slimm609 / **checksec.sh**

Checksec.sh

298 commits | **1** branch | **0** packages | **12** releases | **27** contributors | View license

Branch: **master** | **New pull request** | **Create new file** | **Upload files** | **Find file** | **Clone or download**

slimm609 Merge pull request #120 from ckujau/patch-1 ··· Latest commit `bf85698` on Nov 12

.github	fixed readelf and symbols, added checksec issue template #73	last year
extras	zsh: restore syntax correctness of completion files without short opts	4 months ago
kernel_configs	updated a few checks and added docker for testing	last year
tests	signed update	5 months ago
.gitignore	added gitignore and rpm spec	3 years ago
ChangeLog	signed update	5 months ago
Dockerfile.arch	signed update	5 months ago
Dockerfile.ubuntu	fixed several issues around formating json and xml	7 months ago
LICENSE.txt	add LICENSE.txt	4 years ago
README.md	signed update	5 months ago
checksec	Move PATH check up so that sysctl works for for non-root users.	last month
checksec.sig	signed update	5 months ago
checksec_automator.sh	signed update	5 months ago
docker-compose.yml	signed update	5 months ago

README.md

checksec

Checksec is a bash script to check the properties of executables (like PIE, RELRO, PaX, Canaries, ASLR, Fortify Source). It has been originally written by Tobias Klein and the original source is available here: http://www.trapkit.de/tools/checksec.html

Updates

** MAJOR UPDATES ** 2.1.0

- Changed structure to be more modular and switched to getopts so options can be in any order. e.g. format=json can be at the end now, however.
- All options now require `--$option=$value` instead of `--$option $value`
- --extended option now includes clang CFI and safe stack checks

Last Update: 2019-07-29

For OSX

Most of the tools do not work on mach-O binaries or the OSX kernel, so it is not supported

Examples

normal (or --format=cli)

```
$checksec --file=/bin/ls
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   /bin/ls
```

csv

```
$ checksec --output=csv --file=/bin/ls
Partial RELRO,Canary found,NX enabled,No PIE,No RPATH,No RUNPATH,/bin/ls
```

xml

```
$ checksec --output=xml --file=/bin/ls
<?xml version="1.0" encoding="UTF-8"?>
<file relro="partial" canary="yes" nx="yes" pie="no" rpath="no" runpath="no" filename='/bin/ls'/>
```

json

```
$ checksec --output=json --file=/bin/ls
{ "file": {
"relro":"partial","canary":"yes","nx":"yes","pie":"no","rpath":"no","runpath":"no","filename":"/bin/ls"
} }
```

Fortify test in cli

```
$ checksec --fortify-proc=1
* Process name (PID)                       : init (1)
* FORTIFY_SOURCE support available (libc)    : Yes
* Binary compiled with FORTIFY_SOURCE support: Yes

------ EXECUTABLE-FILE ------- . -------- LIBC --------
FORTIFY-able library functions | Checked function names
-------------------------------------------------------
fdelt_chk                      | __fdelt_chk
read                           | __read_chk
syslog_chk                     | __syslog_chk
fprintf_chk                    | __fprintf_chk
vsnprintf_chk                  | __vsnprintf_chk
fgets                          | __fgets_chk
strncpy                        | __strncpy_chk
snprintf_chk                   | __snprintf_chk
memset                         | __memset_chk
strncat_chk                    | __strncat_chk
memcpy                         | __memcpy_chk
fread                          | __fread_chk
sprintf_chk                    | __sprintf_chk

SUMMARY:

* Number of checked functions in libc                : 78
* Total number of library functions in the executable: 116
* Number of FORTIFY-able functions in the executable : 13
* Number of checked functions in the executable      : 7
* Number of unchecked functions in the executable    : 6
```

Kernel test in Cli

```
$ checksec --kernel
* Kernel protection information:

Description - List the status of kernel protection mechanisms. Rather than
inspect kernel mechanisms that may aid in the prevention of exploitation of
userspace processes, this option lists the status of kernel configuration
options that harden the kernel itself against attack.
```

```
Kernel config: /proc/config.gz

       GCC stack protector support:            Enabled
       Strict user copy checks:                Disabled
       Enforce read-only kernel data:          Disabled
       Restrict /dev/mem access:               Enabled
       Restrict /dev/kmem access:              Enabled

* grsecurity / PaX: Auto GRKERNSEC

       Non-executable kernel pages:            Enabled
       Non-executable pages:                   Enabled
       Paging Based Non-executable pages:      Enabled
       Restrict MPROTECT:                      Enabled
       Address Space Layout Randomization:     Enabled
       Randomize Kernel Stack:                 Enabled
       Randomize User Stack:                   Enabled
       Randomize MMAP Stack:                   Enabled
       Sanitize freed memory:                  Enabled
       Sanitize Kernel Stack:                  Enabled
       Prevent userspace pointer deref:        Enabled
       Prevent kobject refcount overflow:      Enabled
       Bounds check heap object copies:        Enabled
       JIT Hardening:                                Enabled
       Thread Stack Random Gaps:                   Enabled
       Disable writing to kmem/mem/port:       Enabled
       Disable privileged I/O:                 Enabled
       Harden module auto-loading:             Enabled
       Chroot Protection:                            Enabled
       Deter ptrace process snooping:                Enabled
       Larger Entropy Pools:                   Enabled
       TCP/UDP Blackhole:                      Enabled
       Deter Exploit Bruteforcing:             Enabled
       Hide kernel symbols:                    Enabled

* Kernel Heap Hardening: No KERNHEAP

The KERNHEAP hardening patchset is available here:
 https://www.subreption.com/kernheap/
```

Kernel Test in XML

```
$ checksec --output=xml --kernel
<?xml version="1.0" encoding="UTF-8"?>
<kernel config='/boot/config-3.11-2-amd64' gcc_stack_protector='yes' strict_user_copy_check='no'
ro_kernel_data='yes' restrict_dev_mem_access='yes' restrict_dev_kmem_access='no'>
       <grsecurity config='no' />
       <kernheap config='no' />
</kernel>
```

Kernel Test in Json

```
$ checksec --output=json --kernel
{ "kernel": { "KernelConfig":"/boot/config-3.11-2-
amd64","gcc_stack_protector":"yes","strict_user_copy_check":"no","ro_kernel_data":"yes","restrict_dev_mem_
},{ "grsecurity_config":"no" },{ "kernheap_config":"no" } }
```

Using with Cross-compiled Systems

The checksec tool can be used against cross-compiled target file-systems offline. Key limitations to note:

- Kernel tests - require you to execute the script on the running system you'd like to check as they directly access kernel resources to identify system configuration/state. You can specify the config file for the kernel after the -k option.
- File check - the offline testing works for all the checks but the Fortify feature. Fortify, uses the running system's libraries vs those in the offline file-system. There are ways to workaround this (chroot) but at the moment, the ideal configuration would have this script executing on the running system when checking the files.

The checksec tool's normal use case is for runtime checking of the systems configruation. If the system is an embedded target, the native binutils tools like readelf may not be present. This would restrict which parts of the script will work.

Even with those limitations, the amount of valuable information this script provides, still makes it a valuable tool for checking offline file-systems.