Boosting MVX Systems Through Modern OS Extensions

Lennert Franssens

Supervisor(s): prof. dr. ir. Bjorn De Sutter, prof. dr. Bart Coppens, dr. ir. Bert Abrath

Abstract — **Multi Variant Execution (MVX) Systems provide a technique for detecting security problems related to memory corruption in software. Traditional MVX Systems are very secure but work relatively slowly. Relaxed Monitoring (ReMon) is a design that effectively speeds up the execution of MVX Systems. A kernel patch was used to implement that design. Many people prefer not to apply a kernel patch to their system. This reduces the use of ReMon. In this dissertation we investigate whether the use of a kernel patch in the implementation of the ReMon design can be replaced by modern OS Extensions.**

***Keywords* — security, MVEE, MVX systems, Relaxed Monitoring, ReMon, modern OS extensions, seccomp-BPF**

I. Introduction

MVX Systems are used to detect memory-related vulnerabilities during a program execution. To discover such vulnerabilities, several variants of an application are executed simultaneously. The behavior of these variants is compared on the level of system calls by a monitor. A monitor is a ptrace process that manages the execution of different variants.

A ptrace process, the monitor in the case of an MVX system, introduces a large overhead during program execution. This is because a ptrace process does a few context checks per system call that it intercepts. To reduce this overhead, a new design was made. That design is Relaxed Monitoring or ReMon for short. A set of non-safety-sensitive system calls are not analyzed by the monitor using the ptrace process. Instead, they are analyzed by a monitor that does not use the ptrace technology and is therefore much faster. Instead, it is mapped in the address space of the executing variant.

The original ReMon design was implemented using a kernel patch. The intention of this dissertation is to replace the design of ReMon and the operation of the kernel patch by using modern OS extensions. It is important that a high level of security is maintained, and that the execution speed is on a par with the original ReMon design and its implementation.

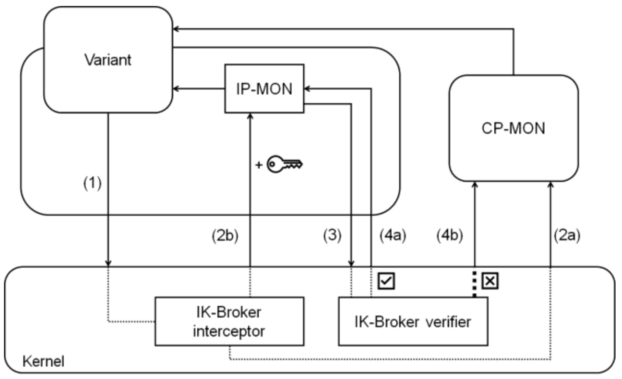


Figure 1. Original ReMon design

II. ReMon

The original design of ReMon is shown in Figure 1. When a variant makes a system call (1), the in-kernel-broker interceptor (IK-Broker interceptor) will intercept it. The IK-Broker interceptor is a component that lives in the kernel and thus is implemented through the kernel patch. The interceptor will choose what to do with the system call. If it is a security-sensitive system call, it is forwarded to the cross-process monitor (CP-MON) for further analysis (2a). CP-MON is a - slow but highly reliable - monitor based on a ptrace process.

When the IK-Broker interceptor decides that it is not a security-sensitive system call, it will generate a secret that is forwarded along with the system call to the in-process monitor (IP-MON) (2b). In IP-MON, some security checks are done. After that, IP-MON will try to execute the system call. That attempt is again intercepted by a component in the kernel (3), namely the in-kernel-broker verifier (IK-Broker verifier). The verifier will check, based on the secret obtained in step (2b), whether the system call was forwarded by the interceptor to IP-MON. In addition, it will also check if the system call still has the same properties as when it passed through the interceptor. If the secret matches, and the system call has the same properties as before, IP-MON may analyze the system call during its actual execution (4a). In all other cases, the system call will be analyzed by CP-MON (4b).

III. Seccomp-BPF

seccomp-BPF is a technology in the Linux kernel that allows the delegation of the further course and or execution of system calls that are made by the process where seccomp-BPF is enabled and installed. To do this, it uses a programmable Berkeley Packet Filter (BPF). This is a filter that makes decisions based on the number of a system call and its arguments. The following list summarizes some - for this dissertation interesting - decisions:

SECCOMP_RET_ALLOW — The system call can be executed as usual.

SECCOMP_RET_ERRNO — A 12-bit errno value is returned from the kernel without executing the system call.

SECCOMP_RET_TRACE — The kernel will notify the tracer process, which uses ptrace technology, of the program being filtered even before the system call is executed. The tracer will delegate the remainder of the system call.

As presented in Figure 2, a program will make system calls such as read, write.... The seccomp-BPF mechanism will intercept these system calls and have them evaluated by the pre-programmed BPF filter. The BPF filter will determine the further course of the system call based on its statements. The filter explicitly describes which system calls will pass through directly and which will go to a ptrace process (tracer) for analysis, by the signals SECCOMP_RET_ALLOW, and SECCOMP_RET_TRACE respectively.

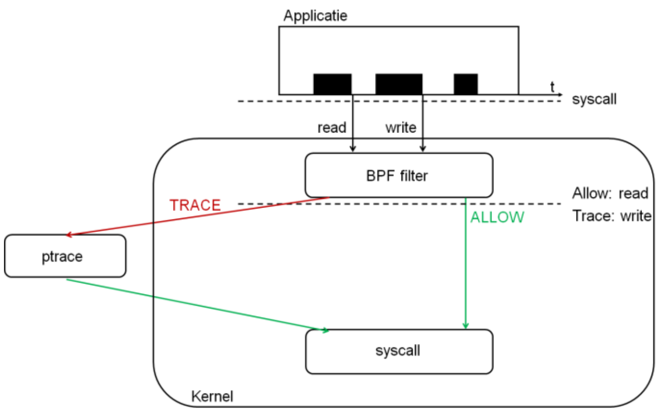


Figure 2. Flow of filtering system calls with seccomp-BPF

One minor drawback is that Berkeley Packet Filters cannot be used to dereference the arguments of a system call, such as strings. The filter will only be able to evaluate direct values, and not pointers.

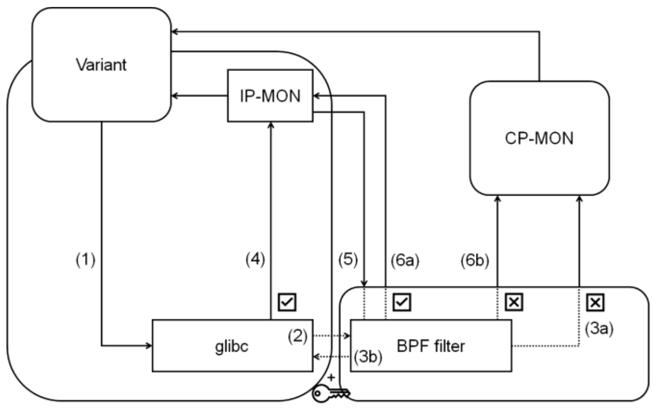


Figure 3. New ReMon design with seccomp-BPF

IV. New Design

We modify the MVEE, IP-MON and a modified version of glibc to replace the kernel patch. Figure 3 is the new design and shows the interaction between the aforementioned components.

System calls coming from a variant are converted to the correct form by glibc and then forwarded to the kernel (1). The first step is to catch the system call in glibc. In glibc, we can modify the function that performs the system call. Through those modifications, a two-stage system can be made.

The first step in that system is to execute the system call by calling the syscall instruction from glibc the first time. That ensures that the system call must pass through the seccomp-BPF filter (2). That filter determines whether the system call should be executed by the cross-process monitor or the in-process monitor.

If the filter decides that it is a security-sensitive system call, it will forward the system call to CP-MON (3a). If the filter decides that it is a non-security-sensitive system call, it will send a secret back to glibc (3b). Through the secret, we indicate that we should execute the second step in the system in the custom glibc function. In addition, we use that secret to let glibc know the address of the in-process monitor. That way, glibc knows which address to jump to in order to go to IP-MON (4).

In IP-MON, we execute the system call again (5). The seccomp-BPF filter will catch this call again and reevaluate it. If the filter sees that the system call is made from a specific, predefined address IP-MON and it is still a non-security sensitive system call, the system call will simply be executed and analyzed by IP-MON (6a).

System calls that do not come from IP-MON can never be executed and analyzed directly in IP-MON. CP-MON will analyze them instead. If the system call comes from the IP-MON but the arguments or the number of the system call is changed, it will be forwarded to CP-MON (6b).

V. Implementation

To make the new design work, two crucial changes were made. The first change is in IP-MON itself. IP-MON - which is mapped in the address space of the variant that is being analyzed – enables and installs a seccomp-BPF filter directly into the variant. This filter is set up to work with the two-stage system from glibc. Listing 1 shows in pseudocode the operation of the filter. We see a filter that allows only harmless, non-security-sensitive system calls to be analyzed by IP-MON. The filter evaluates to a redirect to CP-MON in the case of a security-sensitive system call. The first if-else clause refers to the second stage from the two-stage system. The second if-else clause, in turn, refers to the first stage from the two-stage system introduced in glibc.

```
set bpf_allowed_syscalls = [__NR_getpid, ..., __NR_gettid]

if ipmon_syscall_address_ptr == seccomp_data.instruction_pointer do
    if seccomp_data.nr in bpf_allowed_syscalls do
        return SECCOMP_RET_ALLOW
    end else do
        return SECCOMP_RET_TRACE
    end
end else do
    if seccomp_data.nr in bpf_allowed_syscalls do
        return SECCOMP_RET_ERRNO
    end else do
        return SECCOMP_RET_TRACE
    end
end
```

Listing 1. Operation of the seccomp-BPF filter in pseudocode

Through the SECCOMP_RET_ERRNO action, we can pass secrets from the seccomp-BPF filter to glibc. That secret is the address that glibc must jump to in order to get into IP-MON.

The second change is in glibc. There we modify the system call macro to a custom system call function to be executed. Listing 2 gives the instructions contained in that function. Again, we see the two-stage system coming back. An initial system call is executed. When the seccomp-BPF filter evaluates to SECCOMP_RET_TRACE, it will go to the end of the custom system call function and the system call will be evaluated by CP-MON. When the seccomp-BPF filter evaluates to SECCOMP_RET_ERRNO, which will only return a value greater than 0x07FF in its first execution, the continuation of the custom system call function is executed. There, the address of IP-MON will be stored, then jumped to, after which the second stage in the two-stage system will be entered. The number of times we return an errno value may increase as we seek to transmit larger secrets.

```
    movq %rax,%r12
    syscall
    cmpq $-0x07FF,%rax
    jge glibc_custom_syscall_exit

    neg %rax
    shl $12,%rax
    movq %rax,%r11
    movq %r12,%rax
    call %r11

glibc_custom_syscall_exit:
    nop
```

Listing 2. Operations in the custom system call function in glibc

VI. Security Evaluation

Care must be taken with the modified version of glibc in conjunction with the seccomp-BPF filter. When a system call enters the filter for the first time, it is either redirected to IP-MON by returning a secret or it goes directly to CP-MON. This is a decision that should only happen if we are sure that the system call first passes through the modified, and loaded into the variant, version of glibc. If we don't impose restrictions on when the secret may be returned, the secret may leak. One possibility is that there is inline assembler written into the program, which will not pass through glibc but will pass through the seccomp-BPF filter to find out the secret.

To prevent potential abuse of that secret, the idea is to potentially redirect only system calls that pass through our modified glibc to IP-MON by only then sending the secret along. If it does not pass through our custom version of glibc, it will be sent to the cross-process monitor anyway. That can be implemented by mapping glibc on a known address. When a system call enters the seccomp-BPF filter for evaluation, we can compare the instruction pointer with a precalculated pointer of where our custom system call function is mapped.

VII. Benchmarks

The first measurement to test the new design and its implementation is a microbenchmark. To calculate the execution time of a getpid system call as accurately as possible, we write a program that will execute the getpid system call repetitively.

The measurements are performed on four versions. A default MVX system, ReMon that does not use the IP-MON component. A second version is the original version of ReMon that uses IP-MON. The third and fourth versions are versions of the new implementation with seccomp-BPF. A distinction is made in the size of the secret that is passed. In the version with a 12-bit secret, the secret is passed in 1 pass. In the 48-bit version, the filter is passed 4 times in order to pass the secret.

Measurements were performed for two cases: analysis of getpid in CP-MON and analysis of getpid in the faster IP-MON monitor.

In Figure 4 we see the results of the measurements. We can see that the new implementation introduces a small overhead by having to execute the filter and evaluate to a response through which the getpid system call is forwarded to CP-MON for analysis.

In the case where the getpid system call is forwarded to IP-MON for analysis, we see speeds of the new implementation that are in the same order of magnitude as those of the original implementation. At a secret of 12 bits, the new implementation is even faster. The 48-bit version is more secure but has a larger overhead due to running the seccomp-BPF filter multiple times to obtain the secret.

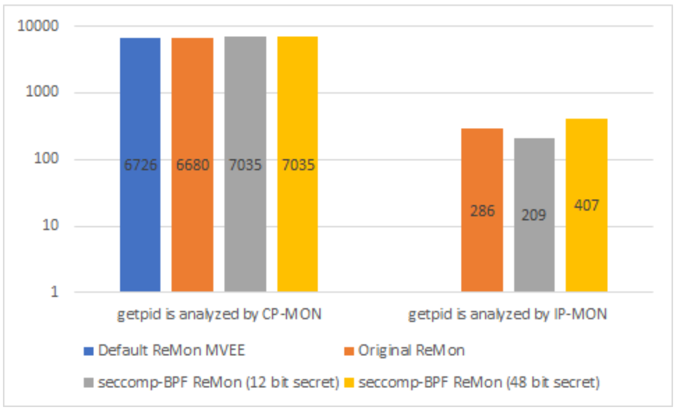


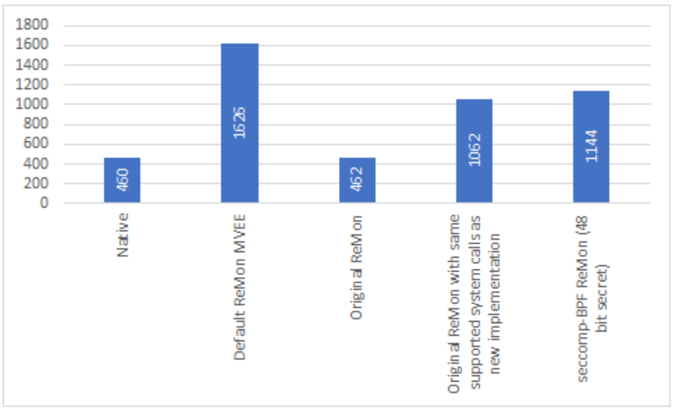
Figure 4. Time to execute one getpid system call [*ns*]

The second benchmark is the execution of a web server program nginx under the watchful eye of the MVX system. The benchmark was executed in five different ways. A native execution, an execution under the supervision of ReMon without using IP-MON. There is also an execution in the original implementation of ReMon. The latter two versions are both the new implementation of ReMon with seccomp-BPF and the original version of ReMon that uses IP-MON with the same set of supported system calls as those of the new implementation.

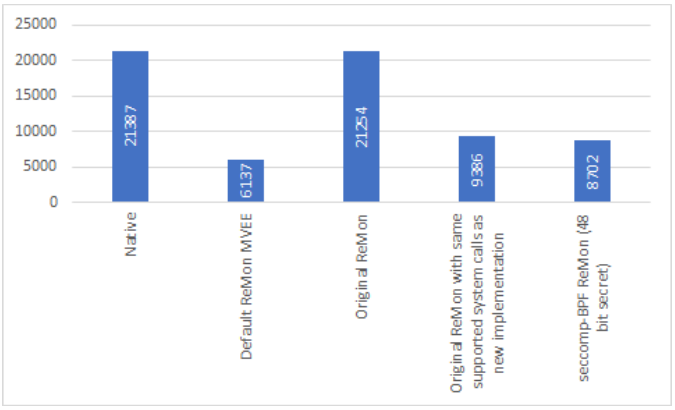
In that benchmark, the latency that is introduced by how the application is executed was measured. In addition, throughput was also measured, which has an inverse relationship with latency.

We see that the native execution and the execution under the original implementation of ReMon with IP-MON achieve equal execution time, and thus latency and throughput. The default MVEE ReMon, which does not use IP-MON is clearly the slowest and has the highest latency and the lowest throughput.

The new implementation is substantially slower than the original implementation of ReMon but remains faster than the default MVEE version of ReMon. To verify that this is solely due to the limited set of supported system calls that can be analyzed in IP-MON in the new version, we also did the measurement for the old version with an equal set of supported system calls for IP-MON. We a speed of both versions similar to each other. We can say that the extra latency is introduced by not yet supported system calls in the new version of ReMon with IP-MON. There is also another small overhead (~80 μs), which we can attribute to the use of a large secret that requires the seccomp-BPF filter to be executed multiple times.



(a) Latency [μs]



(b) Throughput [*requests/s*]

Figure 5. Web server (nginx) benchmark results

VIII. DRAWBACKS

The new implementation of ReMon has only a limited set of system calls that can be analyzed in IP-MON.

When multiple filters are installed, by performing clone or execve in the variant, the evaluation of the filter runs incorrectly. This is because all set filters will be executed. The first action with the highest priority that was evaluated in the set of filters is taken as the return value. Because of that feature, the evaluation potentially goes awry because a difference in priority of return values is normally bridged by the two-stage system we have implemented. A solution to this is to map IP-MON to the same place in memory as the parent process after a clone or execve.

Multiple filters can also be installed by the program to be monitored itself. For example, modern browsers use seccomp-BPF filters. When such applications are monitored by ReMon, they set up new, additional seccomp-BPF filters, in the process where we installed our own filter. Again, this potentially leads to problems due to a difference in priority of the return value of the seccomp-BPF filter.

IX. CONCLUSION

The new design and implementation, which uses seccomp-BPF filtering, is promising. The new implementation achieves speeds comparable to the original implementation for the system calls that already have support for the new implementation. In addition to the speed aspect, efforts were also made to maintain a high safety aspect, which is also met. That by expanding the new implementation to support larger-sized secrets by returning multiple errno values as a secret. However, an overhead in speed must be taken into account, which grows linearly as the size of a secret grows.

The set of system calls that already are supported in the new implementation is not yet ready for mind-blowing performance. But there is potential in this new version, which can be said to be comparable in speed to the original implementation when we compare only the already supported system calls of the new implementation.