Integration of Security Measures and Techniques in an Operating System
(considering OpenBSD as an example)

Igor Böhm
Institute for Information Processing
and Microprocessor Technology
University of Linz
4040 Linz, Austria
Email: igor@bytelabs.org

Abstract— **This paper covers defensive security technologies and secure programming techniques employed in the *OpenBSD* operating system. Privilege separation, a secure design pattern which is implemented in many privileged processes, is discussed based on the network time protocol daemon *OpenNTPD*. Afterwards a closer look at address space and memory protection techniques like W^X and the Stack Smashing Protector is taken. Finally the process of gathering entropy in order to produce high quality random numbers is discussed based on examples found in the *OpenBSD* operating system.**

I. Introduction

OpenBSD believes in strong security and thus has been chosen to serve as an example for an operating system which is implementing security measures into many areas of the system. Their open software development model permits the *OpenBSD* team to take a more uncompromising view towards increased security than Sun, SGI, IBM, HP [5]. There is also a strong believe in full disclosure of security problems which means that security issues do not stay hidden from the users. The source code is constantly audited by members of the *OpenBSD* team who search for and fix software bugs. Because of this audit process, flaws have been found in just about every area of the system. Entire new classes of security problems have been found and often source code, which had been audited earlier, needs re-auditing with these new flaws in mind. Many new ways of how to solve problems have resulted from the audit process. Sometimes these ideas have been used before in some random application written somewhere, but perhaps not taken to the degree that *OpenBSD* does:

- *strlcat()* and *strlcpy()* - size-bounded string copying and concatenation.
- Memory protection purify
  - W^X: This policy is called (W *xor* X) and means that a page may be either writeable or executable, but not both (unless application requested...)
  - Randomized *malloc()*
  - Randomized *mmap()*
  - *atexit()* and *stdio* protection
- Privilege separation
- Chroot jailing
- ProPolice (SSP - StackSmashingProtector)
- ...

The goal of this paper is to take a closer look at some of these topics, explain them in greater detail and present some strategies for secure software development.

II. Privilege Separation

A common problem many daemon processes like *NTPD* (Network Time Protocol Daemon), *SMTPD* (Simple Mail Transfer Protocol Daemon), *HTTPD* (Hyper Text Transfer Protocol Daemon) or *BGPD* (Border Gateway Protocol Daemon) share with each other is that they run with extra privileges granted to them either by the setuid or setgid bits or by the user which has executed them (e.g. root). The reason for this is because such programs require extra privileges at various times throughout their lifecycle [3] (e.g. binding a socket to a privileged port) and thus can not drop the extra privileges permanently.

One way to solve this problem is to use *privilege separation*. The concept of privilege separation is to set up two processes where one process is solely responsible for performing all privileged operations, and it does absolutely nothing else. The second process is responsible for performing the remainder of the program's work, which does not require any extra privileges.

Usually the two processes are closely related, which means that they are the same program split during initialisation into two separate processes using *fork()*. Right after the separation of the two processes, the child process drops all privileges and usually does a *chroot()* into some special directory which has

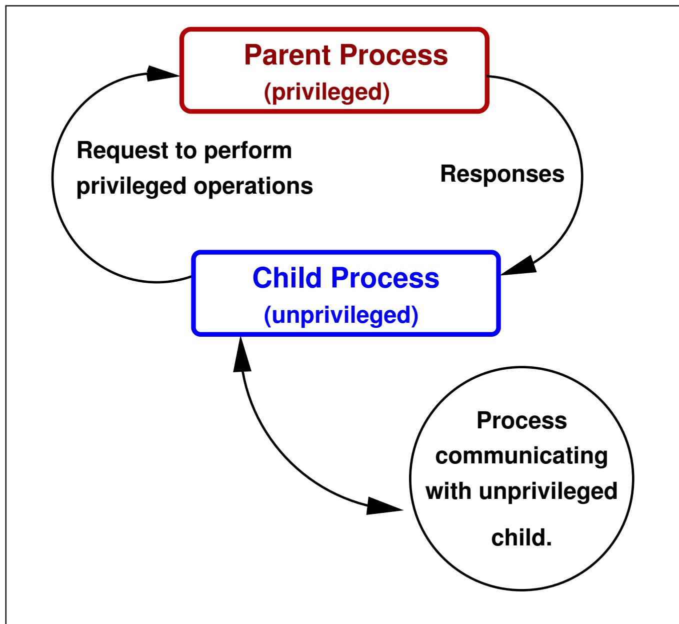

Fig. 1. Privilege Separation: Data flow.

been created for it (e.g. */var/empty*). As illustrated in Figure 1, a bidirectional communications channel exists between the two processes to allow the unprivileged process to send requests to the privileged process and to receive the results.

By splitting the processes into a privileged and an unprivileged part, the risk of privilege escalation attacks is significantly reduced [3]. Since the parent process will refuse to do any operation the child does not need, there is only a small window of attack possibilities left. The unprivileged child usually is responsible for most of the program's functionality and thus stands the greatest risk of compromise, but since it has no extra privileges of its own, an attacker does not stand to gain much from the compromise.

A. Privilege Separation in OpenNTPD

OpenNTPD will serve as a good example for the concept of privilege separation and a closer look at the setup of a privilege separated daemon is taken.

During the initialisation phase, a Unix domain socket pair is created using *socketpair()*, which creates two endpoints of a connected unnamed socket. Afterwards a child process is forked using *fork()* and it immediately drops all extra privileges and *chroot()*'s to */var/empty*. Now there is a parent **ntpd** process running as *root* and the child **ntp** engine runs as the user *_ntp:_ntp*. The communication between the parent and the child is established through the Unix domain socket and a *buffer-* and *imsg-* framework is used for passing messages between parent and child (see Figure 2).

OpenNTPD uses two message types for the communication between the parent *ntpd* process and the childs *ntp engine*:

- **IMSG_ADJTIME:** ntp engine asks the parent to do call *adjtime()* in order to correct the time to allow synchronization of the system clock. This process requires **root** privileges.
- **IMSG_HOST_DNS:** ntp engine asks the parent to resolve hostnames which requires access to */etc*.

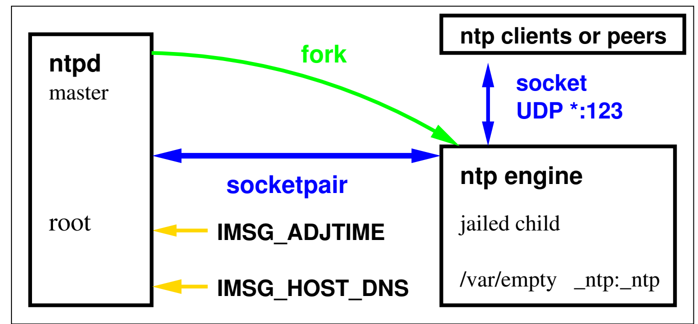

Fig. 2. *OpenNTPD* privilege separated with internal message flow.

This demostrates that there is only a tiny fraction of code running as *root* which corrects the current system time by some offset or resolves hostnames. The rest of the implementation which is basically responsible for

- filtering replies to increase accuracy
- sending queries to all peers
- collapsing the offsets learned from each peer into a single median offset and call *adjtime()*

is much more complex and runs as an unprivileged user *chrooted* to a safe directory.

III. W^X - The Mechanism

A modern operating system defines many different access permissions for files. Some may be executable, some writeable and some may be readable (of course there are also combinations of these permissions) and a good system administrator knows that by giving a file the correct permissions, trouble can be avoided. Taking a look at the operating systems address space reveals that there are many situations where memory is both writeable and executable (permissions = W | X) where it does not need to be, and because of this many bugs are exploitable.

A way to solve this permission problem would be to think about a generic policy for the whole address space, so that each page may either be writeable or executable, but not both unless the application requests it. This policy is called W^X (W *xor* X).

The implementation of such a policy depends on the MMU (MemoryManagementUnit) architecture. Some architectures like *sparc*, *sparc64*, *alpha*, *amd64*, *ia64* and *hppa* have a per-page X bit, whereas with the *i386* architecture it already gets tricky since there is no per-page X bit, but there is a code segment limit which can be used to achieve the same goal.

A. W^X transition on architectures with per-page X bit support

Before it is possible to take advantage of the per-page X bit, a few process address space changes need to be done. First a look at how static binaries look like in memory (see figure

3) must be taken, in order to see why some things have to be moved around.

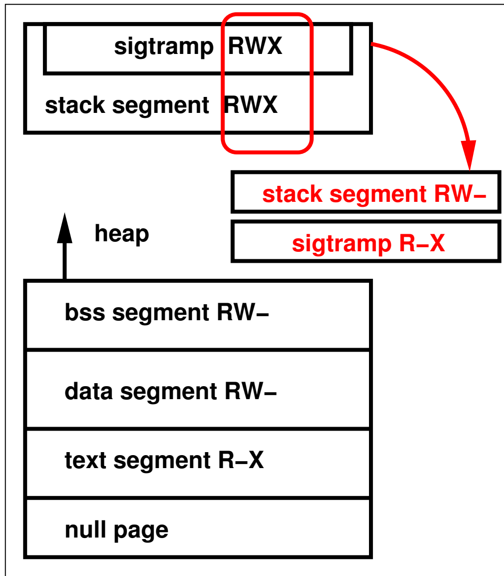

Fig. 3. Signal trampoline separation.

The first goal is to make the stack non-executable, but in order to achieve that, the signal trampoline (*sigtramp*), which is a special piece of code providing support for invoking signal handlers for a process, is at the top of the stack when a new process is created, and thus has to be moved away from the top page of the stack first. The *sigtramp* originally has **RWX** permissions but the write permission is not necessary and thus removed.

Taking a look at how shared libraries are mapped, reveals more places where things can be done better. Even though the *data* segments are supposed to be only **RW-**, they contain objects which are **RWX**. An additional danger is that some objects like *GOT* (shared library Global Offset Table) and *PLT* (shared library Procedure Linkage Table) are writeable when they do not need to be (see left hand side of figure 4).

In order to purify the page permissions *GOT* and *PLT* get their own pages and become non-writeable. After this change, the *data* segment has no more objects with **X** permissions (see right hand side of figure 4). In order for this to work, the runtime link editor has to be changed to cope with these rearrangements. Finally the ELF binary which has been mapped into memory, has no more pages which have write and execute permissions.

B. W^X transition on architectures without per-page X bit

On architectures where the MMU does not support a per-page X bit as on *i386*, other ways have to be found in order to achieve the same goal. On *i386* there is a code segment limit which leads to a less refined range of execution. The basic idea is to split the address space in a *data* and *code* part. This means that there is a bordeline in the address space above which execution doesn't work, so whenever memory is

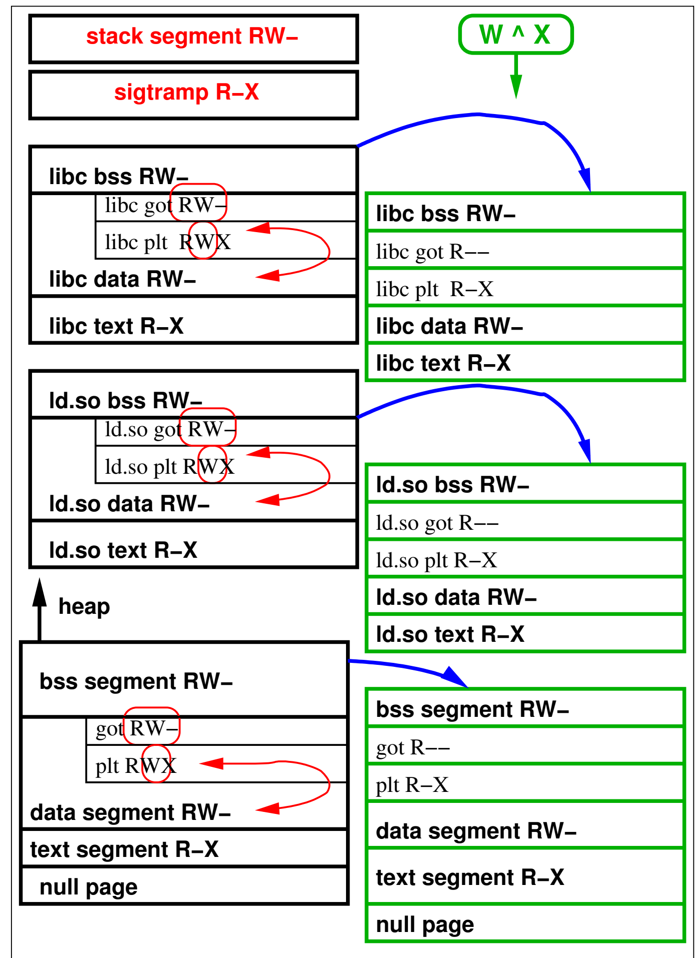

Fig. 4. Mapping dynamic libraries.

mapped, code is split from data and moved low, whereas data is moved up high into the non-executable area of the address space. With this approach it is again possible to split up a binary in order to purify permissions to get the desired W^X effect.

IV. Stack Smashing Protector

The Stack Smashing Protector which has been developed at the IBM research division in Tokio, presents some new ideas for improving the state of the art in buffer overflow detection. Basically the main ideas are the reordering of local variables to place buffers after pointers to avoid the corruption of pointers that could be used to further corrupt arbitrary memory locations, the copying of pointers in function arguments to an area preceding local variable buffers to prevent the corruption of pointers that could be used to further corrupt arbitrary memory locations, and the omission of instrumentation code from some functions to decrease the performance overhead.

In order to fully understand the concepts of the Stack Smashing Protector, a closer look at how a typical stack structure is organized, after a function is called, must be taken, and the possible kinds of attack scenarios must be classified.

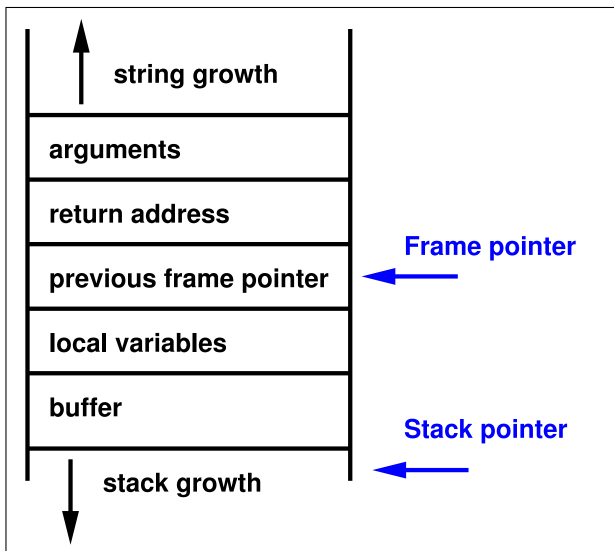

Fig. 5. Stack structure after a function is called.

```
void foo()
{
   long *variable;
   char  buf[10];
   ...
   strcpy(buffer, "Hi my name111122223333");
   ...
}
```

Fig. 6. Buffer overflow example.

The typical stack structure, after a function is called, is shown in Figure IV with the *stack pointer* pointing to the top of the stack. The C programming language uses the area from the top of the stack in the following order:

- local variables,
- the previous frame pointer,
- the return address,
- and the arguments of the function

where the frame pointer locates the current frame and the previous frame pointer stores the frame pointer of the caller function.

Now let's have a look at a simple buffer overflow example in order to get a better understanding of the problem. The function *foo* (see Figure IV) demonstrates a buffer overflow which occurs because of the fact that the function *strcpy()* does not check the size of the output, thus it can copy more than 10 bytes of data to *buf*. What will happen is that the characters "Hi my name", 0x01010101, 0x02020202 and 0x03030303 will be assigned to *buf*, *variable*, the previous frame pointer, and the return address respectively, if we assume that 32 bit variables are used. After the *foo* function finishes, it will return to it's caller by jumping to the address pointed to by the previous frame pointer, but since this pointer has been overwritten with the value 0x03030303, the function will return to that address instead, which isn't the caller address. If we even further assume that malicious code is located at the address 0x03030303, it will be executed with the same privilege level as the application.

Since it should be clear by now how buffer overflows can be exploited, a classification of attack methods targeting the stack can be made:

- *Return address:* This is the most popular attack method. The functions return address is overwritten with an address pointing to malicious code.
- *Local variables:* Attacking the local variables usually has very little effect and is seldomly used.
- *Argument variables:* The function pointer variable is another popular target for attacks. Assigning the function pointer variable of an argument or a local variable to the attack code is a typical attack method. In this case, the vulnerable place can be found by checking the source program.
- *Previous frame pointer:* Since the location of the return address is determined by the frame pointer, and the frame pointer is assigned to the value of the previous frame pointer at the time of function return, it is possible for an attacker to create a fake frame that has the return address pointed to attack code.

A. Stack Protection Method

As described in the previous section, there are four areas on the stack which need special attention and protection: the location of the arguments, the return address, the previous frame pointer and the local variables.

In order to be able to protect the first three areas from change, a guard variable is introduced. The guard is inserted next to the previous frame pointer and it is prior to an array, which is the location where an attack can begin to destroy the stack.

A random value is chosen for the guard variable at the function prologue and the random value assigned to the guard is checked by the function epiloge. So the main requirement of the guard value is that it must be a value that an attacker can't know. Random numbers are used as guard values which are calculated at the initialisation time of the application which can not be discovered by a non-privileged user.

Now it is possible to introduce a safety function model, which involves a limitation of stack usage (see figure IV-A) in the following manner:

- the location **(A)** has no array or pointer variable
- the location **(B)** has arrays or structures
- the location **(C)** has no array

Now the only vulnerable location for stack smashing attacks is location **(B)**. But since it is now possible to detect such attacks by verification of the guard value, program execution will stop immediately if damage outside the frame has been detected.

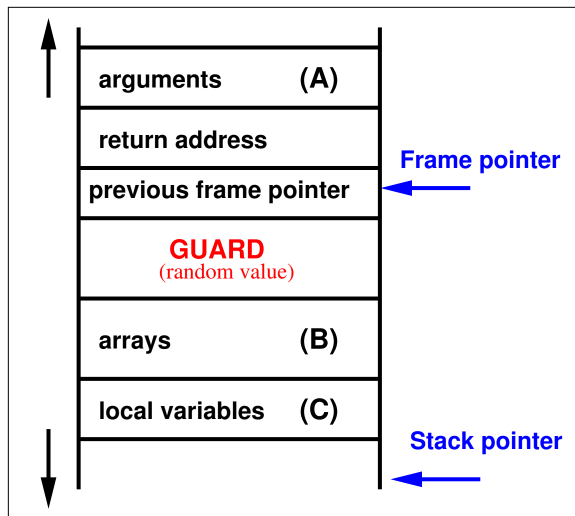

Fig. 7. Safe stack frame structure after a function is called.

In order to protect the fourth area from change, the location of arrays and local variables is swapped. Arrays are now placed closer to the random guard value and integers and pointers are placed further away. Because of this reordering, an attack on pointer variables in a function frame will also not succeed since the location **(B)** is the only vulnerable location for a stack smashing attack and the damage goes away from location **(C)**.

V. Random Numbers

The computer has been designed to be deterministic and very predictable. Hence it is very hard and difficult to produce truly random numbers on a computer as opposed to pseudo random numbers, where the sequence of numbers is deterministic. So even with a large period of random numbers, it is easy for the skilled attacker to guess the correct sequence, and for some applications this is not acceptable. Instead, ”environmental noise” from the computer’s environment, which must be hard for outside attackers to observe, is gathered in order to generate random numbers. The best place for gathering environmetal noise on a UNIX system, is from inside the kernel. Sources of randomness must be chosen with care and they must fulfill the following requirements:

- non-deterministic
- hard for an outside observer to measure

Examples of such sources of randomness would be:

- inter-keyboard timings
- inter-interrupt timings
- finishing time of disk requests
- mouse-interrup timings
- finishing time of net input
- *tty* activity which consists of inter-keypress timings as well as the character which has been entered
- audio randomness
- ...

Randomness from these sources is added to an ”entropy pool”, which is mixed using a CRC-like function. This is not cryptographically strong, but it is adequate and fast enough so that the overhead on doing it on every interrupt is reasonable. All these random bytes are mixed into the entropy pool, and the routines keep an estimate of how many bits of randomness have been stored into the random number generator’s internal state.

Whenever random bytes are desired, they are obtained by taking the MD5 hash of the content of the entropy pool. An MD5 hash is used in order to avoid exposing the internal state of the entropy pool, since it is believed that it is computationally infeasibale to derive any useful information about the input of MD5 from its output. Even if it would be possible to analyze the output of MD5, the output data would still be totally unpredictable as long as the amount of data returned from the generator is less than the inherent entropy in the pool. For this reason, the routine which outputs random numbers, decreases its internal estimate of how many bits of ”true randomness” are contained in the entropy pool.

OpenBSD offers various random devices producing various random output data with different random qualities. As described earlier, entropy data is collected from system activity, and then run through various hash or message digest functions to generate the output.

- **/dev/srandom:** Provides strong random data. If the entropy pool quality starts to run low, which means that sufficient entropy is currently not available, the driver pauses while more of such data is collected.
- **/dev/urandom:** Provides random data without the guarantee that random data is strong. So if the entropy pool quality runs low, the driver will continue to output data.
- **/dev/prandom:** Simple pseudo-random generator.
- **/dev/arandom:** This generator re-seeds an ARC4 generator with its entropy pool, and the ARC4 generator then generates high-quality pseudo random output data.

A. Usage of Random Numbers

Since it has been explained in the previous section how random numbers are being generated, it is time to look at the various places where they are being used in an operating system. The TCP/IP network stack implementation has various places where it relies on a good random number generator in order to make it harder for an outside attacker to spoof packets, to blindly inject data or to reset established connections. *OpenBSD* assigns source port numbers randomly whereas many other operating system vendors allocate source port numbers sequentially which makes it a lot easier for an attacker to find the correct source port.

The 32-bit sequence number field in the TCP header, a value that starts with a randomly generated arbitrary integer which then increments sequentially with each transmitted packet, is another place where a very fast and good random number generator is needed.

Another interesting place in *OpenBSD* where random data is used, can be found in *sys/kern_exec.c* where the routines for the

execution of executables are implemented. Before a process can be safely executed, one of the components which needs to be set up, is the processes stack. Usually buffers are always at the same place on the stack, which can be a security problem. Stack-based buffer overflows rely on this predictable way of how the top of the stack is allocated. Basically an attacker overflows a buffer on the stack, the overflow overwrites the function return address with a fixed value pointer into the overflow buffer and execution starts. A possible solution for this problem is to introduce a random-sized gap at the top of stack. This method is called Stackgap and it minimizes the chances of such a stack-based buffer overflow attack.

Since address space allocations and mappings are fairly predictable, randomization of address space is introduced. Each time when the system call *mmap()* is used, which maps files or devices into memory, and the flag MAP_FIXED is not specified, a random address is chosen for the allocation. So each time a program is run, it will have different address space behaviour. Also the addresses of objects which are allocated by *malloc()* are fairly predictable and some recent exploits have even relied on this behaviour. *malloc()* handles two types of objects:

- Objects which are smaller than a page:
  For such objects *malloc()* maintains buckets of ”chunks” and it is possible to randomize chunk selections out of the bucket.
- Objects which are equal or greater than a page:
  In this case *malloc()* relies on random *mmap()*.

When an ARPA internet protocol socket is bound to a specific port number using the *bind()* system call, the system can choose the specific port, or elect that the system chose [7]. Normal UNIX behaviour resulted in the system allocating port numbers starting at 1024 and incrementing. In *OpenBSD* a random port in the range of 1024 to 49151 is chosen.

Another more obvious place where good random numbers are needed can be found during the initialisation of volatile encryption keys. Obviously a strong source of randomness which is represented by a pseudo-random generator whose output is not really random, but depends on so many entropy providing physical processes that an attacker can not pactically predict its output, is a fundamental and important basis for many cryptographic applications.

Of course there are many more applications and examples which rely on random numbers, but it would exceed the scope of this paper to describe them all.

VI. Conclusion

Security is like an arms race, because the best attackers will continue to search for flaws in a system and craft an exploit which grants them an advantage. This means that it is high time for defensive technologies which make it harder to write an exploit [2], by making a system environment more hostile towards exploitation without impacting well-behaving processes. The *OpenBSD* team has in many ways proven that their approach of proactive security and their ”secure by default” policy has made it a very secure and functional operating system, ready for production usage in a hostile environment.

References

[1] Henning Brauer, ”OpenNTPD”, *http://www.openbsd.org/papers/*, September 2004.

[2] Theo de Raadt, ”Exploit Mitigation Techiques”, *http://www.openbsd.org/papers/*, 2004.

[3] John Viega, Matt Messier, ”Secure programming cookbook”, Sebastapool CA, 2003.

[4] B. W. Kernighan and D. M. Ritchie. ”The C Programming Language: ANSI C Version, second edition.” PrenticeHall, 1988.

[5] ”Security goals of the OpenBSD project”, *http://www.openbsd.org/ security.html*, 2004.

[6] Hiroaki Etoh, Kunikazu Yoda , ”Protecting from stack-smashing attacks”, IBM Research Division, Tokyo Research Laboratory, June 19, 2000.

[7] Theo de Raadt and Niklas Hallqvist and Artur Grabowski and Angelos D. Keromytis and Niels Provos, ”Cryptography in OpenBSD: An Overview”, *http://citeseer.ist.psu.edu/article/raadt99cryptography.html”*, 1999.

[8] N. Provos. ”Encrypting virtual memory”. In Proceedings of the Ninth USENIX Security Symposium, pages 35-44, Denver, CO, August 2000.