Papillon 0.5.1 – Solaris Security Module Documentation and Manual

Konrad Rieck (kr@roqe.org)

14th April 2003 – Revision: 1.69

Contents

1 Introduction

Papillon is a security module designed for the Solaris Operating Environment 8 and 9 [7]. It provides security mechanisms and protections that improve the overall security of the system by adding new functionality to the kernel. The security mechanisms and protections have been inspired by the Openwall [5] and the HAP [2] Linux kernel patches and address common Unix security issues that are also present in the Solaris Operating Environment.

Papillon follows the philosophy of *prevention through restriction*. By adding minimal restrictions to resources, such as symbolic links or FIFOs, compromises based on common attack techniques can be prevented without influence on the system's usability. Papillon is a great addition to already existing security solutions such as the Solaris Basic Security Module (BSM) and the non-executable stack on the Solaris SPARC Edition. All features and protections of the Papillon module can be en- and disabled at compilation time or even at runtime, therefore the functionality can be optimally adapted to a specific system.

Even though there have been several requests, the module is not designed to weaken the superuser's privileges and protect against intruders that already compromised the system. Restricting these privileges requires a special design of the operating system, e.g. in Trusted Solaris [12] or OpenBSD [4], and is nearly impossible to implement using plain loadable kernel modules.

The following documentation and manual is divided into two major parts. The first part introduces Papillon's functionality and its impact on the system's security in detail. The second part describes the compilation, installation and configuration of the Papillon module and its components. In order to correctly integrate the module into your system, it is essential to study both parts of this documentation, in order to learn about the provided functionality and its installation. Papillon is not one of those simple software packages, incorrect installation or misconfiguration may have an opposite effect on the security of the target system.

2 Papillon's functionality

For simplicity Papillon's functionality has been divided into functional units that act either as so called *features* or security *protections*.

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

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

2.1 Features

Following is a description of each feature implemented in the Papillon module 0.5.1 . For each feature the following part lists a short introduction to the addressed security issue and a description how the module fixes the problem. Additionally some implementation details are provided that are intended for developers that want to modify or extend the module's code.

As mentioned above features, loaded with the Papillon module into the Solaris kernel, can be switched *on* or *off* either at compilation time or later at runtime.

2.1.1 Restricted Proc

By default users in the Solaris Operating Environment are able to monitor all active processes (e.g. by using the programs `ps` or `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 represents a lack of privacy, if a system hosts several users.

If the Restricted Proc feature is enabled, users are only able to view own processes which are running under their user ID (UID). It is impossible for an attacker to monitor other users' processes because Papillon directly restricts access to the `/proc` file system which is the global source for all information about running processes.

In order to allow system's maintenance the super-user is able to view all processes. A special group can be added whose members are also able to view running processes when the Restricted Proc feature is enabled.

Papillon extends the `access()` function of the procfs vnode operations provided by `prvnodeops` in order to implement the above feature. The Solaris OE does set the correct permissions on the files inside the `/proc` file system but does not implement an `access()` function in the procfs kernel module. Papillon simply adds this missing `access()` function. The Restricted Proc feature has been inspired by the Openwall [5] Linux kernel patch and is also briefly discussed in the presentation [9] by Konrad Rieck and Job de Haas.

2.1.2 Pseudo Promiscuous Flag

The Solaris Operating Environment 9 and previous Solaris versions don't provide a promiscuous mode flag for network adapters that is exported to the user. An administrator is not able to monitor a network device for an attacker sniffing on the device.

Papillon is able to log all attempts to turn a network device into promiscuous mode that are done using the DLPI interface. Most sniffers, e.g. `snoop` or libpcap-based sniffers as `tcpdump`, use that interface to communicate with network adapters. Requests that are performed using a different approach are not detected. Below is an example syslog entry created by the Papillon module indicating that a network device has been put into promiscuous mode

```
Mar 26 20:16:37 fluffy papillon: WARNING: Promiscuous mode enabled on
interface hme (cmd: tcpdump, pid: 6179, uid: 0, gid: 1).
```

Papillon intercepts the `putmsg()` system call and filters messages that match DLPI *promiscuous on* requests. If such a message is detected a warning is send to the syslog. The module is not able to detect a network interface changing back from promiscuous mode to normal operation mode.

2.1.3 Secure STDIO File Descriptors

By default Unix uses the file descriptors 0, 1 and 2 for special purposes – the standard IO (STDIO) which includes in- and output of programs to and from the console. Typically these in- and outputs can be combined and filtered using so called pipes. Below is a list of the three STDIO file descriptors.

STDIN	File descriptor 0 is used for the standard input stream.
STDOUT	File descriptor 1 is used for the standard output stream.
STDERR	File descriptor 2 is used for the standard error stream.

If an attacker closes one of these file descriptors and executes an program with the setuid or setgid bit set, a file descriptor inside the program might be assigned to one of the closed STDIO 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 modify system files.

Papillon intercepts the execution of all binaries that have the setuid or setgid bit set. If one of the STDIO file descriptors is closed before executing such a program, Papillon fake opens the descriptor during the execution of the program. Therefore no program with the setuid or setgid bit set is able to accidentally assign a file to the STDIO file descriptors. Below is an example entry from the syslog that illustrates the fake opening of STDIO file descriptors.

```
Mar 26 20:25:47 fluffy papillon: WARNING: Fake opening STDERR before
executing /tmp/a (cmd: sh -c /tmp/a, pid: 6472, uid: 101, gid: 101).
```

Papillon intercepts the `execve()` system call and watches vnodes with the setuid or setgid bit set. If one of the STDIO file descriptors is closed before execution, it is faked opened using the kernel allocation routine `ualloc()` and unallocated after execution. The problem of closed STDIO file descriptors is discussed inside the Openwall [5] Linux kernel patch.

2.1.4 Module Hiding

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

Papillon is able to remove itself from the list of loaded kernel modules and can operate invisible. It also denies any access to the module's files and hides them from directory listings, including the module itself, init-scripts and the control program. The super-user is able to view and access all of these files. The list of files that are hidden can be extended at compilation time, but not at runtime, so that an administrator can add other files that are not visible to the system's users.

Papillon unlinks itself from the list of loaded modules and relinks itself back in if requested. The module intercepts the `vop_lookup()` and `vop_readdir()` functions from the root file system in order to hide files from direct access and directory listings. The file hiding mechanism is based on Job de Haas' kernel module [3]. Directory entries are removed by patching the length of previous `dirent64` entry using `d_len`.

2.2 Protections

Following is a description of each protection integrated in the Papillon module 0.5.1 . Similar to the previous section for each protection an addressed security issue is introduced and a detailed description of the protection given. Additionally some implementation details are provided that are intended for developers that want to modify or extend the module's code.

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

2.2.1 Symbolic Link Protection

Directories with the sticky bit (octal mode 1000) and write-all permissions have a specific behavior: files created in these directories can only be removed by the file owner or the super-user even though write permissions are granted to all users. A typical example for such a directory is the `/tmp` directory. An attacker can use this specific behavior to perform a symbolic link attack which is based on a symbolic link that redirects output from a temporary file.

Papillon provides a simple symbolic link protection based on the Openwall [5] Linux kernel patch. 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 parent directory of the symbolic link has a different owner than the symbolic link.
+ The symbolic link is not owned by the user who is accessing it.

Due to this protection an attacker is not able to increase his privileges if the super-user executes a binary that uses temporary files which are vulnerable to a symbolic link attack. Following is a syslog entry illustrating the output of Papillon if opening a symbolic link is denied by the module. Instead of denying access, the administrator can also set this protection to warning only mode.

```
Mar 26 20:25:47 fluffy papillon: WARNING: Denied following symlink
/tmp/a (cmd: cat /tmp/a, pid: 6448, uid: 0, gid: 1).
```

Papillon watches all calls to the `open()` and `open64()` system calls. If a symbolic link should be opened that is placed in a directory with the sticky bit and the above conditions match, the open fails with permission denied (`EPERM`).

2.2.2 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 future exploits. Therefore it is necessary to bundle a Hard Link Protection with the existing Symbolic Link Protection of Papillon.

There is also another common problem with hard links in the Solaris Operating Environment. Users are able to create hard links to file which they don't own. After the creation of such a hard link the user is not able to delete the created link.

Papillon fixes both problems. If the hard link protection is enabled users can not create hard links to files which they don't own. The super-user is able to create hard links to all files. Following is an example entry from the syslog.

```
Mar 26 20:25:47 fluffy papillon: WARNING: Denied creating hardlink
from a to b (cmd: ln a b, pid: 6443, uid: 101, gid: 101).
```

The `link()` system call is intercepted to implement above protection. The module returns permission denied (`EPERM`) if a user tries to create a hard link to a file which he doesn't own.

2.2.3 FIFO Protection

The so called FIFO special file is used to queue input using a first-in first-out algorithm, it can be created using the command `mkfifo`. If the sticky bit is set and write-all permissions are granted to a directory, an attacker is able open a FIFO special file inside this directory with the `O_CREAT` flag and clear all the content stored in the FIFO queue.

Papillon implements a FIFO Protection that prevents a FIFO special file from loosing its content if an attacker tries to open it and the following conditions occur. For simplicity the term FIFO is used instead of FIFO special file.

+ The parent directory of the FIFO has the sticky bit set.
+ The parent directory of the FIFO has a different owner than FIFO.
+ The `O_CREAT` flag is set in the opening mode.
+ The FIFO is not owned by the user who is accessing it.
+ The user is not the super-user.

By adding this restriction to the opening of FIFOs all attacks based on the removal of FIFO special file content can be prevented. Below is an example entry from the syslog that illustrates a denied open request for a FIFO special file.

```
Mar 26 20:25:47 fluffy papillon: WARNING: Denied opening FIFO
/tmp/a (cmd: ./fifoattack /tmp/a, pid: 6453, uid: 101, gid: 101).
```

Papillon watches all calls to the `open()` and `open64()` system calls, if a FIFO is to be opened with the `O_CREAT` flag in a directory with the sticky bit, access is denied if the above conditions match. In this case Papillon returns permission denied (`EPERM`). This protection is also based on the Openwall [5] Linux kernel patch.

2.2.4 Chroot Protection

The `chroot()` system call 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) installation environment. An attacker that gained super-user privileges in a chroot environment will focus on removing the chroot restrictions.

There are several methods for breaking out of a chroot environment, common techniques include: re-calling the `chroot()` system call to set a new file system root, mounting an outside resource into the chroot environment, creating block or character devices to outside resources from inside the chroot environment, loading malicious kernel modules and changing permissions on administration binaries inside the chroot environment.

Based on the HAP [2] Linux kernel patch Papillon prevents these attacks if a process runs inside a chroot environment. Several system calls will restrict access when called from a chroot environment, e.g. the system call for creating block or character devices. Below is an example entry from the syslog showing a denied device creation request inside a chroot environment.

```
Mar 26 20:25:48 fluffy papillon: WARNING: Denied creating device
node /dev_null chroot'ed (cmd: ./break, pid: 6481, uid: 0, gid: 1).
```

Papillon intercepts the following system calls and restricts access if the running process has the chroot vnode set (`u.u_rdir`): `chroot()`, `mount()`, `mknod()`, `xmknod()`, `modctl()` and `chmod()`. The system calls return `EPERM` called from inside a chroot environment. The initial version of this protection has been implemented by Heiko Krupp / MIP GmbH.

2.2.5 Setuid Execution Protection

A lot of vulnerabilities that allow a local attacker to change his privileges exploit bugs in setuid or setgid binaries. Usually the attacker executes a shell or another program from within the setuid or setgid binary to gain more privileges.

The Setuid Execution Protection monitors the execution of programs on the system and is activated whenever a program with the setuid or setgid bit executes a child program. The protection can be used to simply log the execution of these child programs or might also be used to deny any execution of child programs from within setuid or setgid programs (which might be too restrictive). An example output from the syslog is listed below.

```
Mar 26 19:01:31 fluffy papillon: WARNING: Executing /tmp/a by
setuid parent /tmp/b (cmd: /tmp/b, pid: 5039, uid: 101, gid: 10).
```

Papillon intercepts the `execve()` system call to monitor the execution of programs and their parent processes. The `p_exec` entry is used to retrieve the parent process' vnode.

3 Installation and Configuration

3.1 Installation

Papillon can be installed from source or binary packages which are available at the Papillon website [6] (http://www.roqe.org/papillon). In general it is recommended to compile and install the module from a source package, even though the installation of binary package has become a practical and common method. Compiling the Papillon module allows better code optimization and performance due to the target system's C compiler and corresponding compiler options, and also supports compilation time configuration which can be used to minimize the module's size and set default values to optimally adapt to the target system.

The following section describes the compilation and installation process including details on compilation time configuration. The section closes with a few instruction that check the functionality of the installed Papillon software module and its components.

3.1.1 Requirements

In order to compile Papillon you need some general development environment. In most cases all components have already been installed on your system. For each required component in the listing below an Internet reference is given that allows free download of the component or an equivalent software.

Make tool: `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 Companion Software [8] or available at Sunfreeware [11].

C Compiler: `cc` **or** `gcc`
You need a C compiler that supports the generation of 64 bit objects. You can use the Sun C Compiler [10] which is part of the `SUNWspro` package and bundled with the Sun ONE Studio 7 Compiler Collection or the GNU C Compiler [1] version 3.x or above available at Sunfreeware [11].

If you choose to use the GNU C Compiler, check that you are using *version 3.x or above* for the *correct* Solaris version. You can use the command `gcc --version` to retrieve both information.

Linker: `ld`
A linker is also necessary to link the compiled object files. 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 Companion Software but available at Sunfreeware [11].

3.1.2 Compilation Time Configuration

Before building the Papillon module from the source files inside the `src/` directory. You should configure the features and protections of the module to fit your needs. The following list shows files that contain configurable parts.

`src/Makefile`	Compilation details and pathnames
`src/papillon.h`	User and group IDs, communication system call
`src/papillon.c`	Default setting for features and protections

Configuration `src/Makefile`

You need to decide which features and protections to compile into the module and where to store the components of Papillon. 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 `papctl` program stay on the same type of file system.

`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.

`FEATURES=[...]`
By changing the values of this variable you can exclude features. To exclude a feature simply 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. Following is a list of all possible definitions and the feature they enable.

`-DRSTPROC`	Restricted Proc
`-DSECSTDFD`	Secure STDIO File Descriptors
`-DPPROMISC`	Pseudo Promiscuous Flag
`-DMODHIDING`	Module Hiding

`PROTECTIONS=[...]`
By changing the values of this variable you can exclude protections similar to the `FEATURE` variable. To exclude a protection simply remove its definition from the `PROTECTIONS` variable. As with the features, excluded protection can only be included through recompilation. Following is a list of all possible definitions and the protection they enable.

`-DSYMPROT`	Symbolic Link Protection
`-DFIFOPROT`	FIFO Protection
`-DHARDPROT`	Hardlink Protection
`-DCHROOTPROT`	Chroot Protection
`-DSEXECPROT`	Setuid Execution Protection

`CC=gcc`
Depending on your C compiler you have to change this variable and the `CFLAGS32` and `CFLAGS64` variables. There are uncommented settings for both C compilers in the Makefile. If you are using the GNU C Compiler, you can also add `-Wall` to the `COPTS` variable in order to get all warning messages during the compilation process.

You may also modify other variables in the file `src/Makefile` but in general everything should work without further modification on a default installation of the Solaris Operating Environment.

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 grant read-access to a group of users inside 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
  Papillon associates the `UID 0` with the super-user. 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 system call
  Papillon uses an unused system call for communication. `papctl` uses this system call

to export and import the configuration of Papillon from user space to kernel space and vice versa. The system call number is defined by `SYS_papcomm`. If you are sure that this system call 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`.

Configuration `src/papillon.c`

If you are really experienced with C source code, you can also configure some settings in the file `src/papillon.c`.

When the Papillon module is loaded it activates a default configuration, which can be changed at runtime using the tool `papctl`. To change this default configuration in `src/papillon.c`, modify the initial values of the `pap_config_t config` struct.

```
pap_config_t config = {
    /* rstproc, ppromisc, modhiding, secstdfd */
       ON,      ON,       OFF,       ON,
    /* fifoprot, symprot, hardprot, chrootprot, sexecprot */
       DENY,     DENY,    DENY,     DENY,       WARN
};
```

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 a new line before the the last triple NULL line. Note that you can only hide files on the same file system type.

```
pap_modfiles_t modfiles[] = {
    [...]
    { "/usr/bin/foobar", NULL, NULL },
    {NULL, NULL, NULL}
};
```

3.1.3 Compilation

The compilation process itself is rather simple and straight-forward and should build the module in a few seconds.

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

If an error occurs, check if you configured everything correctly as described in the previous section. If you configured parts inside the files `papillon.h` and `papillon.c` watch for typos, missing brackets and semicolons.

If you still cannot build the module, pipe the output of the make process into a file and send it to the author. See the section 4.2 for more information about how to create a bug report and where to send feedback.

3.1.4 Testing the build

Even though Papillon has been designed with stability and compatibility in mind, it is wise to run some functionality and stability tests before installing the module permanently. As the first step load the module into the running kernel by executing the following command.

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

If the execution of the program `modload` fails, it will report the message *"No such file or directory"* even though all files are in place. Don't get confused, consult the syslog for information about the failure, if necessary extend the syslog logging configuration to catch kernel error events. If no failure is reported, run the control tool `papctl` to check if the module has been loaded successfully and the configuration gets exported.

```
# ./src/papctl -g
Current configuration of the Papillon v0.5.0 module:
[...]
```

In order to allow functionality tests a test suite has been added to the Papillon source package that consists of several C sources and a shell script that helps generating the test environment. Compile the test suite by executing the commands below. Maybe it is necessary to make some changes in the corresponding Makefile `test/Makefile`.

```
# cd test
# make
```

Stay in the `test/` directory and execute the shell script `test.sh` that will build a test environment and launch several fake attacks to test the features and protections of the Papillon module. All of these fake attacks have been designed with security in mind and minimal privileges by using the *nobody user* as an attacker, nevertheless you should shutdown critical system services and disable logins during the execution of the test suite.

```
# ./test.sh
[...]
* General environment
- Checking for a restricted proc...           Yes
- Checking for hardlink attack protection...  Yes
- Checking for symlink attack protection...   Yes
[...]
Done.
```

The output of the test shell script may vary depending on the compilation time configuration you have made. If you enable all protections and features the output should show a *yes* at the end of each fake attack.

In order to complete the tests, make the module visible and unload it. Use `modinfo` to determine the module ID of Papillon and replace `ID` in the last line with this number.

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

If during the process described above the system panics or any other minor or major problems occur, please spend some time and create a bug report. See the section 4.2 for more information about how to create a bug report and where to send feedback.

Installation

In order to install the Papillon kernel module and its components execute the following sequence of programs that will install and launch Papillon. If you have skipped the previous section that introduced the test suite for Papillon, go back and perform the tests in order to guarantee system's stability and Papillon's functionality.

```
# cd src
# make install
# /etc/init.d/papillon start
```

These commands will create the following files on your system and enable Papillon automatically the next time you reboot your system. If you have modified some of the path variables in the file `src/Makefile`, pathnames may differ.

`/usr/kernel/misc/papillon`	The 32 bit kernel module
`/usr/kernel/misc/sparcv9/papillon`	The 64 bit kernel module
`/usr/sbin/papctl`	The control tool
`/etc/init.d/papillon`	The init script to load papillon
`/etc/rc*.d/*papillon`	The hard links to the init script

If you don't like Papillon, you can use the following sequence of programs to uninstall the module and the corresponding files. Note that is necessary to unload the module from the kernel before removing the file when module hiding is activated.

```
# make uninstall
```

If you don't like Papillon, why not drop a note to the author about the things you dislike? See section 4.2 for contact information.

3.2 Runtime Configuration

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

Note that if the Papillon module 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.

3.2.1 Control Tool Options

This section covers the command line options of the control tool `papctl` which is available with the Papillon module and used to communicate from user land with the kernel module.

Usage:

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

Options:

`-g`	get current configuration of the loaded module.
`-s variable=value ...`	set current configuration of the loaded module.
`-f`	force setting the current configuration.
`-h`	print this help.
`-V`	print version information.

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

Variable	Feature 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`

Variable	Protection Description	Possible values
`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`
`x`	Setuid Execution Protection	`none, warn, deny`

3.2.2 Command Line Examples

1. One of the most common situations is the manual unloading of the module. In general this can be done by using the init script, but for completeness the following example demonstrates how to turn of the module hiding and then unload the module.

```
# papctl -s m=off
# ID=‘modinfo | grep papillon | cut -d1 -f" "‘
# modunload -i $ID
```

2. If you want to disable the complete module but not unload it, you have to disable all features and protections except module hiding. The module will then simply idle and wait to be activated again. The following example illustrates the long command line for this purpose. Note that the module is not visible in this example (`m=on`).

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

3. If you want to view the current configuration and then enable all features of Papillon and leave the protections' configuration untouched, execute the following sequence of commands.

```
# papctl -g
# papctl -s r=on p=on m=on i=on
```

4. This last example demonstrates some of the fake attacks that are performed in the Papillon test suite. If you have enabled the corresponding features and protections you should notice the enhanced security.

```
# touch /tmp/a
# su nobody -c "ln -s /tmp/a /tmp/b"

# /etc/init.d/papillon stop
# su nobody -c "ln /tmp/a /tmp/c"
# su nobody -c "ps -e"
# cat /tmp/b
# rm /tmp/c

# /etc/init./papillon start
# su nobody -c "ln /tmp/a /tmp/c"
# su nobody -c "ps -e"
# cat /tmp/b
# rm /tmp/a /tmp/b /tmp/c
```

4 Closing words

4.1 Known Problems

Control tool blocks

The Papillon kernel module uses strict locking mechanisms to prevent data inconsistency therefore the internal configuration is protected by a read-write-lock. This lock allows concurrent execution of the control tool `papctl`.

If a kernel thread operates inside a system call that is intercepted by the module, the control tool `papctl` might return the following error message and doesn't update the configuration.

```
# papctl -s  m=off
Error 16
Configuration blocked.
```

Usually this is a temporary problem and waiting a few seconds is a possible solution. If the control tool still blocks, some application's kernel thread is blocked within an intercepted system call. The best solution is to find this application and terminate it, in most cases leaving the graphical interface solves the problem. Unfortunately finding the application can be an annoying challenge. You can use the `-f` option to force setting the configuration, but you will risk inconsistency in Papillon's configuration.

```
# papctl -f -s m=off
```

Lost Inodes

Some people have reported lost inodes if rebooting the system after the Papillon kernel module has been loaded. The inodes loose their reference if the module is loaded *twice* into the kernel.

Avoid loading the kernel module twice. Before loading the module manually into kernel, check if the module has been already loaded at boot time and unload it if necessary. You can find out if the module has been loaded using the control tool.

```
# papctl -g
Error 12
Papillon is not loaded.
```

4.2 Feedback

Papillon is not yet another security solution by a major company, it is an non-commercial open source project aiming at security enhancing the Solaris Operating Environment. The Papillon module, its components and the documentation have been written in the free time of the author, therefore your feedback is *essential*.

If you discover a bug, the systems panics, you can't compile the source or there is anything else you like to comment, please feel free to drop a mail to Konrad Rieck (`kr@roqe.org`). If you are reporting a problem with Papillon include information about your actual system setup, e.g. by executing the following commands.

```
# uname -a
# isainfo -v -b
# psrinfo -v
# dmesg | tail -10
```

If you want to report a compilation problem pipe the output of the make process into a file and attach it to your mail.

```
# make > /tmp/report 2>&1
```

If the system panics and dumps core, follow the instructions below to generate a stack back-trace of the core image and include the trace in the email. Please only send these traces and don't send core images via email, because they are very large.

```
# cd /var/crash/‘hostname‘
# echo \$c | mdb unix.0 vmcore.0
```

If you have ideas or criticism regarding Papillon and its functionality feel free to drop an email and if you are an experienced programmer take a look Papillon's source, maybe you can contribute a new feature or security protection.

If you are using Papillon in a company or large network and feel that it is great software, please send an email or, if you like, a little donation.

4.3 Thanks

The author would like to thank the following people (in no special order):

- Job de Haas — For his ideas, support and the fun at HAL2001 conference
- Heiko Krupp — For contributing the initial implementation of the chroot protection.
- Fabian Krönner — For hosting Papillon's website.
- Casper Dik — For his helpful advices on the system-dependent GCC headers.
- Philipp Stucke — For providing test environment during the early development.

Thanks to all the people who sent in bug reports and feedback (in order of "appearance"):

Sergei Rousakov	Adam Morley	Eric Thern
Michael Parkin	Juri Haberland	Rikard Skjelsvik
Adam Mazza	Erik Parker	

Bibliography

[1] *GNU Compiler Collection*. GNU Project, Free Software Foundation.
(http://www.gnu.org/software/gcc)

[2] *HAP Linux Kernel Patches*. Hank Leininger.
(http://http://www.doutlets.com/downloadables/hap.phtml)

[3] *Kernel Hacking Project*. Job de Haas, ITSX.
(http://www.itsx.com/projects-kernel.html)

[4] *OpenBSD SPARC and SPARC64 Port*. OpenBSD Project.
(http://www.openbsd.org)

[5] *Openwall Linux Kernel Patches*. Solar Designer, Openwall Project.
(http://www.openwall.com/linux)

[6] *Papillon – Solaris Security Module*. Konrad Rieck, Roqefellaz.
(http://www.roqe.org/papillon)

[7] *Solaris 9 Operating System*. Sun Microsystems, Inc.
(http://www.sun.com/software/solaris)

[8] *Solaris Companion Software*. Sun Microsystems, Inc.
(http://www.sun.com/software/solaris/freeware)

[9] *Solaris Kernel Programming Presentation*. Konrad Rieck, Job de Haas, HAL2001.
(http://www.roqe.org/solkern/solkern.html)

[10] *Sun ONE Studio 7 Compiler Collection*. Sun Microsystems, Inc.
(http://www.sun.com/software/sundev/suncc)

[11] *Sunfreeware – Freeware for Solaris*. Steven M. Christensen and Associates, Inc.
(http://www.sunfreeware.com)

[12] *Trusted Solaris Operating Environment*. Sun Microsystems, Inc.
(http://www.sun.com/software/solaris/trustedsolaris)