Scout Documentation

Release 2.0.0

Eyal Itkin

May 26, 2021

User Guide:

Scout consists of two main parts:

1. Server Side - C code
2. Client Side - Python code

The server is the debugger that is being sent / injected into the debugee, and the client is the user API that issues the instructions.

CHAPTER 1

Folder Structure

- **docs** - This documentation
- **examples**
- **embedded_scout** - Use case example for an “Embedded Mode” compilation
- **kernel_scout** - Use case example for a Linux “Kernel Mode” compilation
- **src**
- **scout** - Source code for the debugger (core of the server side)
- **utils** - Python compilation scripts and network API for the client/server
- **tests** - A simple exploit_me.c for checking PIC compiled binaries

Note: More information on the different compilation modes can be found under the “Compilation Modes” section.

CHAPTER 2

Installation

- Installing the python package: `python3 setup.py install`
- Dedicated compilers: A list of compilers per-architecture is found on `compilers.txt`

CHAPTER 3

Beginner's Guide

When deploying Scout for a new research project, we recommend to pick the suitable use case out of the two examples. Both the "Embedded Mode" and the Linux "Kernel Mode" use case examples are supplied so that they will serve as templates for new projects.

The project part of the server side consists of the following parts:

- Main (Init) code - Only needed in the Embedded Mode
- Project Instructions - When adding custom instructions
- Loader PIC - globals / plt for the loader in an Embedded Mode
- Project PIC - globals / plt for the project's extensions in an Embedded Mode
- Compilation Script - `compile_scout.py` with the compile instructions

CHAPTER 4

Compilation Modes

Scout is a configurable debugger, that could be deployed in several different environments:

- Linux User-Mode Process - “User Mode”
- Linux Kernel Driver - “Kernel Mode”
- Linux In-Process Debugging - “User Mode” + “PIC Mode”
- Embedded “In-Process” Debugging (low privileges) - “PIC Mode” + “User Mode”
- Embedded “In-Process” Debugging (high privileges) - “PIC Mode” + “Kernel Mode”

To decide what will be the suitable compilation mode / architecture flags, one should check the following parameters. Each of the defined parameters is a C MACRO (define) that controls the behavior (and compilation) of the resulting binary.

Important Note: When using `scoutCompiler`, it will automatically generate most of the needed flags by deducing the right values based on the architecture and configuration flags that are supplied to it. Please see the `embedded_scout` example for more info.

4.1 Target Endianness

- SCOUT_BIG_ENDIAN - Scout is executed on a Big Endian architecture
- SCOUT_LITTLE_ENDIAN - Scout is executed on a Little Endian architecture

Only one of above flags can be defined. If none are defined the base library will define “SCOUT_LITTLE_ENDIAN” on it’s own.

4.2 Target Bitness

- SCOUT_BITS_32 - Scout is executed on a 32 bit machine
- SCOUT_BITS_64 - Scout is executed on a 64 bit machine

Only one of above flags can be defined. If none are defined the base library will define "SCOUT_BITS_32" on it's own.

4.3 Target CPU Architecture

- SCOUT_ARCH_INTEL - Scout is executed on an Intel (x86 \ x64) CPU
- SCOUT_ARCH_ARM - Scout is executed on an ARM (maybe thumb mode) CPU
- SCOUT_ARCH_MIPS - Scout is executed on a MIPS (*not* mips16 mode) CPU

Only one of above flags can be defined. If none are defined the base library will define "SCOUT_ARCH_INTEL" on it's own.

Additional Flags: SCOUT_ARM_THUMB - Scout will be executed on an ARM cpu in Thumb mode. Can only be used together with the "SCOUT_ARCH_ARM" flag.

The flags are needed only in PIC mode, in which we use inline assembly.

4.4 Target Permission Level

- SCOUT_MODE_USER - Scout is executed in User-Mode (& low CPU privileges)
- SCOUT_MODE_KERNEL - Scout is executed in Kernel-Mode (& high CPU privileges)

Only one of above flags can be defined. If none are defined the base library will define "SCOUT_MODE_USER" on it's own.

"SCOUT_MODE_KERNEL" will also be the right choice for an RTOS (Real-time OS) in which every task / our task has high privileges. The flag will lead to the definition of "SCOUT_HIGH_PRIVILEGES" by the compilation environment.

Important Note: As flushing the CPU cache usually requires high privileges, the per-architecture implementation will only be available if compiling using the "SCOUT_MODE_KERNEL" flag.

4.5 Position Independent Mode - SCOUT_PIC_CODE

Scout will be compiled for full Position Independent Code (PIC) mode. Any access to an external function / global variable will pass through a unique "Context" object. Read the section about "PIC Compilation" for more information.

"SCOUT_PIC_CODE" will lead to the definition of "SCOUT_ISOLATED_ENV" by the compilation environment, because a PIC blob will always be isolated from the environment, and won't have the luxory of a proper executable loader such as "ld.so".

4.6 Host LibC Implementation

When injecting our (PIC) code into a host binary, we should make sure to use the proper constants for the matching standard library implementation that is used by the respective binary:

- SCOUT_HOST_GLIBC - The used library is Glibc
- SCOUT_HOST_UCLIBC - The used library is uClibc (or uClibc-NG)

If none are defined the base library will define "SCOUT_HOST_GLIBC" on it's own.

4.7 Loader Flags

- SCOUT_LOADER - We are now compiling a loader (that might be using it's own pic plt / globals).
- SCOUT_LOADING_THUMB_CODE - The loader will load a Scout that was compiled to be executed on an ARM cpu in Thumb mode.
- SCOUT_RESTORE_FLOW - The default loaders (`tcp_client_server.c`, `tcp_loader_server.c`) will clean-up after themselves if the loaded scout will finish the endless loop.

If the loader will be compiled to be Position Independent (PIC), which is probably the most common use case, it will also define a new flag of "SCOUT_SLIM_SIZE", to help shrink the size of the binary (to serve as an effective shellcode). Under this definition the TCP server would expect the following flags (if needed):

- SCOUT_TCP_CLIENT - There is a need to include the feature of a TCP client
- SCOUT_TCP_SERVER - There is a need to include the feature of a TCP server
- SCOUT_TCP_SEND - There is a need to include the ability to reliably send TCP messages

4.8 Additional Flags:

- SCOUT_INSTRUCTIONS - Scout is going to use the instructions api (using the TCP server for instance)
- SCOUT_DYNAMIC_BUFFERS - Scout will dynamically `malloc()` buffers to be used by the tcp server. Otherwise static buffers will be used.
- SCOUT_PROXY - Scout is going to act as a proxy (user scout passing instructions to a kernel driver for instance)
- SCOUT_MMAP - Should scout's loaders use `mmap()` and `mprotect()` when loading (if defined) or should they simply use `malloc()` (if undefined)

CHAPTER 5

Position Independent Code (PIC) Mode

In PIC mode Scout is no longer loaded using the native loader of the operating system, since it is injected into a running executable. This means that Scout is compiled to be fully independent, enabling it to work no matter where it was injected to.

5.1 Entry Point

The entry point to our executable blob is at offset 0, meaning that we can literally jump into our buffer's start.

IMPORATNT: This can only be achieved if the `*_pic_wrapper.c` file will be linked as the FIRST file in the list of files. In case there are any errors it is always recommended to check if the compiled files were compiled in a different order.

From the project's point of view, the main function is called `main` (as usual), since this is the function that will be executed by the start stub at offset 0.

5.2 The Context

The PIC Context consists of 4 parts:

- Base scout "PLT"
- Project "PLT"
- Base scout "Globals"
- Project "Globals"

The code is compiled so that any call to an external function (library call) will pass through an indirection layer (named after the ELF's "PLT"). This "PLT" will locate Scout's context object, and will lookup the wanted address in the context's "GOT".

In a similar fashion, the code is compiled (and developed) so that any access to a global variable will pass through an indirection layer. This layer will locate the scout's context object, and will find the offset of the global variable inside the PIC context.

Important Note: There is an edge case in which Scout will be compiled as an Arm Thumb PIC blob, to be executed in the address space of a regular Arm process, or vice versa. In this case the user's compile script could either supply the GOT addresses with their actual (+0/+1) memory addresses, or could use the `is_host_thumb` flag of function `populateGOT` of the `scoutCompiler` to fix it automatically.

5.3 Locating the Context

Our code can be compiled so that it will access the PIC context instead of being linked as an ordinary ELF. Now we only need to tell our code how to be able to locate the context's address in runtime, using a PIC way. The context itself will be embedded INSIDE the full executed binary blob, so that it will be stored in a known relative offset from our code. The final step of deriving the absolute address of the context will be solved using an assembly layer: `*_pic_wrapper.c`.

Important Note: Locating the context is done in assembly, hence this is the only part that depends on the architecture on which we will be executed.

5.4 Expanding the Context

In order to be able to use additional symbols (functions) or globals (global variables), we need to add them to the respective `project_plt.c` and `project_globals.c` files, accordingly. Please make sure that the compilation script will be notified of these additions, so that the GOT will contain the needed addresses and the globals section will be of the correct size.

5.5 Testing - exploit_me

The `exploit_me.c` module in the testing folder lets you test your PIC scout (with / without a loader). As you can see in the "Known Gaps" section (point 2), it is recommended that the test will be done on a setup that contains RWX environments (Linux VMs without the NX bit exposed, for instance).

5.6 Known Gaps

1. In PIC mode our globals will need to be initialized manually as we will have no "init" methods that will be executed before our main function.
2. RWX flags: Since the context is embedded inside the compiled binary, that has executable (X) permissions, it means that the PIC Scout assumes it is loaded into a memory area with RWX permissions so to be able to update global variables.

CHAPTER 6

Scout Instructions

Scout is an instruction-based debugger, that commonly uses a TCP network session on which the instructions are received and their output is being sent.

6.1 Default Instructions

- **NOP** - Used as a Ping (or Keep-Alive) instruction to make sure the debugger is active and responds to commands
- **Memory Read** - Reads (virtual) memory from the given address, and sends it back
- **Memory Write** - Writes a given binary content to a (virtual) memory in the debuggee's address space

Each supported instruction must be pre-registered by the debugger before it enters his server loop, usually by calling `register_all_instructions()`.

6.2 Network API

Each instruction is sent together with a network header that includes the following:

- Instruction ID - 2 Bytes
- Length field - 4 Bytes

The length field specifies the length, in bytes, of the serialized instruction.

Note: All instructions should be serialized to NETWORK order. See `manager\scout_api.py` for a python sample that prepares the instructions for network transmission.

CHAPTER 7

Adding Custom Instructions

Being an instruction-based debugger, Scout supports project extensions.

7.1 Registration - C Code

- Each of the instructions that are added by the project, should be registerred by calling `register_instruction()`.
- The registration should take place inside the function `register_specific_instructions()`.
- This design makes sure that when invoking `register_all_instructions()`, all of the default instructions, and extension instruction, will be registerred correctly.

7.2 Implementation - C Code

In order to implement a new instruction, one should define each of the required parts:

- Instruction ID - must be unique, but not necessarily consecutive
- Minimal Length - minimal amount of bytes needed for a valid instruction (robustness checks)
- Maximal Length - maximal amount of bytes needed for a valid instruction (robustness checks)
- Instruction handler - a handler function with a fixed signature of: `int32_t (*instrHandler)(void * ctx, uint8_t * instruction, uint32_t length)`

Note: The instructions are stored in a global array with a **fixed** capacity. When adding new instructions, one should make sure to adjust this capacity accordingly (both in the C and .py files). The capacity is defined in `scout_api.h` and is set by default to `#define SCOUT_MAX_INSTRS (10)`.

7.3 Examples - C Code

- Embedded mode (`embedded_scout`) - files `project_instructions` (*.c and *.h)
- Linux Kernel mode (`kernel_scout`) - files `driver\scout_kernel_instructions` (*.c and *.h)

7.4 Client Side - Python Code

In the client side, adding a new instructions is even easier, and requires only 2 definitions:

- Defining the Instruction ID (as was defined in the C code)
- Implementing a serializer for the instruction

7.5 Examples - Python Code

- Embedded example (`manager`) - file `embedded_scout_api.py`
- Linux Kernel example (`manager`) - file `kernel_scout_api.py`

CHAPTER 8

Scout's Loader

Embedded debuggers often require an initial code execution vulnerability, that will load up the debugger (instead of using a normal "shellcode"). Exploits for such vulnerabilities are often limited:

1. Size limits on the controlled input
2. Forbidden chars that must not be used

To overcome these limitations, the scout debugger comes with a list of supported default loaders. These loaders are self-contained and are compiled to a smaller memory footprint that the entire scout debugger (especially in case there are project specific extensions to the debugger).

The defualt loaders are:

1. `tcp_client_loader.c` - Connects to a predefined TCP server
2. `tcp_server_loader.c` - Waits for an incoming TCP client

Both of the loaders use the same network protocol:

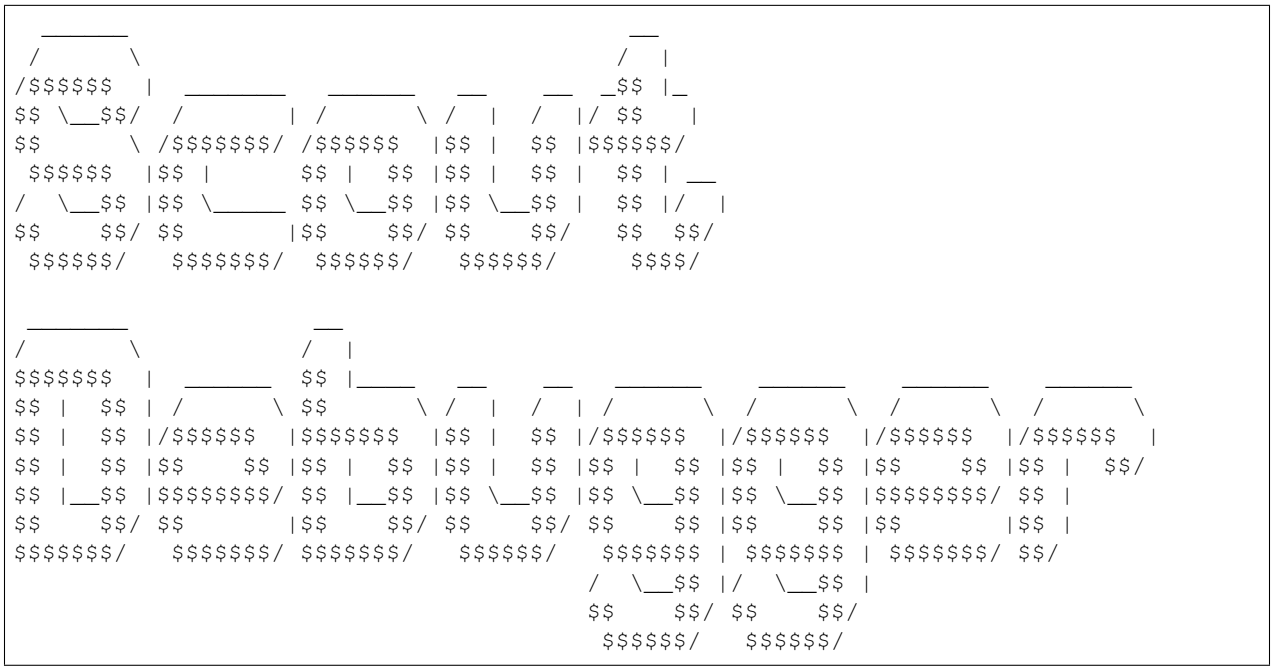
- header - 4 bytes of length field (in network order, i.e. Big Endian)
- data - X bytes of data (X is the value that was sent in the header)

After a TCP connection was established, the loader will follow these steps:

1. The loader will receive the header and `malloc()`/`mmap()` a memory buffer of appropriate size.
2. The data will be received and stored in this memory buffer.
3. The loader will flush the D-cache and I-cache of the buffer (only in architectures were this is needed, and only if we have high enough CPU privileges for it)
4. The loader will `mprotect()` the memory to use the correct permissions (only in architectures were this is needed)
5. The loader will jump into the buffer's start (offset 0, as mentioned in the "PIC Compilation" section)
6. In the flow restore case, once the loaded executable finishes, the loader will free the memory and close the used sockets.

The functions `remoteLoadServer()` and `remoteLoadClient()` in `scout_network.py`, implement the required protocol for communicating with the loader, and loading up the full Scout. A working example is shown under the `manager.py` of the `embedded_scout` example.

Note: In case there are any W^X style limitations in your environment, it is recommended to make sure that the allocated memory for the full debugger will have both Write and eXcutable permissions (should probably use the SCOUT_MMAP flag in this case).



CHAPTER 9

Brief

"Scout" is an extendable basic debugger that was designed for use in those cases that there is no built-in debugger / gdb-stub in the debugee process / firmware. The debugger is intended to be used by security researchers in various scenarios, such as:

1. Collecting information on the address space of the debuggee - recon phase and exploit development
2. Exploring functionality of the original executable by accessing and executing selected code snippets
3. Adding and testing new functionality using custom debugger instructions

We have successfully used "Scout" as a debugger in a Linux Kernel setup, and in an several embedded firmware research projects, and so we believe that it's extendable API could prove handy for other security researchers in their research projects.

9.1 Supported Architectures

- x86 - Intel 32 bit
- x64 - Intel 64 bit
- ARM 32 bit - Little & Big endian (Including Thumb mode)
- MIPS 32 bit - Little & Big endian (Without Mips16 mode)

Future Architectures

- ARM 64 bit - Little & Big endian
- MIPS 16 bit - Little & Big endian
- MIPS 64 bit - Little & Big endian
- ...

9.2 Supported Operating Systems

- Linux - User-mode (PC Mode)
- Linux - Kernel-mode (PC Mode)
- Any Posix-like operating system (Embedded Mode)

9.3 Credits

This projects combines together design and compilation tricks that I learned from many fellow researchers during the years.

9.4 Links

- Original repository - https://github.com/CheckPointSW/Scout
- Maintained repository - https://github.com/eyalitki/Scout

Scout was used in our following research projects:

- https://research.checkpoint.com/sending-fax-back-to-the-dark-ages
- https://research.checkpoint.com/say-cheese-ransomware-ing-a-dslr-camera
- https://research.checkpoint.com/2020/dont-be-silly-its-only-a-lightbulb/
- https://research.checkpoint.com/2020/apache-guacamole-rce/
- https://research.checkpoint.com/mmap-vulnerabilities-linux-kernel

9.5 Contact

- @EyalItkin