Campaign Listener Implementations for Hyper-V

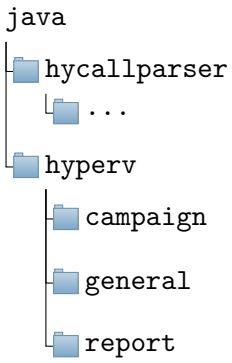

Figure 5.7.: Structure of the `hyperv` Package

The implementation of the Hyper-V-tailored campaign listeners is located pretty much exclusively in the subpackages of the package `hyperv`. In addition to that, an implementation of the `CampaignListenerProvider` interface has been created in the package `hyperv.visitors_and_listeners.campaignListeners`. It is responsible for reporting all Hyper-V campaign listeners to the compiler.

Package `hyperv.general`

No campaign listeners are implemented in the `general` package. Here, only utility classes exist. The class `ByteTransformUtils` offers methods to convert between byte arrays of arbitrary sizes and the numerical value they store. The transformations expect the binary data to be in Little-Endian format, which means the byte array starts with the least-significant byte at index 0. The Little-Endian format is chosen because the injection driver stores data this way (native byte ordering of x86_64 is Little-Endian). The conversion is needed in both directions because the binary campaign generator has to transform integer values to byte sequences of specific sizes, while the report generators read byte sequences from the binary log and need to retrieve their numerical value for human-readable display.

The knowledge about hypercall call codes, offsets, and sizes of the parameters on the input page, as well as the sizes of the output values on the output page, is provided by the `HyperVHypercallDict` class. It is essentially just a wrapper around the hypercall data stored in a JSON file to allow the other classes to retrieve this information by calling Java methods instead of having to handle JSON parsing at multiple locations. This class also implements the method `getRequiredInputPageSize`, which returns how small the encoded input page can be for the binary campaign, based on the hypercall's parameters.

Package `hyperv.campaign`

All the classes related to generating binary campaign files for the injection driver reside in the `campaign` package. For helper classes, there is the `InputPageGenerator`, which allows storing values at offsets, and, based on those, can build a byte array to be placed in the binary campaign. Furthermore, `HyperVHypercall` is a class for representing hypercall entries of the binary campaign. Thus, it stores the call code, the repetition count, and an instance of `InputPageGenerator`. The class provides a method to generate a byte array containing a corresponding binary campaign entry, followed by the variably sized input page.

`HyperVCampaignGenerator` is the implementation of the `CampaignListener` interface that is supposed to produce the campaign files for the injection driver. No output is written

until, at the end of the campaign, the interface method `finish` is called. During execution, a list of bytes is used to store the binary entries. The file path where the output should be written to has to be passed as an argument to the interface method `setParameters`.

When `encodeHypercall` is called during a campaign, the `HyperVHypercallDict` is used to retrieve the call code, and offsets and sizes of the parameters. With this information, first, an `InputPageGenerator` is created, and the parameters are written to it, then, a `HyperVHypercall` is instantiated. If there is another instance remembered, with the same call code and identical input page, the count of the old instance is increased. If the hypercalls differ, the remembered one is converted to bytes and appended to the campaign. The new instance is remembered to compare it with the next hypercall. By using this approach, a sequence of identical hypercalls can be encoded in one campaign entry, without modifying the list of bytes that represents the campaign.

Package `hyperv.report`

The campaign listeners supposed the make binary campaign logs human-readable are located in the `report` package. The `ResultFileWalker` is a wrapper around the binary log file and allows to step through it while delivering arbitrarily sized byte arrays. There is also a wrapper class for the flags indicating which information is included in the campaign: the `LogFlagDecoder` reads the bits representing the flags and uses boolean values to make them easily accessible. Because raw timestamps are 8-byte values holding the number of 100 nanosecond-intervals since January 1, 1601, they have huge values. Converting a series of timestamps to a series of relative timestamps with the first time being 0, and all difference between timestamps staying the same, is provided by the `RelativeTimeTracker`.

One of the campaign listeners implementations is the `HyperVConsoleReporter`, which prints the information - inputs as well as logs - about hypercalls and delays to console. A sample of its output can be seen in Listing 5.8.

Listing 5.8: `HyperVConsoleReporter` Output

```
Hypercall:
        Name: HvFlushVirtualAddressSpace
        Exec time: 6.8us
        Result value: 0
Delay:
        Expected: 1000us
        Actual: 1010.6us
Hypercall:
        Name: HvSignalEvent
        Exec time: 5.4us
        Result value: 18
```

The other campaign listener implemented is `HyperVCSVTimingReporter`, which was used for parts of the evaluation in this thesis. Depending on whether execution times or raw timestamps are included in the binary log file, it outputs the event type and the timing information in the CSV (Comma-Separated Values) format. The event type is either "delay" or the name of the hypercall. In the case of timestamps, the `RelativeTimeTracker` is used to make the times more readable. If no file path argument is supplied, output is written to console, otherwise to the specified file.

Listing 5.9 shows a sample of the CSV-formatted output for a log file containing timestamps.

Listing 5.9: `HyperVCSVTimingReporter` Output

```
Event,start,end
hcall_HvFlushVirtualAddressSpace,0.0,9.9
delay,62.5,1396.4
hcall_HvSignalEvent,1508.5,1516.1
delay,1565.3,1615.6
hcall_HvSignalEvent,1678.1,1682.4
Compile time: 83
```