL.txt` then has to be made available to each node. If you are only using a single machine, the next step can be omitted. If you are using multiple machines, the easiest way to proceed is to edit `nodes.txt`. This file should contain the IPs of the nodes where data generation will take place. There should not be any comments in the file and each IP should be on its own line. For example:

```
$ cat nodes.txt
```

```
192.168.15.2
192.168.15.3
192.168.15.4
192.168.15.5
192.168.15.6
```

You can now use `pushout_initial.sh` to `scp` the file `INITIAL.txt.txt` to each node in the list. `pushout_initial.sh` takes the location of `INITIAL.txt` as a parameter. So if my `INITIAL.txt` file in `/tmp` and I want to `scp` it to each of my six nodes, I would run:

```
$ ./pushout_initial.sh /tmp/INITIAL.txt
```

and the remote destination, `${IMPORT_FILE_DIR}`, is declared in the `env.sh` file.

Step Two, Data Generation

Once this has completed, you can generate data on each node in parallel. As mentioned above, temporary files will be created and then consolidated. This has also been scripted. The script to run is `malgen.sh`. You can log into each node and run this by hand or you can call `fork_malgen.sh` from any machine. `fork_malgen.sh` is a simple script that reads the `nodes.txt` file. It uses `ssh` to connect to each machine and then invoke the local `malgen.sh`. This is convenient if there are a lot of nodes in your list.

When `malgen.sh` has completed on each node, the data has been generated and you are ready to load it into your favorite application.

Other scripts

Hadoop and Sector are common applications for processing large data sets of this type. We have also included fork scripts to load the data into the HDFS and Sector File System.

- `fork_hdfs_load.sh` reads the list of machines in `nodes.txt` and logs into each one, calling the Hadoop upload command to load the data file into the HDFS.
- `fork_sector_upload.sh` does the same thing, but on each node, it calls `exec_sector_upload.sh`. This script is also provided and it performs a Sector upload.

The Python Code

The data generation in both steps one and two uses `malgen.py`. This code has several configurable parameters. Information is available from the script itself:

```
$ python malgen.py -help

Usage:
            Seed Run-
            malgen.py [options] 0 nrecs nrecsperblock blocks
            All other runs-
            malgen.py [options] seed_index

              Use nonzero seed_index for non-seed runs.
            For seed runs, the arguments correspond to:
            number of events seed run,
            number of events per non-seed run, and
            total number of non-seed runs, respectively.

Options:
  -p PCOMP, --pComp=PCOMP
                        Compromise Probability (default .70)
  -P POWER, --power=POWER
                        Power for events per site distribution (default -3.5)
  -d DELAY, --delay=DELAY
                        Delay in days in tagging compromise (default 1.0)
  -D NDAYS, --ndays=NDAYS
```

```
                          Number of days for data sample (default 365)
  -m EVENTS_PER_ENTITY, --events_per_entity=EVENTS_PER_ENTITY
                          Mean number of events per day per entity (default 27)
  -s STD_EVENTS_PER_ENTITY, --std_events_per_entity=STD_EVENTS_PER_ENTITY
                          Std Deviation in number of events per day per entity
                          as a fraction of the mean number of events per day
                          (default .1)
  -O OUTDIR, --outdir=OUTDIR
                          Directory for output and for seed initialization data
                          (default /tmp/)
  -o OUTFILE, --outfile=OUTFILE
                          Output filename (default events-malstone.dat)
  -g BACKGROUND, --background=BACKGROUND
                          Number of background sites that contribute to external
                          compromises (default 0)
  -l LOCAL, --local=LOCAL
                          Number of sites acting as source of compromise
                          (default 1000)
  -S SITESCALE, --sitescale=SITESCALE
                          Scale factor determining typical number of events per
                          site (default 10000)
  -t, --truncate          Truncate the generated records so that exactly the
                          specified number of records is generated rather than
                          following the statistical distribution (default False)
  --version               show program's version number and exit
  -h, --help              show this help message and exit
```

When Step One is executed, the syntax for generating the initial seed data is:

```
$ python malgen.py [options] \
    0 <num_records> <records_per_nonseed_block> <num_nonseed_blocks>
```

In this case the first argument of zero indicates that you are generating data for compromised sites, which is what we call the seed block.

- `num_records` is the number of records that you want generated in the compromised data file.
- `records_per_nonseed_block` is the number of records that each non-seed block will generate (step 2).
- `nonseed_blocks` is the number of times that you will run step two across all nodes. The default number of blocks is 5 in `env.sh` and in the example `nodes.txt` file above, there are `5` nodes, so we would use `25` for `num_nonseed_blocks`.

Valid options for this step are:

```
-p PCOMP, --pComp=PCOMP
-P POWER, --power=POWER
-d DELAY, --delay=DELAY
-D NDAYS, --ndays=NDAYS
-m EVENTS_PER_ENTITY, --events_per_entity=EVENTS_PER_ENTITY
-s STD_EVENTS_PER_ENTITY, --std_events_per_entity=STD_EVENTS_PER_ENTITY
-O OUTDIR, --outdir=OUTDIR
-o OUTFILE, --outfile=OUTFILE
-g BACKGROUND, --background=BACKGROUND
-l LOCAL, --local=LOCAL
-S SITESCALE, --sitescale=SITESCALE
-t, --truncate
```

To run step two, execute:

```
$ python malgen.py [options] <start_index>
```

`start_index` must be non-zero and should be greater than the number of records already generated.

Valid options for this step are:

```
-O OUTDIR, --outdir=OUTDIR
```

```
-o OUTFILE, --outfile=OUTFILE
-t, --truncate
```

Examples

MalGen records represent entires in site-entity logs where some number of the entities have been compromised. The data generated follows certain statistical distributions which we believe presents a usable model for such logs.

There are two intended uses for **MalGen**. The first is to generate a large, possibly distributed, data set for use with analytics. The second is generate data for use with benchmarking algorithms or applications.

With the first use, records are generated probabilistically and extra records may be produced so that the entire data set follows the specified distribution. With the second use, strict adherence to the distribution is not necessary as the user is more interested in generating exactly the specified number of records.

Before the `v0.9` release, **MalGen** always attempted to follow the statistical distribution. This means that the exact number of records specified was usually not generated. The overage, when dealing with `100`'s of millions of records per node, tended to be in the `1/100`ths of a percent and was not a significant issue when the records were intended solely as data to benchmark algorithms or applications against.

Release `v0.9` exposes a switch which can be used at the command line to toggle between following the distribution and generating exactly the number of records specified. When the distribution is followed, the number of records generated is probabilistic, so there is no way to accurately determine how many records will be included in each generated file. When the exact number of records is generated, the data may be slightly inappropriate for statistical analysis.

The flag is `-t` (`--truncate`). If it is specified in the call to **MalGen**, then the last batch of records generated in each run will be truncated so that the numbers of records in the produced file is exactly the number specified. The default value, when the flag is not used, is to follow the distribution.

Example 1:

We will walk through what calling `./initial_generator.sh, pushout_initial.sh` and `malgen.sh` does.

In this example we will have a single machine generate the initial seed file and the associated compromised data. We will generate a `500` million record file using the default parameters.

First run the following command on the machine:

```
$ python malgen.py -O /raid/testdata/ 0 500000000 100000000 25
```

The parameter given by `-O` is the directory where the files will be created. The first argument of zero indicates that this is step one and so we will generate data for compromised sites. The second argument (`500000000`) is the number of records that we want to produce with this step. The third argument (`100000000`) is the number of records that we want to produce each time we run step two. The fourth argument (`25`) is the number of times that we plan on running step two across all nodes: five blocks each on five nodes. This is equivalent to running `initial_generator.sh`. with the default values in `env.sh` and `nodes.txt`.

If want to generate exactly `500000000` records, then include `-t` in the above line:

```
$ python malgen.py -O /raid/testdata/ -t 0 500000000 100000000 25
```

Second, copy `INITIAL.txt` to each node where data will be generated.

```
$ for NODE in `cat nodes.txt`; do
    scp /raid/testdata/INITIAL.txt ${NODE}:raid/testdata/ ;
  done;
```

This is the equivalent of running `pushout_initial.sh /raid/testdata/INITIAL.txt`.

Third, log into each of those nodes and run

```
$ python malgen.py 500000000
```

```
$ python malgen.py 600000000
$ python malgen.py 700000000
$ python malgen.py 800000000
$ python malgen.py 900000000
```

This is the equivalent of running `malgen.sh` on each node. Alternatively, you could have run `fork_malgen.sh` from a single machine which would have called `malgen.sh` on each node list in `nodes.txt`. For each run, the argument is the record number of the first record number that has not yet been generated. Following the behavior of the script, we use `500000000` as the start index for the first run.

You have to use start indexes that are far enough apart so that identifiers in one run do not overlap with those from another. For example, with our values, `python malgen.py 500000000` will generate `100000000` records. If the next run used `580000000` as the start index, then there would be an overlap of `20000000` records and they would have non-unique keys (the Event ID field).

We start at `500000000` instead of `1`, to cover the case when the initial compromised data generation takes place on the same node. If this is not the case, then you can safely start at any positive number. It was our goal to be able to use the same scripts in both the single and multiple machine scenarios, so we always use a start index greater than the last one generated by the compromised data generation step.

It is fine to have the same start indexes across multiple nodes, as the hash of the host name is used to ensure uniqueness. This fails if nodes have identical hostnames, *e.g.* localhost, or in the unlikely event of collisions.

In between each run, you must rename or move the metadata and data files or they will be overwritten. You could have used the `-o` parameter to give each file a unique name:

```
$ python malgen.py 500000000 -o data.part1
$ python malgen.py 600000000 -o data.part2
$ python malgen.py 700000000 -o data.part3
$ python malgen.py 800000000 -o data.part4
$ python malgen.py 900000000 -o data.part5
```

and then you would not have to rename after each run and in fact you could run them in parallel on each node. We do execute in parallel across nodes but not within a node. The data generation is too memory intensive and we have experienced degraded performance when trying to generate multiple files simultaneously.

Again, if want to generate exactly the specified number of records, use `-t` with the above commands.

Example 2:

If I am happy with all the default values in `env.sh`, and I have updated `nodes.txt` with the IPs of my nodes, then to generate one `500` million `(NUM_BLOCKS * BLOCK_SIZE)` record file on each node in `/raid/testdata (IMPORT_FILE_DIR)` with a compromised data file of `500000000` (`INITIAL_BLOCK_SIZE`) records, all I need to do is:

Log onto the machine where the compromised file will be generated and go to the directory where the **`MalGen`** release was extracted.

```
$ cd bin/cloud/malgen
$ ./initial_generator.sh
$ cd ../fork
$ ./pushout_initial.sh /raid/testdata/INITIAL.txt
$ ./fork_malgen.sh
```

As of writing this document, `malgen.py` has been tested against Python 2.4.4 and 2.5.2 on Debian and Ubuntu Linux distributions. It has also been tested on OpenSolaris 2008.11 with Python 2.5.2 with one known issue. The `malgen.sh` script uses `hostname -i` to get the IP of node generating the data. This is used in the naming of the output files. On OpenSolaris, `hostname` does not accept `-i`. The the file names become something like `events-malstone—-seed.dat`. You may want to use something like:

```
/sbin/ifconfig -a | awk '/inet/ {print $2}' | egrep -v "127|::"
```

instead. The `hostname` command is also called from within `malgen.py`, but it does not use `-i`.

Release Notes

This section is a formatted version of the `RELEASE.txt` file in the `docs` directory of the source package.

version 0.9
release date 2009-04-29

Features
1. Fixed-width fields. See the `FIXED_WIDTH.txt` file in the `docs` directory of the source package for more information.
2. Added the `-t, --truncate` option which at the expense of strictly following the statistical distribution, generates exactly the number of specified records. See the `STRICT_OPTION.txt` file in the `docs` directory of the source package for more information.

Bugs
1. Fixed incorrect version number when `--version` is called.

Other
1. Released under the Open Cloud Consortium.

version 0.8
release date 2009-04-20

Features
1. Removed the historical Aux field. This reduced the number of fields to five from six.
2. Records are now a fixed length. Each recorded generated is `100` bytes unless values are used for the distributions which make this impossible.

Bugs

Other

version 0.7
release date 2009-02-20

Initial Release to Google Code

MalGen Roadmap

The following are some enhancements and changes for future releases of **`MalGen`**:

Short Term

Logging / Metadata
1. Improve metadata provided.
2. Enhance logging options.

Performance
1. Determine parameter boundaries limiting performance.
2. Document performance.

Medium Term

Data
1. Provide pre-run calculation of relevant derived parameters.
2. Allow alternate distributions.

Interface
1. It will no longer be required to use `0` as the start index (first parameter) in calls to `malgen.py` when creating the initial seed. This will be controlled by a flag that can be passed in from the command line with the other configurable options.

Long Term

Data

1. Allow user-tailoring of output file structure.

`MalGen` is available on google code. The project page is http://code.google.com/p/malgen.