# LABELESS – NO MORE

*Alexander Chailytko & Aliaksandr Trafimchuk*
Check Point Software Technologies, Israel

Email {alexanderc, aliaksandrt}@checkpoint.com

## ABSTRACT

How many times have you been in this situation: you've dumped a decrypted body of really hard core malware after unpacking, and several hours of work later you have a perfectly documented *IDA* database (IDB), with some 'blind spots' that need to be investigated dynamically. You drop the executable to *Olly* and… have absolutely no idea what's going on, since there are no labels, function names or comments. All you've got is jmp loc_00401000, call 0040135F, etc.

One possible answer is to export a '*.map' file from *IDA*, and use the 'mapimp' plug-in for *Olly* to import it. However, there is one strong limitation: the 'mapimp' plug-in does not support rebasing of the module, making work with packed malware (especially if it injects itself into other processes) basically impossible. Another disadvantage of 'mapimp' is that when you make changes to your IDB, you cannot update information in *Olly* in real time.

What's the solution? Read on to learn more about 'Labeless'.

Consider the following situation: at the beginning of our research we have an empty *IDA* database and binary code without labels and comments in *Olly*. After some dynamic analysis we will name a few functions. If, for some reason, an analysis is restarted, all the collected information will be lost. So we will have to fill in the labels one more time or debug without them at all. Labeless provides a solution for that.

Since IDB can be treated as a solid information storage, we are filling it with names and comments during analysis and Labeless will automatically synchronize all information with the debugged process in *Olly*, so we can return to any execution point we want while debugging (for example, using VM snapshots or just restarting the process) without losing any previously collected data about the debugged process.

Another benefit we provide is the functionality to execute Python scripts from *IDA* inside *Olly* with the full support of *Olly* API. Just imagine: you can write a script in *IDA*, collect some info, transfer it to *Olly*, execute it there, feed the results back to *IDA* and propagate all changes to your IDB. In other words, you have almost unlimited possibilities.

Last but not least, we've made dumping easy and called this feature 'IDADump'. It can dump memory straight into IDB, automatically fixing imports on the fly to make this a seamless experience. We support the dumping of multiple modules' memory into one IDB, so you have a single 'research case' IDB with all the information in one place.

More than that, Labeless will be available as open source, so you are welcome to use it.

## 1. OVERVIEW

Labeless is a plug-in system for dynamic, seamless and real-time data exchange between *Olly* and an *IDA* database. Synchronization of labels is performed correctly even if the module has been relocated, which is usually the case with multi-stage packed malware or following injections.

Labeless also allows remote execution of Python scripts inside *Olly* (PyExCore).

## 2. PYEXCORE

PyExCore is one of the components of the *Olly* side plug-in that implements a Python wrapper around the *Olly* API (using SWIG [1]). An important feature is the RPC [2] server that implements the synchronization of labels and dumping of code inside the *IDA* database (IDADump).

PyExCore comes with a set of helper scripts. These are used as a supporting backend for handling RPC requests, logging, PE analysis, safe memory read and much more. Figure 1 shows the structure of PyExCore.

### 2.1 Remote Execution of Python code inside Olly

The Python wrappers around the *Olly* API allow us to control the debugging process and internal data structures (memory map, SEH, stack, call stack, context, etc.) that provide the functionality to dump the required code regions. This is useful for dynamic unpacking, hooking and fixing stolen bytes.

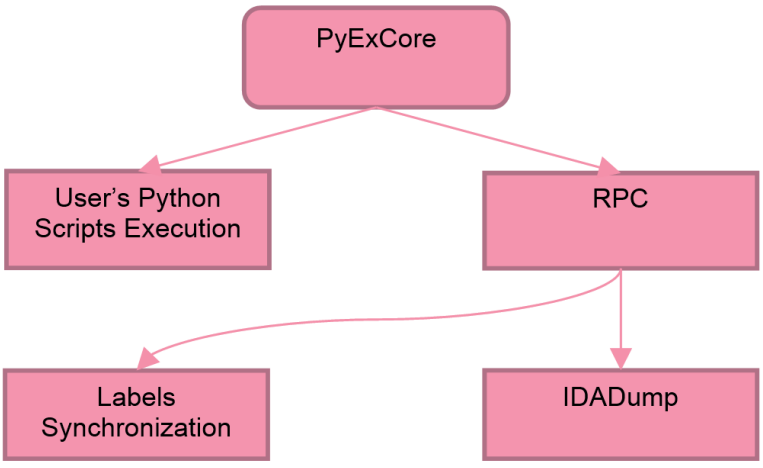


*Figure 1: PyExCore structure.*

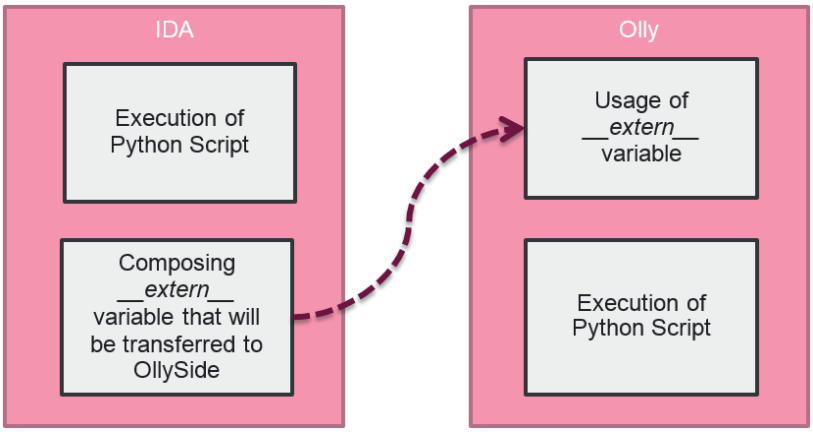


*Figure 2: How our system works.*

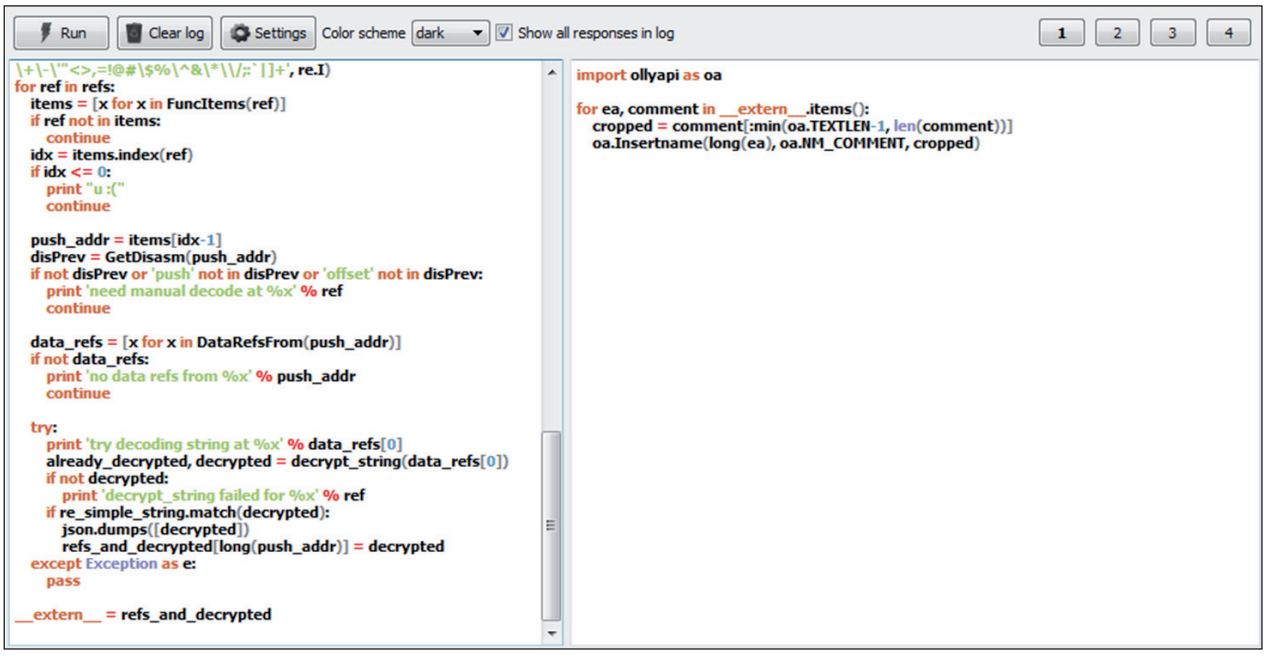


*Figure 3: Example of script execution (IDA side script on the left, Olly side on the right).*

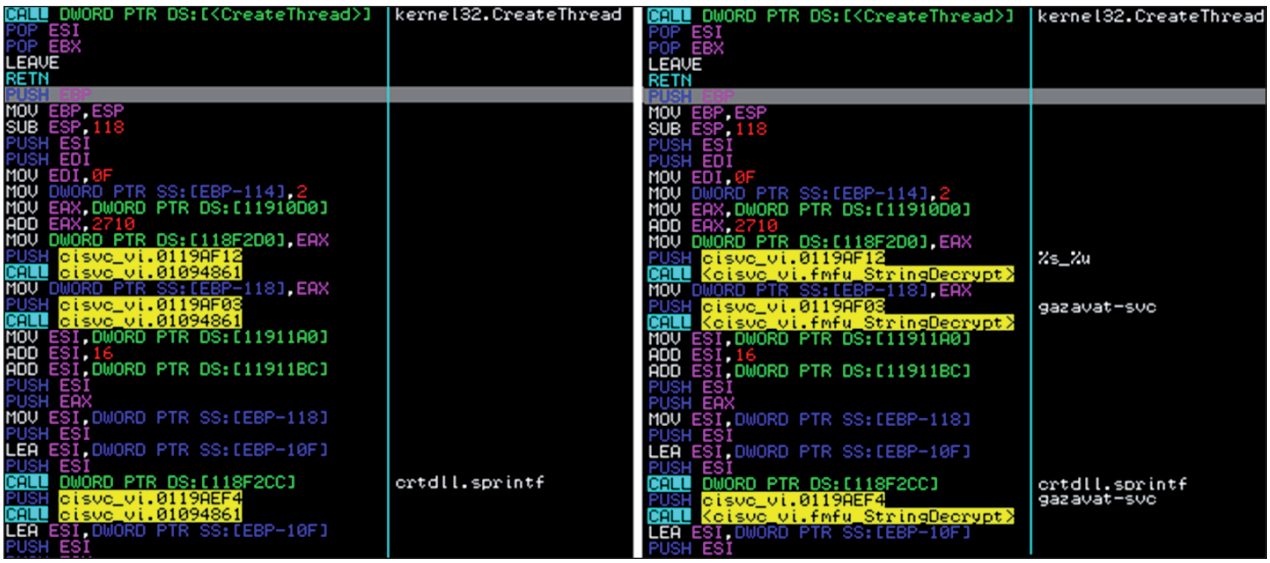


*Figure 4: Example of synchronization of labels and comments (before and after).*

Figure 2 shows how our system works.

We can implement helper scripts on the *Olly* side, which can then be called with a single line of Python code from *IDA*.

Any data available to *Olly* can be easily sent to *IDA* and vice versa. As a result, the information contained in the IDB is the best of both worlds (static and dynamic).

Figure 3 shows an example script that will decrypt all strings in the specific sample and propagate them into comments both in *Olly* and *IDA*.

### 2.2 Label synchronization

Every change of method name or label in *IDA* is automatically propagated to *Olly*. We support manual synchronization of all names/labels using the 'Sync now' button in the plug-in 'Labeless configuration' view (see Figure 6). In addition, the 'Remote module base' field synchronizes properly if the module bases in *IDA* and *Olly* are different. Aside from its simplicity, the remote module base is a very powerful feature of the plug-in.

Compare the two screenshots shown in Figure 4: without Labeless and with Labeless.

### 2.3 IDADump

IDADump provides dynamic dumping of debugged process memory regions. The following is the list of prime examples where it can be useful:

- The debugged process has an extracted/temporary/injected module which doesn't appear in the modules list.
- A heap spray was performed.
- There is no valid PE header.
- There is a corrupted import table.
- The analysed module contains stolen bytes.

We can take the dumped memory region and put it in the *IDA* IDB, automatically recovering imports (including direct/indirect calls/jumps) with *Olly* functionality. There is no need for ImpRec or Scylla to search for the memory regions that contain the real IAT, as we get that information dynamically from the debugged process itself.

As a result, we have a lot of memory regions that may even represent different modules (if the unpacking process is multi-staged).We can build valid references between multiple dumped memory regions, and end up with one large IDB which contains all the information relating to the specific research case.

Here is how the IDADump feature works:

1. Get the memory map (RPCT_GET_MEMORY_MAP) of the target process and prompt the user to choose the regions which need to be dumped.
2. Check if the PE header is valid (RPCT_CHECK_PE_HEADERS) and get the list of the exports and section information.
3. Get the list of loaded modules in the debugged process and parse all their export entries. These are saved for later use.
4. Load the selected memory (RPCT_READ_MEMORY_REGIONS) of the debugged process (TARGET_MEMORY) into the *IDA* database.
5. Export the entries from step #2 and add to the exports view in *IDA*.
6. TARGET_MEMORY is analysed for references to external APIs (RPCT_ANALYZE_EXTERNAL_REFS). As a result, we get a list of all possible places where APIs could be called.
7. Disassemble all possible variants of code with the byte from step 1 (this helps cover jumps inside other instructions) to handle the obfuscated code.
8. References to old and invalid APIs inside TARGET_MEMORY are fixed and placed in one 'extern' segment in *IDA*.

### 2.4 RPC Server

This component is used for handling binary RPC requests. It is based on the TCP server with binary RPC protocol (using the *Google Protobuf* [3] library).

## 3. USER INTERFACE

### IDA:

Our GUI is based on the Qt framework [4]. We implement the top-level menu in *IDA* with the 'Labeless' caption, as shown in Figure 5.

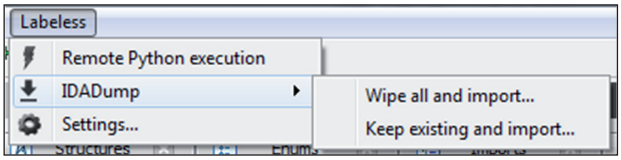


*Figure 5: Labeless IDA plug-in menu.*

The Labeless configuration view is shown in Figure 6.

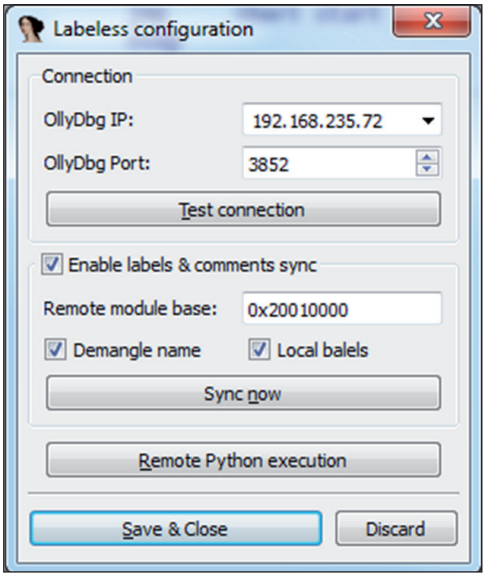


*Figure 6: Labeless configuration view.*

We specify the IP address of the machine where *Olly* is running, the port number (the same one that you will enter in *Olly*) and the remote module base.

Our script editor has syntax highlighting as well as auto completion for Python scripts, allowing you to compose your own scripts without switching to an external editor. This can be seen in Figure 3.

The main view of the plug-in in *Olly* is shown in Figure 7.

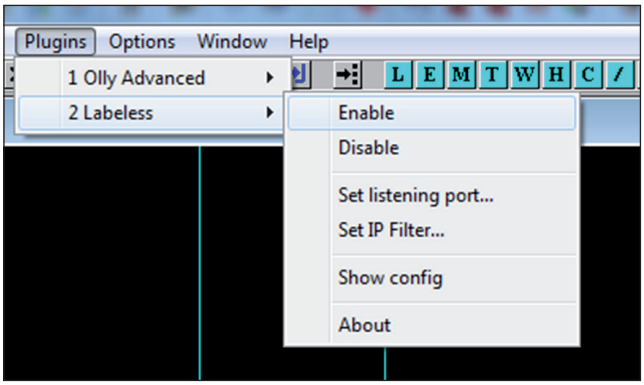


*Figure 7: Labeless Olly plug-in menu.*

As *Olly* is actually a backend, there are not a lot of things to display.

## 4. COMMUNICATION PROTOCOL

### Overview

We created a custom binary RPC protocol using *Google Protobuf*. The RPC server running on the *Olly* side provides RPC service functionality over the TCP stack. RPC requests are handled by PyExCore implemented on the *Olly* side. To execute an RPC request, we provide an RPC::Execute command:

```
message Execute {
   optional string script;
   optional string script_extern_obj;
   optional bytes rpc_request;
   optional bool background;
   optional uint64 job_id;
}
```

It accepts the following parameters:

- 'script' – script that will be executed inside *Olly*.
- 'script_extern_obj' – any Python object that can be serialized with the JSON library. Used to transfer data to *Olly* prepared by *IDA*.
- 'rpc_request' – serialized RPC request.
- 'background' – this flag indicates that rpc_request will probably take a long time and the RPC engine should place it in the queue, returning job_id immediately.
- 'job_id' – an optional field. Setting this field tells the RPC engine not to execute rpc_request, but to check if the result for the command with the specified job_id is ready and return it.

While using the RPC::Execute command, you can specify either the script or rpc_request parameter, but not both at the same time. Therefore, if the RPC::Execute command has a script field, this script is executed in Python inside *Olly*. If not, the rpc_request field should contain the encoded RpcRequest structure:

```
message RpcRequest {
  enum RequestType {
     RPCT_UNKNOWN;
     RPCT_MAKE_NAMES;
     RPCT_MAKE_COMMENTS;
     RPCT_GET_MEMORY_MAP;
     RPCT_READ_MEMORY_REGIONS;
     RPCT_ANALYZE_EXTERNAL_REFS;
     RPCT_CHECK_PE_HEADERS;
  }
  required RequestType request_type;
  optional MakeNamesRequest make_names_req;
  optional MakeCommentsRequest make_comments_req;
  optional ReadMemoryRegionsRequest read_memory_
regions_req;
  optional AnalyzeExternalRefsRequest analyze_
external_refs_req;
  optional CheckPEHeadersRequest check_pe_headers_
req;
}
```

| Field | Description |
|---|---|
| 'request_type' | RPC request type |
| 'make_names_req' | RPCT_MAKE_NAMES request |
| 'make_comments_req' | MAKE_COMMENTS |
| 'read_memory_regions_req' | RPCT_READ_MEMORY_REGIONS request |
| 'analyze_external_refs_req' | RPCT_ANALYZE_EXTERNAL_REFS request |
| check_pe_headers_req' | RPCT_CHECK_PE_HEADERS request |

Each RPC request is followed with an rpc::Response by the server:

```
message Response {
   enum JobStatus {
      JS_FINISHED ;
      JS_PENDING;
   }
   required uint64 job_id;
   optional string std_out;
   optional string std_err;
   optional string error;
   optional bytes rpc_result;
   optional JobStatus job_status;
}
```

| Field | Description |
|---|---|
| 'job_id' | Assigned job ID. |
| 'std_out' | Output stream content. |
| 'std_err' | Error stream content |
| 'error' | RPC error description. |
| 'rpc_result' | Encoded result of RPC command that is filled in with data according to the RPC request type. |
| 'job_status' | Job status: finished, or pending if the initial command was marked to be executed in the background and the job was not finished yet. |

## RPC requests and responses

### *RPCT_MAKE_NAMES*

This request is used to create names (labels) inside *Olly*. *Olly* receives a list of name structures and propagates them. No response is expected.

Request:

```
message MakeNamesRequest {
  message Name {
   required uint32 ea;
   required string name;
  }
  repeated Name names;
  required uint32 base;
  required uint32 remote_base;
}
```

| Field | Description |
|---|---|
| 'names' | List of name structures |
| • 'ea' | Effective address where the name is applied |
| • 'name' | The name itself |
| 'base' | Address of the module in IDA |
| 'remote base' | Address of the module in Olly |

Response: N/A

### *RPCT_MAKE_COMMENTS*

This request is used to create comments inside Olly and works exactly like the RPCT_MAKE_NAMES request. The only difference is that it propagates comments and not names. No response is expected.

Request:

```
message MakeCommentsRequest {
  message Name {
    required uint32 ea;
    required string name;
  }
  repeated Name names;
  required uint32 base;
  required uint32 remote_base;
}
```

Response: N/A

### *RPCT_GET_MEMORY_MAP*

This request gets a list of memory regions from *Olly*. No additional fields are sent.

Request: N/A

Response:

```
message GetMemoryMapResult {
   message Memory {
     required uint32 base;
     required uint32 size;
     required uint32 access;
     required string name;
   }
   repeated Memory memories;
}
```

| Field | Description |
|---|---|
| 'memories' | List of memory regions |
| • 'base' | Memory base |
| • 'size' | Region size |
| • 'access' | Memory access rights (protect flags) |
| • 'name' | Olly name of the memory region (if the region belongs to a loaded module) |

### *RPCT_READ_MEMORY_REGIONS*

This request safely reads the specified memory regions. If the PAGE_GUARD flag is hit during the read, it is automatically removed and then restored after the read operation is finished. The ReadMemoryRegionsResult response is returned.

Request:

```
message ReadMemoryRegionsRequest {
  message Region {
    required uint32 addr;
    required uint32 size;
  }
  repeated Region regions;
}
```

| Field | Description |
|---|---|
| 'regions' | List of memory regions to read |
| • 'addr' | Address to read from |
| • 'size' | Size of memory to read |

Response:

```
message ReadMemoryRegionsResult {
  message Memory {
     required uint32 addr;
    required uint32 size;
    required bytes mem;
  }
  repeated Memory memories;
}
```

| Field | Description |
|---|---|
| 'memories' | List of read memory regions |
| • 'addr' | Starting address of the memory region |
| • 'size' | Size of the memory region |
| • 'mem' | Memory that was read |

### *RPCT_ANALYZE_EXTERNAL_REFS*

We perform two types of analysis with this request. The first analysis type searches for API constants (addresses) in the specified region. The second analysis type uses the increment step to take every byte sequence of MAX_INSTRUCTION_LENGTH and tries to disassemble them. If the disassembled instruction contains an API constant as an operand, it is added to the response.

Request:

```
message AnalyzeExternalRefsRequest {
   required uint32 ea_from;
   required uint32 ea_to;
   required uint32 increment;
```

```
    required uint32 analysing_base;
    required uint32 analysing_size;
}
```

| Field | Description |
|---|---|
| 'ea_from' | Effective address, where to start the analysis |
| 'ea_to' | Effective address, where to end the analysis |
| 'increment' | Incremental step for analysis |
| 'analyzing_base' | Base address of the first region |
| 'analyzing_size' | Size of all regions |

Response:

```
message AnalyzeExternalRefsResult {
 message PointerData {
   required uint32 ea;
   required string module;
   required string proc;
 }
 message RefData {
        enum RefType {
        REFT_JMPCONST;
        REFT_IMMCONST;
        REFT_ADDRCONST;
    }
    required uint32 ea;
    required uint32 len;
    required string dis;
    required uint32 v;
    required RefType ref_type;
    required string module;
    required string proc;
 }
 message reg_t {
   required uint32 eax;
   required uint32 ecx;
   required uint32 edx;
   required uint32 ebx;
    required uint32 esp;
         required uint32 ebp;
         required uint32 esi;
         required uint32 edi;
   required uint32 eip;
 }
 repeated PointerData api_constants;
 repeated RefData refs;
 required reg_t context;
}
```

| Field | Description |
|---|---|
| 'api_ constants' | List of found APIs from analysis #1 |
| • 'ea' | Effective address of the API procedure start |
| • 'module' | Module name where the API is found |
| • proc' | Procedure name of the API |
| 'refs' | List of successful disassembled instructions where any API constant was found as an address inside that instruction. List of found APIs from analysis # 2. |
| 'ea' | Effective address of the instruction |
| 'len' | Length of instruction |
| 'dis' | Disassembled form of instruction |
| 'v' | Value of the constant |
| 'ref_type' | Reference type |
| module' | Module name where the API is found |
| 'proc' | Procedure name of the API |
| 'context' | Context structure with all the registers from target process |

### *RPCT_CHECK_PE_HEADERS*

This request is used to check whether the PE header is correct and get the export entries and section information.

Request:

```
message CheckPEHeadersRequest {
    required uint32 base;
    required uint32 size;
}
```

| Field | Description |
|---|---|
| 'base' | Base of the first region, where the PE header is possibly located |
| 'size' | Size of all selected regions |

Response:

```
message CheckPEHeadersResult {
     message Exports {
         required uint32 ea;
   required uint32 ord;
   optional string name;
     }
     message Section {
         optional string name;
         optional uint32 va;
         optional uint32 v_size;
         optional uint32 raw;
         optional uint32 raw_size;
         optional uint32 characteristics;
     }
     required bool pe_valid;
     repeated Exports exps;
     repeated Section sections;
}
```

| Field | Description |
|---|---|
| 'pe_valid' | Flag if PE header is valid or not |
| 'exps' | List of exports structures |
| • 'ea' | Effective address of the export entry |
| • 'ord' | Ordinal number of the export entry |
| • 'name' | Name of the export entry |
| 'sections' | List of section structures |
| • name' | Name of the section |
| • 'va' | Virtual address of the section |

| Field | Description |
|---|---|
| • 'v_size' | Virtual size of the section |
| • 'raw' | Raw offset of the section |
| • 'raw_size' | Raw size of the section |
| • 'characteristics' | Characteristics of the section |

## 5. SUMMARY

Labeless significantly reduces the time spent transferring already reversed/documented code information from *IDA* (static) to the debugger (dynamic). There is no need to do the same job twice. In addition, you can document and add data to the IDB on the fly. Your changes are automatically propagated to *Olly*, even if you restart the virtual machine or *Olly* crashes. You will never lose your research.

This solution is highly upgradable. You can implement any helper scripts in Python on the *Olly* side and then just call them from *IDA* with one line of code, parsing the results and automatically propagating changes to the IDB.

## REFERENCES

[1] http://www.swig.org/.

[2] https://en.wikipedia.org/wiki/Remote_procedure_call.

[3] https://developers.google.com/protocol-buffers/.

[4] http://qt.io/.