
# **Papillon** - A Solaris Security Module Documentation

Version 0.4.3

(c) 2000-2002 Konrad Rieck (kr@roqe.org)

September 3, 2002

*I said to the Sun, "Good morning, Sun!*
*Rise and shine today!"* - Dean Martin

[0]The Solaris Logo is a trademark or a registered trademark of Sun Microsystems, Inc.

# Contents

# 1 Introduction

## 1.1 Overview

Welcome to the mysterious world of Papillon. This documentation covers all information regarding the functionality, the usage and the installation of the Papillon module. You should read this documentation *carefully* since only a properly installed and configured module will provide the security mechanisms introduced in this document.

Papillon is a security module designed for the Solaris Operating Environment (Solaris OE) 8 [SOE]. It has been tested against the Intel and the Sparc Edition of the Solaris OE 8. Papillon tries to be as compatible with Sun Microsystems DDI/DDK as possible, and should also work on the Solaris OE 9 beta and following.

## 1.2 What it does

Papillon improves the security of a system by adding new functionality to the kernel. The added security mechanisms have been inspired by Solar Designer's Openwall Linux Kernel Patch [OW] and the HAP Linux Kernel Patch [HAP] which fixes common Unix security problems that are also present in the Solaris OE.

Papillon is designed to prevent attacks driven by system users. It doesn't include any restriction to the super-user. It can be an addition to already exisiting security mechanisms such as the BSM (Solaris' Basic Security Module) and the non-executable stack on Solaris Sparc Edition.

The module is automatically loaded at boot time when entering multiuser level (`init 2`) and installs two kinds of new functionality in the kernel: so called *features* and *protections*.

**Features** Features add completely new functionality to the kernel, they can be switched *on* or *off* either at compilation time or even at runtime using the provided control tool `papctl`.

Features included in Papillon currently are:

- Restricted Proc
- Pseudo Promiscuous Flag
- Module Hiding
- Secure STDIO File Descriptors

**Protections** Protections restrict access to resources if specific conditions occur. A protection has a behaviour that can be *none* (for doing nothing), *warn* (for warning only) or *deny* (for warning and denying access to the resource).

Protections included in Papillon currently are:

- Symbolic Link Protection
- Hard Link Protection

- FIFO Protection
- Chroot Protection

See the section about features and protections for detailed information about how they work and what they do.

## 1.3 Release Notes

- **Version** `0.4.3`
  Added support for compilation on 64 bit system using the GNU C Compiler version 3.x.

  Fixed wrong path to linker and adjusted compilation flags for generic compilation, thanks to Adam Morley.

  Fixed missing memset() symbol when using broken compiler setup and header mixture, thanks to Erik Parker for reporting.

  This version is considered to be stable, the Beta flag has been removed from the version number.

- **Version** `0.4.2b`
  Added the ability to give read permission to the /proc to a group by using the definition `SUSER_GID`.

  Fixed the warning output, module claimed to deny access even though only warning mode was enabled.

- **Version** `0.4.1b`
  The compilation process automatically detects system's bit width and compiles the 64 bit module only on systems running a 64 bit kernel.

  The Sparc package now contains both, the 32 and the 64 bit module.

  Improved small stability and performance issues.

  The documentation is now available in the *Portable Document Format*, thanks to Markus C. Gottwald and Manuel Beetz for support.

  The module has been running stable for over 3 months now, therefore this and future releases are now labeled Beta instead of Alpha.

- **Version** `0.4.0a`
  The chroot protection has been recoded inspired by the newly released HAP Linux Kernel Patch. The following syscalls are restricted in a chroot environment: `chroot()`, `mount()`, `mknod()`, `xmknod()`, `modctl()` and `chmod()`. See last section of this document for a detailed description.

  Reorganized the syscall interception architecture, `syscalls[]` holds the systems default syscalls depending on the bit width while `syscalls32[]` holds the 32 bit syscalls on 64 bit systems.

  Fixed a typo in the `papctl` tool that caused incorrect help output.

  Fixed an incorrect string buffer that caused the pseudo promiscuous flag not to work on 64 bit systems.

  Fixed a 64 bit calculation problem in the `readdir()` routine. File hiding is now supported on all types of Solaris kernel (32 bit and 64 bit).

- **Version** `0.3.5a`
  Heiko Krupp contributed an initial implementation for a chroot protection that is based on matching '..' and '/' ocurrences in path string provided to the `chroot()` syscall.

- **Version** `0.3.4a`
  Fixed a security problem in the communication with `papctl`. An attacker would have been able to change the configuration with a patched version of `papctl`.

  Enabled most of the module and file hiding mechanisms on 64 bit systems. Only the readdir() interception which hides the module's file from directory listings is disabled until the code has been properly ported to support 64 bit dirent stuff.

- **Version** `0.3.3a`
  This is the first public release of Papillon. All features and protections have been tested on Intel and Sparc 32 bit systems. The file hiding has been disabled on UltraSparc 64 bit

systems, all other features and protections have been successfully tested.

# 2 Installation and Configuration

## 2.1 Getting Papillon

Papillon is available from the Roqefellaz website [RQ] You can directly access the Papillon page [PAP] at `http://www.roqe.org/papillon`.

After receiving the source package of Papillon `papillon-0.4.3 .tar.gz`, extract the sources and enter the source directory.

```
# zcat papillon-0.4.3 .tar.gz | tar -xvf -
# cd papillon-0.4.3
```

You should always download the latest version of Papillon. Kernel development is very critical, small bugs can cause real damage. See the section about bug reports for more information.

## 2.2 Requirements

In order to compile Papillon you need some general development environment.

- `make`
  You need a command that automates the compilation process. You can use the Solaris make `/usr/ccs/bin/make` from the `SUNWsprot` package or install GNU make that is part of the Solaris Software Companion CD [SCD] or available at SunFreeware [SFW].

- `cc` or `gcc`
  In order to compile the module, you need a working C compiler. If you are compiling the module for a 32 bit system such as any Intel Edition or any non-UltraSparc system, you can use both the Sun C Compiler [FC] which is part of the `SUNWspro` / Forte C package or the GNU C Compiler also available on the Solaris Software Companion CD [SCD] or at SunFreeware [SFW].

  **Note:** If you receive the following message *no* C compiler is installed on your system. You need to get one of the compilers mentioned above.
  `/usr/ucb/cc: language optional software package not installed`

  **Note:** If you are compiling the module for a 64 bit system, you have to use the Sun C Compiler [FC] or the GNU C Compiler version 3.x. The GNU C Compiler version 2.9.x is not able to build proper 64 bit objects.
  You can check whether your are running a 32 bit or 64 bit sytem, by executing the following command: `/usr/bin/isainfo -b`

- `ld`
  A linker is also necessary to compile papillon. You can use the default linker `/usr/ccs/bin/ld` from the `SUNWtoo` package or the GNU linker which is *not* part of the Solaris Software Companion CD but SunFreeware [SFW].

## 2.3 Compilation

Precompilation configuration is done in three files of the source tree: `src/Makefile`, `src/papillon.h` and `src/papillon.c`. The first is designed for common configurations while the latters are designed for advanced configurations.

### 2.3.1 Configuration `src/Makefile`

You need to decide which features and protections to compile into the module. By default all features and protections are included. Edit the file `src/Makefile` and change the following variables if necessary.

- `SYSCONFDIR=/etc`
  You should not change your system configuration directory unless you store configuration and boot scripts in another directory, which is very untypical.

- `SBINDIR=/usr/sbin`
  This is the location where the control tool `papctl` will be installed. You may change this to any path as long as the Papillon module and the command stay on the same type of filesystem. Otherwise the `papctl` command cannot be hidden.

- `KERNELDIR=/usr/kernel/misc`
  This is the place where the Papillon module will be installed. There is no need to change this unless you know what you are doing. Loading of the module using `modload` should *always* be performed using an absolute path.

- `FEATURES=-DRSTPROC -DSECSTDFD -DPPROMISC -DMODHIDING`
  By changing the values of this variable you can exclude features. To exclude a feature, remove its definition from the `FEATURES` variable. Excluded features are not compiled into the module, they cannot be enabled at later time without recompiling the module.

| **Definition** | **Feature** |
|---|---|
| `-DRSRPROC` | Restricted Proc |
| `-DSECSTDFD` | Secure STDIO File Descriptors |
| `-DPPROMISC` | Pseudo Promiscuous Flag |
| `-DMODHIDING` | Module Hiding |

- `PROTECTIONS=-DSYMPROT -DFIFOPROT -DHARDPROT`
  By changing the values of this variable you can exclude protections. To exclude a protection, remove its definition from the `PROTECTIONS` variable. As with the features, excluded protection can only be included through recompilation.

| **Definition** | **Protection** |
|---|---|
| `-DSYMPROT` | Symbolic Link Protection |
| `-DFIFOPROT` | FIFO Protection |
| `-DHARDPROT` | Hardlink Protection |
| `-DCHROOTPROT` | Chroot Protection |

- `CC=cc`
  This is the definition for the compiler used to compile the Papillon module. Papillon can be compiled with the Sun C Compiler `cc` as with the GNU C Compiler `gcc`. Depending on the choice of the compiler you need to adjust the compiler options `CFLAGS32` and `CFLAGS64`. See the comments in the Makefile.

  > **Note:** If you want to compile Papillon for a 64 bit environment (Solaris OE on UltraSparc) you have to use the Sun C Compiler or the GNU C Compiler version 3.x.

- `COPTS=-DSVR4 -DSOL2`
  This line describes options send to the C Compiler. For debug support add the definition `-DDEBUG`. If you are using the GNU C Compiler, you may also add `-Wall` to see more warning messages.

You can also modify other variables in the file `src/Makefile` but in general everything should work on a default Solaris OE installation.

### 2.3.2 Configuration `src/papillon.h`

If you are an advanced user and have some experience with kernel modules, you can also edit other files inside the `src` directory. The following changes can be done in `src/papillon.h`

- Changing the super-group
  If you want to allow a group of users read access to the restricted proc, change the definition `SUSER_GID` to an existing unix group. By default read access is granted to the super-user group `GID 0`.

- Changing the super-user
  By default Papillon assumes that the super-user is using the `UID 0`. If for some reason you want to change this and also restrict `UID 0`, change the definition `SUSER_UID` to a different user id.

- Changing the communication syscall
  Papillon uses an unused syscall for communication. `papctl` uses this syscall to export and import the configuration of Papillon from userspace to kernelspace and vice versa.

  The syscall number is defined by `SYS_papcomm`. If you are sure that this syscall is used on your system, e.g. by a third party software, change the value to another unused system call. You can retrieve a list of all used system calls by examining the system header file `/usr/include/sys/systm.h`.

### 2.3.3 Configuration `src/papillon.c`

- Modifing the default runtime configuration
  When the Papillon module is loaded it activates a compiled-in runtime configuration. This configuration can be changed using the command `papctl`.

  You may change the default runtime configuration in `src/papillon.c`. Read the section about runtime configuration and adjust the values in the `pap_config_t config` struct.

- Adding files to hide
  The struct `pap_modfiles_t modfiles[]` holds the files to be hidden. If you want to add a file, e.g. `/usr/bin/foobar`, extend the struct by adding the following line *before* the `{ NULL, NULL, NULL }` line:

  `{ "/usr/bin/foobar", NULL, NULL },`

  You can only hide files on the same filesystem type where the module itself resides.

### 2.3.4 Compilation

Compilation is rather simple and straight-forward

```
# cd src
# make
# cd ..
```

### 2.3.5 Testing the module

Before you permanently install Papillon, it is wise to run some tests. Load the module into the kernel by executing the following command.

```
# modload ./src/papillon
```

Run the control tool `papctl` to check if the module has been loaded successfully and the configuration gets correctly exported.

```
# ./src/papctl -g
Current configuration of the Papillon module:

- Features
Restricted Proc:  on
Pseudo Promiscuous Flag:  on
Module Hiding:  on
Secure STDIO File Descriptors:  on

- Protections
Symlink Protection:  deny
Hardlink Protection:  deny
Fifo Protection:  deny
Chroot Protection:  deny
```

Now compile the test suite that is part of the Papillon source package.

```
# cd test
# make
```

Stay in the `test` directory. Execute the `test.sh` shell script. Make sure no one is on your system. Follow the instructions printed by `test.sh`.

```
# ./test.sh
This script will check if Papillon is running and all enabled
protections are working.

WARNING: In order to check for possible attacks, it is necessary
to create a vulnerable environment in /tmp/fake.  Before
proceeding, check that your machine is running in single
user mode.

Continue (y/n):  y

- Checking for a restricted proc...  Yes
- Checking for hardlink attack protection...  Yes
- Checking for symlink attack protection...  Yes
- Checking for fifo attack protection...  Yes
- Checking for fifo64 attack protection...  Yes
- Checking for STDIO attack protection...  Yes
- Checking for STDIO64 attack protection...  Yes
- Checking for chroot protection...  Yes
- Checking for chroot64 protection...  Yes

Done.
```

Make the module visible and unload it. Use `modinfo` to determin the module id of Papillon and replace `ID` in the last line with this number.

```
# cd ..
# modinfo
# ./src/papctl -s m=off
# modunload -i ID
```

If during the process described above the system panics or any other major problems occur, send a bug report to `kr@roqe.org` and describe in detail your system, your configuration and the problem that occured. See the section bug reports for more information.

### 2.3.6 Installation

From the `src` directory you are able to install the module.

```
# cd src
# make install
# cd ..
```

These commands will create the following files on your system. (If you have modified some of the path variables in the file `src/Makefile` , paths may differ)

- `/usr/kernel/misc/papillon`
  The kernel module
- `/usr/sbin/papctl`
  The control tool
- `/etc/init.d/papillon`
  A script that loads papillon at boot time
- `/etc/rc2.d/S06papillon`
  A hardlink to the script `etc/init.d/papillon` which loads papillon in init level 2 (Multi-User).

You can use the following sequence to uninstall the installed files.

```
# cd src
# make uninstall
# cd ..
```

> **Note:** Papillon adds a script to the init directory so that the module is loaded at boot time. If you have *volume management* enabled problems concerning setting the configuration using `papctl` may occur. Disable the volume management by moving the `/etc/rc2.d/S92volmgt` script to a save place. If you don't want to disable volume management, read the part about how to cope with the problems in the runtime configuration section.

If all of these files have been installed on your system, you can activate the module by running

```
# /etc/init.d/papillon start
```

If you reboot your system, Papillon will automatically be loaded when switching to multiuser level.

# 3 Runtime Configuration

If Papillon is loaded, you can use the control tool `papctl` to toggle features and protections. Below is a list of the commandline and some examples.

> **Note:** If Papillon is loaded and hidden, you are not able to view it on the list of loaded modules generated by `modinfo`. The control tool `papctl` is the only way to test if the module is loaded or not in this case.

## 3.1 `papctl` Options

Usage:

```
papctl -s variable=value [variable=value ...]
papctl g | -v | -h
```

Options:

| | |
|---|---|
| `-g` | get current configuration of the loaded module. |
| `-s variable=value ...` | set current configuration of the loaded module. |
| `-h` | print this help. |
| `-v` | print version information. |

In order to toggle features or protections you have to assign variables the correspoding values. This is the table of all variables, their values and their description.

| Variable | Description | Possible values |
|---|---|---|
| `r` | Restricted Proc | `on, off` |
| `p` | Pseudo Promiscuous Flag | `on, off` |
| `m` | Module Hiding | `on, off` |
| `i` | Secure STDIO File Descriptor | `on, off` |
| `s` | Symbolic Link Protection | `none, warn, deny` |
| `h` | Hardlink Protection | `none, warn, deny` |
| `f` | Fifo Protection | `none, warn, deny` |
| `c` | Chroot Protection | `none, warn, deny` |

## 3.2 Examples

1. You have loaded the module and want to turn off the restricted proc while setting the symbolic link protection to warn only mode.

```
# papctl -s r=off s=warn
```

2. You want to disable the complete module but not unload it.

```
# papctl -s r=off p=off m=on i=off s=none h=none f=none
```

3. You want to enable all features of Papillon and leave the protection configuration untouched.

```
# papctr -s r=on p=on m=on i=on
```

## 3.3 Known problems

- Volume management locks `papctl`
  There is a known problem that causes `papctl` to fail if the volume management has been loaded after the Papillon module. You can detect this problem if `papctl` outputs the following error message:

  ```
  # papctl -s f=none
  Error:  ##
  Configuration blocked.  Volmgt running?
  ```

  You can fix this problem by turning the volume management off, configuring the module, and then reactivating the volume management.

  ```
  # /etc/init.d/volmgt stop
  # papctl -s f=none
  # /etc/init.d/volmgt start
  ```

- Volume management locks unloading
  If Papillon reports `device busy` while unloading, there is probably a locking conflict with the volume management. Turning the volumne management off, unloading the module and reactivating the volume management should fix this problem.

- `No such file or directory` message when loading Papillon
  The Solaris tool `modload` that is used to load kernel modules into kernel space outputs the following error message if *any* error occurs: `No such file or directory`. This message doesn't necessary indicate a missing file, but may also be caused by a missing symbol, a broken binary, etc.

  In order to get more information, it is necessary to log more detailed error messages from `modload` using the `syslog` facility `kern.info`.

# 4 Papillon's functionality

This section lists all features and protections that are included in the Papillon module. You should carefully read this section in order to understand how Papillon works and how it achieves an improvement in the system security.

## 4.1 Features

As mentioned in the introduction the so called *features* of Papillon can be switched *on* or *off* either on compilation or even on runtime. See the sections about compilation and runtime configuration how to do both.

### 4.1.1 Restricted Proc

By default users on the Solaris OE are able to monitor all active processes (e.g. `ps`, `top`). An attacker that has local access to the system might gather useful information by watching system daemons and other users processes. The public information about all running processes also represent a lack of privacy, if a system hosts several independant users.

If the Restricted Proc feature is active, users are only able to view their own processes (processes that are running under their user id `uid`). There is no possibilty for a user to monitor other users processes since Papillon effects the `proc` filesystem directly. An attacker which has a local account on the system gains no further information from running commands such as `ps` or `top`. Of cause the super-user is able to view all processes. A special group of users can be added that are also able to view inside the restricted proc.

Papillon extends the `access()` function of the proc vnode operations provided by `prvnodeops` to implement the above feature. Unfortunatley the Solaris OE sets the correct permission on the files inside the `proc` filesystem but does not implement an `access()` in the proc kernel module. Papillon simply adds this missing `access()` function.

### 4.1.2 Pseudo Promiscuous Flag

The Solaris OE 8 and previous versions don't provide a promiscuous mode flag for network cards that is exported to the user. An administrator is not able to monitor a network device for an attacker that is sniffing.

Papillon is able to log all attempts to turn a network device into promiscuous mode that are done using the DPLI Interface. Requests that are performed using a different approach are not detected. Most sniffers use the DLPI Interface.

Papillon intercepts the `putmsg()` syscall and filters messages that match a DLPI requests (`dl_primitive == DL_PROMISCON_REQ` and `dl_level == DL_PROMISC_PHYS`). If a message contains the command for putting a network device into promiscuous mode a warning is send to the syslog. The module is not able to log when a network interface changes back from promiscuous mode to normal operation mode.

### 4.1.3 Module Hiding

In most cases it is not necessary to hide a security module. But if an administrator wants to monitor an exisiting attacker, it might be necessary to make the attacker believe that this system is not protected by any security software.

Papillon is able to remove itself from the list of loaded kernel modules. Papillon denies any access the module's files and hides the module's files from directory listings including the module itself, initscripts and the `papctl` control program. The super-user is able to view and access all of these files. The list of files to be hidden can be extended at compilation time.

Papillon unlinks itself from the list of loaded modules and relinks itself back in if requested. It also intercepts the `vop_lookup()` function from his module's file vnode. Access to the file is denied if not the super-user accesses the file. Papillon also intercepts the `vop_readdir()` function of its parent directory and removes itself from all directory listings by patching the length of previous `dirent64` entry using `d_len`. The entry for Papillon is covered this way.

### 4.1.4 Secure STDIO File Descriptors

By default Unix uses the file descriptors 0, 1 and 2 for special purposes.

- File descriptor 0 represents the Standard Input Stream `STDIN`
- File descriptor 1 represents the Standard Output Stream `STDOUT`
- File descriptor 2 represents the Standard Error Stream `STDERR`

If an attacker closes one of these file descriptors and executes an insecure program with the suid/sgid bit (permission mode 4000/2000 or `u+s/g+s`) set, a file descriptor inside the program might be assigned to one of the closed standard file descriptors. In this case information written to `STDIN`, `STDOUT` or `STDERR` might be written to a file. By using this technique an attacker is able to destroy or even modify system files.

Papillon intercepts the execution of all binaries that have the suid/sgid bit set. If the STDIO File Descriptors are closed, Papillon fake opens them during the execution of the suid/sgid program. No suid/sgid program is able to accidently assign a file to the STDIO File Descriptors.

Papillon intercepts the `execve()` syscall and watches binaries with the suid/sgid bit (Vnode mode `S_ISUID` or `S_ISGID`) set. If the Standard File Descriptors are closed, they are faked opened using the kernel allocation routine `ualloc()`. After executing the original syscall the allocated file descriptors are set free.

## 4.2 Protections

Protections restrict access to resources (e.g. opening file) if specific conditions occur (e.g. the file is located in a directory with the sticky bit set). A protection has a behaviour that can be *none* (for doing nothing), *warn* (for warning only) or *deny* (for warning and denying access to the resource). You can customize the default behaviour of your protections at compilation time. You are also able to change the behaviour on runtime.

### 4.2.1 Symbolic Link Protection

Directories with the sticky bit (permission mode `+t` or `1000`) and write-all (Permission mode `a+w` or `0222`) permissions have a specific behaviour: files created in such a directory can only be removed by the file owner or the super-user eventhough write permissions are granted to all users, e.g. `/tmp`. An attacker can use this feature to drive a symbolic link attack. A symbolic link attack is basicly based on a symbolic link that has the name of a temporary file and links to a file the attacker wants to modify.

Papillon provides a simple symbolic link protection. If a user wants to follow a symbolic link that is within a directory with the sticky bit set, access is denied if all of the following conditions are true:

- The parent directory of the symbolic link has the sticky bit set
- The symbolic link is not owned by the user who wants to follow it
- The parent directory of the symbolic link has a different owner than the symbolic link itself

This protection mechanism especially protects the super-user privileges, an attacker is not able to gain super-user privileges, if the admin executes a binary that creates insecure temporary files.

Papillon watches all calls to the `open()` and `open64()` syscalls, if a symbolic link is to be opened that is placed in a directory with the sticky bit (Vnode mode `S_ISVTX`) set and the above conditions match, the open fails with permission denied (`EPERM`).

### 4.2.2 Hard Link Protection

An attacker can perform most symbolic link attacks by using hard links. If the symbolic links are protected, it is likely that hard links will be used in exploits. There is also another problem with hard links in the Solaris OE. Users are able to create hard links, that they cannot delete afterwards, e.g. by hard linking to `/etc/password` .

Papillon fixes both problems. If the hard link protection is active users can not create hard links to files they donnot own. The super-user is able to create hard links to all files.

The `link()` syscall is intercepted and the above protections are implemented. Papillon returns permission denied (`EPERM`).

### 4.2.3 FIFO Protection

A FIFO (e.g. a file created with `mkfifo`) that is inside a directory with the sticky bit set and write-all permissions can be opened by an attacker using the open flag `O_CREAT`. In this case all saved content inside the FIFO will be lost.

Papillon restricts access to FIFOs inside directories with the sticky bit set and write-all permissions. Open access to FIFOs is denied if all of the following conditions are true:

- The parent directory of the FIFO has the sticky bit set
- The FIFO will be opened with the `O_CREAT` flag
- The FIFO is not owned by the user who wants to open it

- The user is not the super-user
- The parent directory of the FIFO has a different owner than FIFO itself

Papillon watches all calls to the `open()` and `open64()` syscalls, if a FIFO is to be opened with the `O_CREAT` flag in a directory with the sticky bit (Vnode mode `S_ISVTX`) set, access is denied if the above conditions match. In this case Papillon returns permission denied (`EPERM`).

### 4.2.4 Chroot Protection

The `chroot()` syscall is often used to create another security layer between an application and the operating systems, but it has been initially designed as a safe (not secure) development environment. An attacker that gained root in a *chroot jail* will soon focus on breaking out of the jail. Amoung the rich set of possibities Papillon protects against the most common techniques used.

- An attacker could try to change the chroot root node by calling `chroot()` inside a chroot environment.
- He could try to mount a filesystem outside the chroot environment into it.
- He could create block or char nodes that correspond to peripheries outside the chroot jail, e.g. disks, ttys.
- He could try to load a module into the kernel and manually change the chroot root node.
- He could try to set permissions on executable files inside the chroot environment to suid or sgid.

Papillon prevents these attacks if a process runs in a chroot jail. Therefore the module intercepts the following syscalls and restricts access to them if the running process has the chroot vnode set (`u.u_rdir != NULL`): `chroot()`, `mount()`, `mknod`, `xmknod`, `modctl` and `chmod`. Inside the intercepted syscalls Papillon checks if one of the conditions mentioned above has occured and if positive forces the syscall to return permission denied, error number `EPERM`.

# 5 Closing

## 5.1 Bug Reports

Papillon has been developed in the free time of the author. It is a non-commercial opensource project. Papillon tries to offer a maximum of stability, but due to the reasons mentioned above, it can fail to do so.

Therefore it is essential that users experiencing bugs report them to the author by sending an email to

**Konrad Rieck** <kr@roqe.org>

Include detailed information when, where and how the bug occured. If necessary add parts of log files such as syslog. If the system dumped core, *don't* send core files. Instead move to the crash directory /var/crash/<hostname>/ and execute:

```
echo \$c | mdb unix.0 vmcore.0
```

You should retrieve a backtrace of the kernel thread that paniced. Include this trace in the email instead of attaching any core file.

## 5.2 Thanks

The author would like to thank *Job de Haas* for his ideas, support and the fun during their presentation at HAL2001.

Thanks to *Fabian Kroenner* for hours of constructive discussions regardings Papillon and its future, there would have been no release without his support.

Thanks also go to *Heiko Krupp* who contributed the initial implementation of the chroot protection and is supporting the project with his own code and ideas.

*Konrad Kretschmer* was so gentle to read through the complete documentation eleminating at least the obvious misspellings and *Markus C. Gottwald* and *Manuel Beetz* who contributed parts of the documentation system.

The author also would like to thank *Philipp Stucke* and *Skyper* who provided some of the test environment.

Thanks to following persons who contributed beta test reports and/or feedback (in order of appearance): Sergei Rousakov, Michael Parkin, Adam Mazza, Adam Morley, Juri Haberland, Erik Parker and Eric Thern.

# References


[SOE] **Solaris Operating Environment**

The official Solaris Operating Environment site including several downloads information and patches at the Sun Microsystems, Inc. website.

http://www.sun.com/software/solaris

[SFW] **SunFreeware**

Large collection of precompiled packages from the open source community including the GNU Compiler Collection, GNU make and other usefull development utilities.

http://www.sunfreeware.com

[SCD] **Solaris Software Companion CD**

A compilation of opensource utilities released by Sun Microsystems, Inc. and shipped with some Solaris versions including the GNU Compiler Collection, GNU make, GNU emacs and a lot of other interesting packages.

http://www.sun.com/software/solaris/freeware

[FC] **Forte C**

Forte C has replaced the Sun Workshop and includes the Sun C Compiler necessary to compile 64 bit kernel modules. An evaluation version is available at the link below.

http://www.sun.com/forte/c

[OW] **Openwall Linux Kernel Patch**

Site of the Linux kernel patch Papillon has been inspired by. Most of the features relevant to Solaris OE have been adopted from this patch written by Solar Designer.

http://www.openwall.org/linux

[HAP] **HAP Linux Kernel Patch**

A Linux kernel patch that is installed upon the Openwall Linux patch. It adds even more security features to the Linux kernel and also influenced Papillon's features and protections especially the `chroot()` protection.

http://www.doutlets.com/downloadables/hap.phtml

[RQ] **The Roqefellaz**

A small website that has been initiated by the author and some of his friends to publish opensource projects they develop in their free time.

http://www.roqe.org

[PAP] **Papillon - Solaris Security Module**

The official website of the Papillon module. Updates and future versions can be obtained from this place.

http://www.roqe.org/papillon

[WDD] **Writing Device Drivers**

This book published by Sun was a guide when developing Papillon. It focuses on device drivers but also contains information concerning other types of kernel modules.

http://docs.sun.com

[SI] **Solaris Internals**

This excellent book by Jim Mauro and Richard McDougall focuses on the concrete and also on the theoretical parts of the Solaris kernel implementation.

http://www.solarisinternals.com