Aeris 2.1 User Guide

DESCRIPTION

Aeris is an automated implant written in C that supports a number of POSIX-based systems.

PLATFORM SUPPORT

- Debian Linux 7 (i386)
- Debian Linux 7 (amd64)
- Debian Linux 7 (ARM)
- Red Hat Enterprise Linux 6 (i386)
- Red Hat Enterprise Linux 6 (amd64)
- Solaris 11 (i386)
- Solaris 11 (SPARC)
- FreeBSD 8 (i386)
- FreeBSD 8 (amd64)
- CentOS 5.3 (i386)
- CentOS 5.7 (i386)

FEATURES

- Configurable beacon interval and jitter
- Standalone and Collide-based HTTPS LP support
- SMTP protocol support
- TLS Encrypted communications with mutual authentication (Appendices C and D)
- Compatibility with the NOD Cryptographic Specification (Appendices C and D)
- Structured command and control that is similar to that used by several Windows implant- (section IV)
- Automated file exfiltration (section IV)
- Simple and flexible deployment and installation (section III).

DISTRIBUTION

The Aeris distribution consists of a set of Python utilities together with a set of binaries, with one binary per platform listed in Section I. These binaries (which we call unpatched binaries) are fully functional but are not deployable because they do not contain configuration information. Instead, they contain placeholders (GUIDs and static buffers) that will be overwritten with the appropriate information at build time. The Aeris builder generates a valid configuration based on user input and uses that configuration to create a deployable Aeris instance.

Aeris includes the following files:

- aeris/ Python/script libraries
- bin/ Unpatched binaries
- cgi/agnt.c HTTPS CGI LP source code
- cgi/agnt.cgi Statically compiled ELF32 HTTPS CGI program
- docs/ Documentation
- builder.py Script - builds a new instance

- mklp.py Script - creates new LP keys
- patcher.py Script - builds a replica of a deployed instance
- postproc.py Script - processes exfiltrated data
- task_builder.py Script - task generator

All scripts and builders should remain on the high side. The builder will output UNCLASSIFIED binaries, keys, and configurations, as explained in the next section.

Note that we provide the source code for our CGI program, because we want to provide operators with the ability to modify and control the LP infrastructure as they see fit.

BUILD INSTRUCTIONS

Use builder.py, see builder.py -h for details. Python 2.7 is required.

The builder outputs the following directory structure:

- DEPLOY_{target ID}_{timestamp}/: top level directory
    - implant/: target files
        - aeris: implant binary (UNCLASSIFIED)
        - config: encrypted implant config (UNCLASSIFIED)
    - keys/: implant and CA keys
        - ca.cert: CA certificate (UNCLASSIFIED)
        - ca.priv: CA private key (SECRET//NOFORN)
        - tgt.cert: implant certificate (UNCLASSIFIED)
        - tgt.priv: implant private key (UNCLASSIFIED)
    - lp/: LP files
        - {lp 1}/: files for lp 1
            - {lp 1}-inst: installer for lp 1 (UNCLASSIFIED)
            - {lp 1}.cert: certificate for lp 1 (UNCLASSIFIED)
            - {lp 1}.priv: private key for lp 1 (UNCLASSIFIED)
            - {lp 1}.conf: apache config file for lp 1 (UNCLASSIFIED)
        - {lp 2}/...
        - {lp 3}/...
    - receipt: human-readable receipt (SECRET//NOFORN)
    - receipt.xml: XML/machine-readable receipt (SECRET//NOFORN)

Each LP installer contains all keys and Apache configuration files necessary for an LP deployment. It is based on a Fedora 14 LP provided by COG/NOD operators late in 2011. Since this LP may differ in configuration from the operational LP at the time of an Aeris deployment, the LP builder script is provided primarily to provide the user with a step-by-step guide on how to configure an Aeris LP. We recommend that the user not run the script outright but rather adapt the script or issue the steps manually to account for any differences in LP configuration.

DEPLOYMENT INSTRUCTIONS

INSTALLATION

Aeris does not have a separate installer. To deploy it, simply place an Aeris binary in the desired directory. Rename the binary in any way that you wish. Note that the configuration is patched in at build time; hence, no additional files (beyond possibly those related to persistence -- see the next section) are needed.

PERSISTENCE

Since mechanisms for persistence vary greatly among POSIX-compliant systems, Aeris does not have such functionality built-in. The operator must supply persistence methods, and, if desired, the developers can incorporate them into the Aeris builder in a later phase.

Note that any persistence mechanism should invoke Aeris as described in Section IV.

IMPLANT IDENTIFIERS

When Aeris reports back to the LP, it will include its implant identifier along with the payload. This identifier will consist of two parts: A parent (static) part equal to the implant identifier specified at build time and a child (dynamic) part created from hardware addresses of up to four NICs on the target system. The format is as follows:

{parent}-{NIC 1}-{NIC 2}-{NIC 3}-{NIC 4}

The NICs will be sorted so that they appear deterministically (alphabetical order).

CONTROLLING AERIS

INVOKING AERIS

- ./aeris &
- ./aeris -e &
- ./aeris -f &

(Aside: Note that the use of the name aeris is notional here; the operator can name the binary anything that he wishes, per section III.A).

The '-e' and '-f' flags have to do with enforcing singleton instances. By default, Aeris will check for other instances of itself when it starts. If it finds such an instance, it will exit. Since this behavior is not always desirable, the '-e' and '-f' options provide the capability to bypass these checks as follows.

- '-e': If this option is specified, the new instance of Aeris will wait until the previous instance of Aeris has exited
- '-f': When this option is specified, the new instance of Aeris will ignore any other instances of Aeris. This is not recommended in general, because two simultaneous running instances may overwrite each other's on-disk configuration

There is one extenuating circumstance in which using the '-f' flag may be useful. If the implant exits uncleanly (e.g., via a kill -9), the semaphore used to synchronize multiple instances may persist. In this case, a new Aeris instance may believe that another one is running when that is not the case. Specifying '-f' will enable the newer process to circumvent this issue.

COMMANDING AERIS

The Tasker (task_builder.py) generates the command packages used to command an Aeris instance. It also provides the operator with a custom command line interface for creating command files. (The Collide Handler provides a very similar user interface for task generation. See Section VII.A for more information.)

The operator must provide the Tasker with the implant's receipt.xml file, so that the tasking will be encrypted appropriately. Thus, the Tasker may be invoked as follows:

```
# python task_builder.py receipt.xml
```

Once the tasker is running, follow the on-screen instructions.

Aeris allows operators to combine multiple commands into batches that are uploaded to the LP and eventually processed as a unit by an Aeris instance. Up to 65535 commands are allowed in a single batch. Batches are created using the âgenerate batchâsub-shell of the Tasker. In generate_batch mode, there are special commands to modify command order, remove commands, cancel generate_batch mode, and finally, once all tasks have been added, generate a bundled task (i.e., a command file). Once a task bundle is generated from the standalone Tasker, the operator should upload it to the listening post and copy it to the directory /var/www/{static implant ID}/update.pkg. (See section III.C for more information on implant identifiers.)

Tasks within a task bundle are executed in sequence. If a task fails, the remaining tasks are not run. The results of the tasks are returned in one result file.

To see how a specific command works or what its parameters are while running the Tasker, simply type 'help {command}', where {command} is the command in question. For example, typing:

```
# help exec_fg
```

will cause the Tasker to display help information on the execute foreground command.

One final note regarding tasking: The implant will not execute any residual, persistent tasks if no task is present on the LP. This means, for example, that if the operator has previously set a watch path or a drop box, file data will not be exfiltrated without a task present on the LP. For that reason, it is recommended that the operator configure an innocuous âdefaultâtask (such as setting the beacon interval but not changing its value) unless he wants the implant to remain silent.

DEPLOYING TASKS

To deploy the tasking payload, refer to section VII.

UPDATING AERIS

Aeris does not have an explicit update function as other implants might. To update Aeris, one needs to send a series of commands in a batch task: 1. Delete the old (currently-executing binary) using an exec_fg command (note that using exec_bg leads to a race condition between deleting the binary and the put command); 2. Put the new binary in the same location as the old binary (making sure to set the permissions so that the new binary is executable); 3. Issue a hard_reset command.

UNINSTALLING AERIS

Aeris will uninstall in any of the following three cases:

1. The operator issues an uninstall command;
2. The operator has configured a self-delete timeout. If the implant has not beaconed successfully within the specified number of seconds, it will self-terminate;
3. The operator has configured an uninstall date. Once the date is reached, the implant will uninstall.

The implant will take the following steps when uninstalling:

1. It will delete all pending task files (i.e., these are not uploaded);
2. It will delete its config file;
3. If the operator has configured a DNS domain, the implant will query that domain; 4. It will delete itself;
4. It will exit the process.

In each case, the implant will attempt to first overwrite the file with data from /dev/urandom prior to deleting the file. However, âsecureâdeletion is not a given; on Solaris, for example, overwriting the implant binary with pseudo-random data will result the implant's crashing due to an invalid instruction. In any case, the files are deleted.

The implant also allows the operator to specify additional files to purge when issuing an uninstall command. This feature can be used, for example, to clean up any external persistence mechanisms.

COMMUNICATIONS

All communications between implant and endpoint occurs via HTTPS. Each implant instance has a unique certificate authority (CA). The implant's certificate as well as each LP certificate is signed by the CA. When establishing a communications channel with an LP, the implant and the LP verify each other's certificates against the CA. If the certificates do not validate, the connection is terminated. In addition, if the server certificate is invalid, the implant will demote the LP (i.e., cycle to the next LP in the list and use it for the next beacon, provided that multiple LPs were configured). Other conditions causing a demotion include an inability to connect when the implant has Internet access and a failure to decrypt or validate a tasking payload.

Two sets of communications occur during the course of a beacon cycle. First, the implant will issue an HTTPS GET to download the tasking payload. Once this payload has been validated and decrypted, and once the contents of the package have been executed, the implant will issue an HTTP POST to upload the collected data. Important URLs on the LP are listed below.

- /update.pkg
  - This URL should allow the implant to access an encrypted tasking payload.
  - If no package exists and the client receives a 404, it will not take any action for the present beacon cycle. The LP will not be demoted in this case.
- /agnt.cgi
  - This is the URL to which the implant will upload collected data. See Section VII of this document for more information.

Once data files have been uploaded, the user must copy the files from the LP to a high- side system in order for decryption to occur.

CRYPTOGRAPHY

Aeris uses a cryptographic suite that is compatible with the NOD Cryptographic Specification (NSPEC-001). Specifically, the following algorithms are used:

- 2048-bit public/private RSA keys
- RSA key exchange
- AES-256 symmetric key encryption
- Cryptovariable generation using PolarSSL's built-in number generator
- Digital signatures that rely on SHA-512.

In particular, all data exfiltrated to the LP are encrypted and signed using these algorithms. The keys used to decrypt the data (in particular, the CA's private key) reside only on the high-side. Hence, any would-be eavesdropper cannot decrypt the data, even if he gains full control of the LP. Furthermore, each file is encrypted with a separate symmetric key.

See the Aeris-specific cryptographic documentation (Appendix) for more information.

POST PROCESSING

To decrypt collected data files, the user must move them from the LP to a high-side system. Once this is accomplished, he can use the Aeris post-processor (postproc.py) to decrypt.

Each file will adhere to the following naming convention by default: {timestamp}- {random six-character suffix}.pkg, where the timestamp represents the UNIX time at which the file was uploaded to the LP. We recommend that the operator not change file names during retrieval, since the file name itself helps to establish when the file was received and serves as a basis for file naming conventions within the decrypted data store. However, the post-processor does not depend on this particular naming convention.

Post-processor usage is described in detail via the builder's -h option and will not be rehashed here. Once the post-processor has processed each file, it will create (or add to) the following directory structure:

- DATASTORE_{target ID}/: top-level datastore for target {target ID}
  - exec/: data captures from foreground exec commands
  - files: collected files
  - results/: high-level status/result summary for each task
  - file_data.bin: dictionary of SHA-512 checksums for partial files (if known)

Within exec and results, files are named according to the input file name. The files directory mimics the file system on the target box. So that multiple versions of a file can be supported, each file has its time of last modification appended to its name. Note that since the implant is unaware of time zones, the timestamp will reflect the target's local time. Also, if the file is still in transit, the file name will also include a âpartâ extension. Once the entire file has been received and if its SHA-512 checksum matches that of the file on the target system, the .part extension will be removed. A hypothetical file repository might appear as follows:

- DATASTORE_5555/
  - files/
    - home/
      - user/
        - myfile.1346598270
      - bigfile.134607928.part
      - file1.1346070577
      - file1.1346086895

In each case, the relative path from the files directory reflects the path of the file on the target system; the timestamp reflects the time of last modification on the target system; and the .part extension reveals some state of incompleteness. Also note that we have two versions of file1, one that was last modified at 1346070577 (27Aug2012 at 12:29:37) and one at 1346086895 (27Aug2012 at 17:01:35).

A file may remain with a .part extension permanently if one of the following conditions holds:

1. If one or more sections of the file were lost or corrupted in transit;
2. If a target-side error was encountered when trying to exfiltrate (read or copy) part of the file;
3. If the file was modified during exfiltration (and therefore the checksum on target differs from that on the high-side). In such instances, recovering a current, valid copy of the file may or may not require further action on the part of the operator. If the file is part of a watch directory, the implant will detect any file changes and will exfiltrate it at a later time. If the file's retrieval was triggered via a GET command, the operator should reissue that command.

LISTENING POST OPTIONS

Aeris is compatible with either a standalone, CGI-based LP or with Collide. The differences are described below.

Note: Both the Collide lowside and the CGI standalone server require the implant's certificates for mutually authenticated SSL. The example script - inst.txt contains example instructions on how to put the certs in the correct place on the LP. We recommend that the operator not run this script directly, but use it only as a guide.

COLLIDE INTEGRATION

The Aeris Collide handlers are Python packages used to interface between Aeris and the Collide Automated Implant Command and Control system. Aeris provides handlers that define the user interface and facilitate Implant communication. Different sets of handlers are used for the Collide high-side and low-side to limit the exposure of code on the unclassified, internet-facing low-side.

For information on installing and running Collide, see the Collide User's Guide. This guide will only cover the use of the handlers developed for Aeris and special instructions that must be done to setup the keys on the Listening post.

High-side Handlers

The high-side Collide handlers are responsible for defining the user interface, providing crypto services, and supporting the post processing of Implant communications.

Payload

Aeris's Collide Payload defines the user interface required to task implants. The UI provided through Collide is similar to that provided in the Tasker. One distinction is that the Collide consumes tasks directly while the Tasker saves tasks to a file. Another is that the receipt file with keys is automatically loaded when a user selects a target The high-side payload consists of one file, the payload init file. The high-side payload requires the Aeris Python module, named 'aeris'. The module should be located within the Aeris payload on the Collide high-side.

Registering a new Target

To register a new target on Collide, in the Collide prompt type registertarget .

Post Processing Rule

Aeris includes one Collide rule intended to support the post processing of result files, called 'Aeris_meta_extraction_rule.py'. The rule simply sends encrypted copies of files received from any Aeris Implant to the input directory of the Post Processor.

The path of the directory may be specified in the body of the rule by modifying the value of _POST_PROCESSOR_INPUT_DIR; it defaults to '/tmp/Aeris_input/'.

Low-side Handlers

The low-side handlers are responsible for Aeris communications via the Collide listening post.

The low-side payload consists of the payload init file and handler for HTTPS . Unlike the high-side, the low-side payload does not require the Aeris Python module

Adding Certs to Low-Side

To configure apache on the low side with the right domains and certs for each domain, the operator needs to run the -inst.txt script on the LP for each domain. This file is found in the lp folder in the build directory for the implant. If there are multiple domains registered, all of them must be ran on the low side.

To run those scripts on the LP, the operator must first log in as root, which is explained in the Collide documentation. Then the operator must remount the partition as read write, using the following command:

```
# mount -n -o remount,rw /
```

Run the script for each domain and log out.

STAND-ALONE LP CONFIGURATION

The standalone CGI option allows the user to deploy a single CGI binary (agnt.cgi-32 or agnt.cgi-64) together with the LP-specific files on an Apache web server. In general, the Apache configuration mimics that used to support other OS X implants, such as BowmanHeir and Triton. The operator will need to configure Apache with support for SSL, mutual authentication, and SNI (server name indication). The LP configuration script contains all of the required, LP-specific information (keys and vHost files). See Section II regarding use of this script.

To use the CGI-enabled LP, the operator should place the appropriate CGI binary (32 or 64 bit) in the implant's document root. The CGI program must be named âagnt.cgiâ as this URL will be requested when an Aeris instance attempts to post data back to the LP. Note that the CGI script does not handle tasking, as these packages are generated by the Aeris tasker, and the implant fetches them directly with an HTTPS GET. Thus, it handles HTTPS POSTs only and acts as a pass-through, writing any posted data directly to a file on the LP. A summary of expected naming conventions (within the implant's document root) follows.

- agnt.cgi: CGI script that handles HTTPS POST requests. Note: it must be named agnt.cgi. The provided files, agnt.cgi-32 and agnt.cgi-64 must be renamed.
- update.pkg: Tasking payload (encrypted), generated by the Aeris builder and fetched directly by the implant
- fls/: Results directory (the CGI program will write each exfiltrated payload to this directory using the {timestamp}-{six-character suffix}.pkg naming convention)

Note that we have provided source code for the CGI program in addition to statically-compiled 32 and 64 bit binaries so that the operators can modify the CGI program if needed.

When deploying a tasking payload, the operator should copy the file directly to the implant's document root directory on the LP (e.g., /var/www/{implant_id}) and name the file update.pkg.

POSTFIX/SMTP LP CONFIGURATION

Aeris includes a tarball that contains a patched and pre-built version of Postfix-2.10.0 that is suitable for use on Ubuntu 12 Server. You can also build your own version of Postfix and cyrus-sasl using the scripts provided in postifx/build. Please see the README file in that directory for guidance on building a custom Postfix server.

Once Postfix is deployed, it must be configured to accept mail and provide tasking: - The payload should be placed in /var/spool/.emlrcpt - Edit /etc/aliases so that e-mail for the implant's ID is forwarded somewhere useful, then run newaliases - Run saslpasswd2 -c -u localhost , and enter the password, as shown in the implant receipt to allow the implant to authenticate with Postfix

APPENDIX

A. REPLICATING AN EXISTING AERIS INSTANCE

The Aeris distribution includes a capability (patcher.py) that enables an operator to recreate an existing Aeris instance. To do this, run the patcher as follows:

```
# ./patcher.py {path to unpatched binary} {config file (generated during initial build)}
{desired on-target config file name} {output (patched binary) path}
```

As the name suggests, patcher modifies the specified unpatched Aeris binary (see Section I) by adding in both the encrypted configuration and the on-disk config file name and writing the result to the specified output path. By providing the config file and and its on-target name for a previously-deployed binary, the operator can reconstitute the binary that was deployed on the corresponding target.

Aside: Technically, the patcher will not create a pure byte-for-byte replica of an existing deployment, as it always inserts some random data after the configuration during the patching process. However, as this random data is ignored during execution, the binary is, for all intents and purposes, a byte-for-byte replica. For this reason, we say that the patcher yields a functional replica of an existing deployment.

B. GENERATING A NEW LP

We have also included a capability (mklp.py) that allows the operator to create a new listening post for a previously-deployed Aeris instance. (Note that the builder generates files for all LPs specified when building a new instance.) Use mklp as follows:

```
# ./mklp.py {XML receipt file} {URL for the new LP} {target ID}
```

The script will generate an LP-specific output directory whose structure is similar to that described above for the builder (Section II). All files are UNCLASSIFIED (i.e., no private CA or target information is contained in the output directory).

C. KEY MANAGEMENT

Each implant instance has a unique certificate authority associated with it. The CA's private key is used to sign the implant's certificate as well as certificates for each LP associated with the implant in question. If anyone actually reads this paragraph, he or she is entitled to a small monetary prize courtesy of the Aeris team lead. Implant- collected data cannot be decrypted without the CA's private key; hence, this key is considered SECRET//NOFORN and must be maintained on a classified network. All keys and certificates (CA, target, and LP) are 2048 bits in size.

Other keys (LP and target) must, by definition, reside on Internet-enabled, âlow-sideâ systems and are considered UNCLASSIFIED. However, some further restrictions are recommended. In particular, target keys should never reside on an LP, and LP keys should never reside on a target. Since the target and LP use mutually-authenticated SSL, the CA certificate must reside on both systems.

All keys are available in the build repository as described in Section II. It is recommended that this repository be preserved somewhere for posterity since the CA private key cannot be reconstituted if lost. We also recommend that keys and configurations for new LPs be preserved in similar fashion (see Appendix B).

CRYPTOGRAPHIC DESIGN

Aeris uses a cryptographic implementation that is compatible with the NOD Cryptographic Specification (NSPEC-001). All cryptographic algorithms are based on PolarSSL 1.0.0, which is statically-compiled into the Aeris binaries. Specifics of our implementation are outlined below.

ENCRYPTION OF DATA

A tasking payload is generated on a high-side system and consists of three blocks:

[HASH | KEY | DATA]

The data block consists of AES-256-encrypted tasking data. AES keys are generated by reading data from /dev/random and hashing the output with SHA-384. The 48 bytes that result consist of a 32-byte key and a 16-byte initialization vector. This key / vector combination is then encrypted with the target's public key and stored in the key block. Finally, a SHA-512 hash is computed over the key and data blocks. This hash is signed with the CA's private key, and the result is stored in the hash block.

When data is exfiltrated from a target, the encrypted payload takes the following form:

[KEY | DATA | HASH]

The data block consists of AES-256-encrypted data that has been collected from a command request or series of command requests. Here, keys and initialization vectors are generated with PolarSSL's havege random number generator. The 48-byte key / vector combination is encrypted with the CA's public key (certificate), and the result is stored in the key block. Once all data have been encrypted, and the data block has been finalized, a SHA-512 hash is computed over the entire payload. This hash is signed using the target's private key, and the result is stored in the hash block. Note that, because the CA private key resides only on the high side, the original payload can be recovered only on the high side.

ENCRYPTION OF COMMUNICATIONS

All communications between the target and an LP are encrypted using mutually- authenticated TLS (with AES-256 as the preferred cipher). Thus, Aeris uses two separate layers of encryption: One on the data itself (described above) and another on the communications.

The data exchange protocol is HTTP. Tasking payloads are fetched using HTTPS GET, and collected data are exfiltrated using HTTPS POST. When exfiltrating data, Aeris does not use any specific POST fields; instead, it simply appends the data to the HTTP header as one large âblobâ The server and/or processing script should save this blob to a file verbatim.

Both client and target certificates must validate against the CA certificate. Note also that an Aeris LP must be SNI-enabled so that the same LP can support multiple domains and can resolve a specific domain to a specific processing script and a specific set of keys. If the Apache domain does not match the certificate common name, either the server will drop the TLS connection or the implant will fail to validate the server certificate.

ENCRYPTION OF DATA ON DISK

Data may be stored on disk before it is exfiltrated. When this occurs, each data file takes precisely the form described above for data exfiltration.