StackSupervisor - a hypervisor-based, kernel stack protector

Grant Foudree
Computer Engineering
Iowa State University Ames, Iowa
gfoudree@iastate.edu

***Abstract*— Stack-based buffer overflow attacks have been around for a while and have been a popular technique for exploiting software. As a result, several mitigation techniques have been proposed and implemented, however none solve the problem completely. In this paper we will propose a new technique called StackSupervisor that protects software from buffer overflow attacks, has the ability to heal and stop them, and improves upon existing mitigation technologies. Specifically, we will leverage the higher privilege level of a hypervisor to detect/heal stack smashing attacks in a guest operating system's kernel, as well as discuss how the technology could be applied to user-space binaries as well.**

I. Introduction

In this paper, we will be presenting a new solution for stack smashing attacks by leveraging the higher privilege level of a hypervisor and performing introspection of the guest kernel's stack. In order to better understand how the proposed solution to preventing stack smashing attacks works, various relevant technologies and concepts will be explained below.

II. X86 Calling Convention

In the X86 CPU architecture, there is a protocol used by compiled software for calling functions and exchanging the parameters as well as the return address. This protocol varies depending on operating system and is known as the "calling convention" for that specific architecture and OS. In order to preserve application binary interface compatibility, programs and their respective libraries need to follow the same calling convention.

X86 programs typically leverage both a stack and set of registers to perform computation. In a 64-bit, Linux environment, calling a function will result in the first parameter being placed in the RDI register, followed by the second in the RSI register and so on, following the pattern of RDX, RCX, R8, R9, XMM0-7. After the parameters are loaded into the registers, the program will push the address of the next instruction that should execute after returning from the function onto the stack, then jump to the call target [1]. After a function is done executing, it pops off the return address from the stack and jumps to it, continuing the execution of the program.

III. Local Buffers

When a function is called and entered, a region on the stack will be created for any local variables that will be used in the procedure. After the procedure exits, the frame is collapsed and this memory region is reused for other functions [2]. This process is known as creating a stack frame. If we look at the following C function, we can observe that 4 local variables are being used.

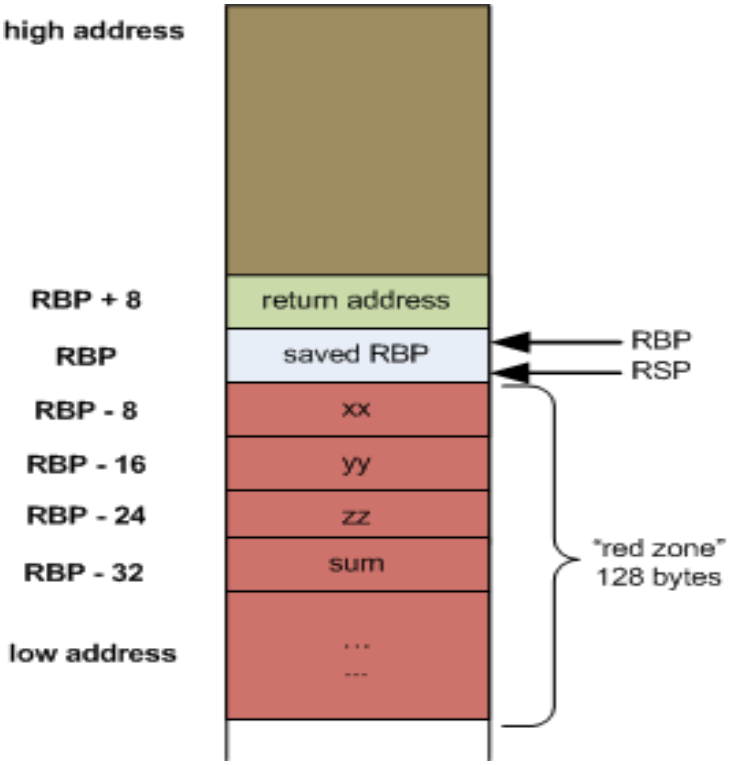

Fig. 1. Stack Frame after calling into a function

```
long utilfunc(long a, long b, long c)
{
    long xx = a + 2;
    long yy = b + 3;
    long zz = c + 4;
    long sum = xx + yy + zz;
}
```

The corresponding stack frame after the function *utilfunc(a,b,c)* has been called is displayed in Fig 1. Note that the local variables and return address are stored on the stack, along with the parameters being passed in the registers as shown in the previous section.

IV. Stack Smashing

A stack smashing attack is a popular technique for exploiting software, and operates by feeding data to a program that causes it to overrun the receiving buffer. Consequences include crashing the program, or worse obtaining arbitrary code execution. The concept exploits errors in software where the programmer does not check or limit the size of input obtained from a user. This is very common in languages where the programmer is responsible for fine-grained memory management such as C or C++.

If we look at the example code in the previous section, suppose we append the following code inside the function:

```
char buf[10];
gets(buf);   //Read from STDIN
```

If an attacker sends more than 10 characters to the program, they will start to overwrite the variable *sum* on the stack, followed by *zz, yy, xx,* and finally the return address. By determining the proper length of the buffer and adjacent stack variables, an attacker can craft a string with the proper length and overwrite the return address to point to a location in memory that contains code to the attacker wants to execute. The formula for doing so on 32-bit systems is:

```
exploit_string = '\x90' * len(buffer
 ↪  + 4) + return_address +
 ↪  shellcode
```

V. Stack Smashing Protections

A. DEP or W xor X memory

Data execution protection and Write xor Execute Memory are the same technique that states that regions of the executable can be marked as write or executable, but not both [3]. From a security standpoint, you don't want to have regions that can be written to also as marked as executable in order to prevent new code from being introduced inadvertently. Write xor Execute Memory acts as a prevention technique for attackers who inject shellcode into the stack after overflow a buffer and pointing the return address back into the stack. Since the stack should not be executable under normal conditions, the permission of the stack should be set to RW without the execute bit, preventing stack-based shellcode execution.

The issue with W xor X memory is that it does not prevent return-oriented programming techniques to achieve arbitrary code execution after a successful stack smashing attack. Therefore, this technique by itself is not a sufficient protection.

1) Return-oriented programming (ROP): Return-oriented programming, or ROP, is a circumvention technique to the W xor X memory protection. The main concept behind ROP is that instead of loading your shellcode onto the stack and then returning to it, you reuse code (gadgets) that already exist inside the binary in a series of return chains. Since the stack is no longer executable, shellcode you load into it cannot be returned to and executed, but all the remaining code inside the compiled binary is fair game since it is marked as executable. In order to leverage ROP, an attacker must assemble a series of gadgets from inside the binary. The criteria for a gadget is that there must be a series of desirable instructions that end with a *RET* instruction. By searching the binary, one can dump various gadgets and order them in a way to complete the desired operation. The larger the binary is, the more likely an attacker can construct a series of gadgets to achieve their goal. Once they have identified the memory regions of their gadgets, an attacker overflows the buffer of a vulnerable program, placing the memory addresses of the gadgets to execute in reverse order with the first one in the overflowed return addresses' spot on the stack. Upon function return, this chain will be activated and one by one the gadgets will be executed, each returning to the next in line.

For example, suppose an attacker wanted to invoke the *syscall exit* as their exploit to a vulnerable program. After searching through the program's binary code, the following gadgets were extracted:

```
0x4004d0: xor %eax, %eax
          ret
;...
0x400100: int $0x80
          ret
;...
0x4FFFFF: inc %eax
          ret
```

In order to call *exit(2)* in 32-bit Linux, the exit code goes in the EBX register, EAX must be set to 1 for syscall 1 (*exit(2)*), and then interrupt 0x80 fired [4]. Using the above gadgets, we can achieve this by executing the gadgets in the following order: *0x4004d0, 0x4FFFFF, 0x400100*. To do this with our stack overflow, we simply need to place the address of the first gadget (*0x4004d0*) in the original function's return address location, followed by *0x4FFFFF* and *0x400100*.

Although this circumvents DEP/W xor X memory, StackSupervisor is effective in protecting against this type of attack due to how it checks the return address as we will describe later.

2) Ret2LibC: Another technique to circumvent W xor X memory is called Ret2LibC and it is similar to ROP in the sense you are reusing code (libc) to circumvent the non-executable stack permissions. Like a standard stack-smashing attack, you need to overflow the buffer and

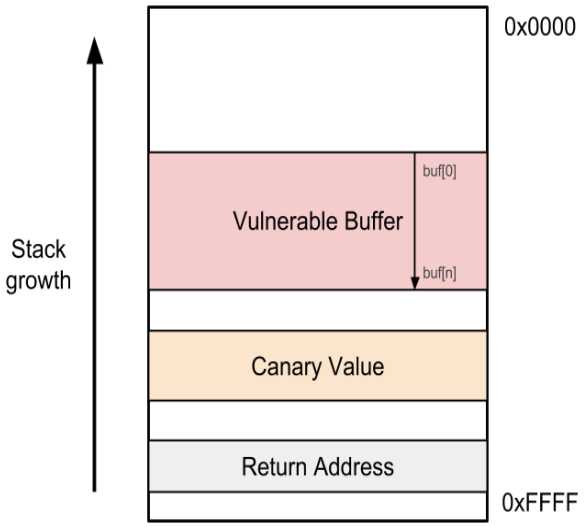

Fig. 2. Stack Canary

modify the return address. However instead of setting the return address to your shellcode on the stack, you point it to an existing function from libc that has been compiled into the program such as *system(3)*. On 32-bit Linux, parameters are passed on the stack - which the attacker controls now. Combining these concepts, an input can be crafted to a vulnerable program to pass in a command that can be executed by *system(3)* [5].

In addition to protecting against ROP, StackSupervisor is effective in preventing a Ret2LibC attack as well based on the same concept of checking the return address.

B. Stack Canaries

Stack canaries are a popular technique that involves placing a canary value on the stack right before the return address. The theory is that if a buffer is overrun, given the order of the stack, a canary placed between the local variables and the return address will get overwritten before the return address. If we check if the canary has been modified before returning from a function, we can infer whether or not a buffer has been overrun and the return address modified. [6]

Canaries are effective if chosen to be a random value and the program itself does not have a vulnerability in which the canary value can be leaked. If this is not the case, an attacker can simply insert the proper canary, by leaking the value via a side-channel, in their exploit string in the proper location as to overflow the buffer, write the exact same canary value back, followed by the malicious return address. In addition, stack canaries introduce some additional overhead in functions due to the checking of the canary value upon the function returning. For these reasons, stack canaries are not entirely perfect.

C. ASLR

In order to combat the predictable nature of the return address in a program, ASLR was introduced. ASLR stands for address space layout randomization, and essentially randomizes the addresses various regions inside of an executable. Specifically, ASLR randomizes the address of the stack and heap when the program is loaded, causing significant uncertainty when performing a stacking smashing attack as the return address to which the attacker's shellcode exists at changes every execution with high entropy. In addition to the stack being randomized, other regions such as the code and dynamic libraries region can be randomized as well. Assembling gadgets for a ROP attack is also complicated due to the shifting of addresses upon each execution [7].

ASLR has some weaknesses, and like stack canaries, if some sort of information leak exists in the program the addresses of the stack and heap can be leaked and used by the attacker to discover the proper return address for their shellcode. Furthermore, some programs allow this address to be brute-forced by allowing unlimited attempts into the program without relaunching the executable and consequently changing the stack address. Through a matter of brute-force, an attacker will eventually guess the correct stack address and be successful [6]. Examples of these programs are network servers which launch a new thread or fork for each client, but ASLR does not randomize these regions.

VI. Hypervisors

VII. X86 Rings

Different regions of memory on an X86 CPU execute with varying levels of privileges. To denote this, the concept of "ring versions" is used, with a lower ring indicating a higher privilege level. For example, an operating system kernel typically runs at a higher privilege level (ring 0) than a program running in user-mode inside of that operating system (ring 3). This concept is important to security as it prevents user-mode applications from reading and writing to kernel memory regions and executing privileged instructions. If this was allowed, user-mode code could escalate privileges to ring 0 with disastrous effects such as gaining direct hardware access and the ability to break process isolation.

These privilege levels are enforced in hardware through the use of page table entry fields. If a user-mode program decides it wants to read or write some memory that is running in ring 0 (kernel space), the CPU will throw a fault [8]. Ring 0, however, is not the highest privilege level available on an X86 CPU. Since the kernel is not the highest privileged code running on the CPU, additional rings were added to allow code to run with higher privilege than the kernel. One main use case for this was running a hypervisor as it naturally had to have higher privilege than the guest

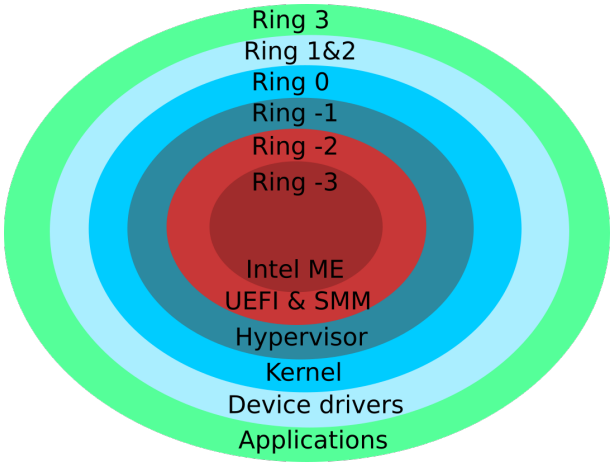

Fig. 3. X86 Privilege Rings

operating system it was virtualizing. In order to accomplish this, the -1 ring was added to the topology. Our solution to stopping stack-smashing attacks, StackSupervisor, will leverage this additional privilege level to protect a guest operating system kernel from exploitation. Because we are able to run at a higher privilege than the guest kernel, we are able to persist in monitoring/protecting the guest kernel even if it crashes or is subjected to a kernel-level exploit. When properly implemented, a hypervisor can remain protected from tampering and modification from a compromised guest operating system in ring 0.

A. *VT-x/AMD-V*

VT-x is Intel's technology that allows an unmodified guest operating system to run on the actual CPU without emulation. AMD has their own version of a very similar technology called AMD-V, however we will be working with Intel's VT-x in this paper.

Prior to VT-x, virtualization was sub-optimal and some clever tricks had to be employed to virtualize a CPU. One popular technique was called binary translation, and effectively would translate privileged instructions (binary translate) on the fly so that an unmodified guest operating system could run [9]. When privileged instructions were encountered, the hypervisor would replace them with a call into the hypervisor which would then handle the operation on behalf of the guest. This provided the isolation and security a virtual machine needed, however it was not very efficient due to the dynamic rewriting of code. To improve upon this, Intel came up with their VT-x technology which allowed true hardware virtualization on their CPUs. This new technology introduced the concept of another privilege level, higher than what the operating system kernel would run at, dubbed "ring -1".

VT-x allows an unmodified guest operating system to run directly on the CPU itself, while providing isolation and protection for the rest of the CPU and hypervisor,

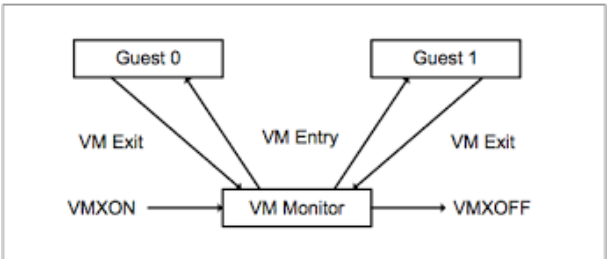

Fig. 4. VT-x State Machine

as well as letting the guest think it is running with full privileges. When VT-x is being used, there are two modes the processor can be in, root mode and non-root mode. Root mode corresponds to the privileged level the virtual machine monitor (VMM) is running at, and the non-root mode non-privileged level is what the guest operating system is executing at.

The following steps and figure 4 roughly outline the life cycle of a hardware virtual machine (HVM) using VT-x [10] [11]:

1) Enable VMX in the CR4 register
2) Initalize VMCS region and run VMXON
3) Write VMCS fields with VMWRITE
4) Start guest with VMLAUNCH
5) Continue guest execution until it exits to VMM
6) Read reason from VMCS via VMREAD
7) Resume execution via VMRESUME or terminate via VMXOFF

The virtual machine control structure (VMCS) contains the configuration for the guest operating system. Inside the VMCS, specific privileged instructions can be allowed/disallowed, register values are set and stored for VMENTER/VMEXIT operations, and various other fields are available for the VMM to set/monitor.

In order to maintain the isolation of the guest operating system, various instructions are not allowed to be executed by the guest directly. Examples of this are reading or writing the paging registers (CR0, CR3), port I/O instructions (IN/OUT), modifying the IDT and GDT, reading the CPU timer, external interrupts, reading unauthorized memory locations, and tamping with the VMX instructions [12]. When the guest operating system decides it would like to perform one of these privileged instructions, it exits into the hypervisor who emulates the instruction carefully and safely before returning execution to the guest operating system.

Another concern for guest isolation is direct memory access (DMA). To combat this, Intel accompanied VT-x with another technology VT-d or IOMMU which is another technology that prevents memory permissions from being circumvented via DMA. VT-d can be configured to restrict which memory regions can be accessed via DMA to prevent

malicious behavior.

B. Hypercalls

Similar to how userspace programs can execute a *syscall(2)* to change privilege level and invoke the kernel by trapping into it, guest operating systems can do the same to their hypervisor via a *hypercall*. This mechanism is used across several hypervisors such as KVM [13], XEN [14], and Hyper-V [15]. Each hypervisor has their own set of hypercalls that they support making the interfaces non-portable across different platforms. The corresponding X86 instruction to invoke a hypercall is *VMCALL* [16] and can be invoked from the guest operating system kernel or userspace, and traps into the hypervisor when executed. In KVM, hypercalls are managed inside the function

```
int kvm_emulate_hypercall(struct
    kvm_vcpu *vcpu)
```

in the file *arch/x86/kvm/x86.c*. The calling convention involves placing the hypercall number inside of the *EAX* register. Definitions of the hypercall numbers KVM supports are contained in the header file *include/uapi/linux/kvm_para.h*. There are alternative methods for the guest to transmit information to the hypervisor such as memory-mapped IO (MMIO) and port-mapped IO (PIO), but hypercalls were chosen due to the simplicity and ease of use.

C. VMENTER/VMEXIT

VMENTER and VMEXIT corresponds to when the CPU switches from non-root to root mode, or in and out of the guest operating system. This can occur for many reasons besides a hypercall. Privileged instructions such as the ones outlined above in the VT-x section, as well as many other reasons [12] all can cause a VMEXIT to the hypervisor. Since it is in the best interest of the VMM to maintain isolation of the guest, these instructions are trapped and then emulated by the the VMM. The hypervisor can also configure some of the instructions that are privileged that it wants to allow the guest operating system to execute via the VMCS fields [12]. It is also part of the normal execution process for the VMX preemption timer to expire and fire a VMEXIT back into the VMM so that virtual machines can be scheduled. Unfortunately these VMEXIT/VMEMTER operations are costly and are similar to a context switch by the kernel when compared to user-mode binaries. When a VMEXIT occurs, the CPU has to store all of the current registers into the VMCS, as well as the reason for why the exit occurred [11]. One benchmark at AnandTech shows that the round-trip time for a VMCALL to VMRESUME can average about 400ns on a modern processor [17].

VIII. SOLUTION

In order to create a robust solution to the stack smashing problem, we propose leveraging the higher privilege level of a hypervisor to monitor the stack frame for overflows in select functions during kernel execution. Since the hypervisor has complete control over the guest operating system, it is able to inspect all of its memory including the stack. By instructing the protected program to notify the hypervisor of a function enter/exit, the hypervisor can inspect/record the stack after entering then validate the return address before exit. Additionally, this allows the hypervisor to not only detect that a stack smash has occurred, but also replace the proper return address on the stack and heal the program without having it crash.
To do this, an LLVM-pass has been created to insert the following assembly stub before the *RET* instruction and immediately after the stack frame has been setup in every function. The stub has been written to minimize the overhead of both increased code-size and reduced performance when doing the stack check. The effectiveness of this is demonstrated in the performance evaluation section.

```
mov $0xb, %eax
mov $0x1, %ebx  ;0x2 for function exit
vmcall
```

In the above snippet, the first line loads in our custom hypercall number *0xB* into the EAX register. The second line denotes that we are entering a function (this stub would go at the beginning of a function) by loading *0x1* into the EBX register. The value 0x1 would be substituted with 0x2 at the end of a function to signify the end of a call. Finally, the last line invokes the hypercall and traps into the hypervisor where we handle the request.

LLVM-passes provide the ability to register callbacks during program compilation with LLVM, which allow the modification of the generated code. By registering a callback for each function in the source code, we can traverse the instructions inside and insert our stub in the proper location. In order to prevent all functions from being guarded, we have introduced an attribute that can be used on functions that the programmer would like to protect. Since adding the stack protection does induce overhead, this feature allows the developer to only select functions that might be vulnerable such as operating on tainted input.

```
__attribute__((annotate("StackSupervisor")))
```

For a specific example of how StackSupervisor works, assume we have a simple C program with the following source code:

```
__attribute__((annotate("StackSupervisor")))
 ↪ int main() {
        char buf[10];
        gets(buf);
    }
```

Compiled to assembly, we have the following before the LLVM-pass:

```
00000000004004d0 <main>:
  4004d0: push   %rbp
```

```
  4004d1: mov    %rsp,%rbp
  4004d4: sub    $0x10,%rsp
  4004d8: lea    -0xa(%rbp),%rdi
  4004dc: mov    $0x0,%al
  4004de: callq  4003d0 <gets@plt>
  4004e3: xor    %ecx,%ecx
  4004e5: mov    %eax,-0x10(%rbp)
  4004e8: mov    %ecx,%eax
  4004ea: add    $0x10,%rsp
  4004ee: pop    %rbp
  4004ef: retq
```

After the LLVM-pass with our inserted guard code:

```
00000000004004d0 <main>:
  4004d0:        push   %rbp
  4004d1:        mov    %rsp,%rbp
  4004d4:        sub    $0x10,%rsp
  4004d8:        mov    $0xb,%eax
  4004dd:        mov    $0x1,%ebx
  4004e2:        vmcall
  4004e5:        lea    -0xa(%rbp),%rdi
  4004e9:        mov    $0x0,%al
  4004eb:        callq  4003d0 <gets@plt>
  4004f0:        mov    $0xb,%eax
  4004f5:        mov    $0x2,%ebx
  4004fa:        vmcall
  4004fd:        xor    %ecx,%ecx
  4004ff:        mov    %eax,-0x10(%rbp)
  400502:        mov    %ecx,%eax
  400504:        add    $0x10,%rsp
  400508:        pop    %rbp
  400509:        retq
```

We can see that two stubs have been inserted at the beginning and end of the function. After *gets()* returns, we have a check to see if the return address on the stack has been corrupted before returning to it. At both of the *VMCALL* instructions, the code traps into KVM via the hypercall interface and we can use various routines inside KVM to inspect the guest's state. To read the registers, we can invoke the KVM function

```
long kvm_register_read(struct
 ↪ kvm_vcpu *vcpu, enum kvm_reg
 ↪ reg)
```

To read the guest memory (such as the stack), we can call

```
int kvm_vcpu_read_guest(struct
 ↪ kvm_vcpu *vcpu, gpa_t gpa, void
 ↪ *data, unsigned long len);
```

Armed with the values of the *ESP* register which points to the stack and the *EBP* register which points to the base of the stack, we can then load this address range into the KVM function *kvm_vcpu_read_guest()* and read the guest's stack. Computing the return address on 32-bit programs is easy and is at *ebp - 4*. Using this, we can read out the stack as soon as a function as entered, and extract the return address and keep a copy of it. On the subsequent exit before the *RET* instruction, we will trap back into our hypercall and do the same process, comparing the return addresses. If they differ, we can choose to kill the guest OS, generate some sort of alert, or attempt to heal the stack by replacing the original return address onto the stack and resuming execution. In order to keep track of recursive or nested function calls, a stack data structure can be used with each function call resulting in the hypervisor extracting the return address and storing it on a stack. Subsequent returns would pop off the appropriate return address for comparison in a LIFO manner.

As described thus far, StackSupervisor's protection applies to a guest operating system running in kernel mode. With some additional work, however, this technology could be extended to user-space programs as well if necessary. The primary issue in doing this is that this would likely be highly dependent on the guest operating system. Since the hypervisor does not really have a concept of processes running inside of the guest, a bridge of some sort would have to be built between the guest kernel and the hypervisor. One solution that would work with the Linux operating system is as follows:

1) Instrument user-mode binary with a similar LLVM-pass that inserts a *syscall()* to a custom syscall in the kernel
2) The custom syscall then records the PID of the process into a CPU register then invokes a custom hypercall via *vmcall*
3) Hypervisor records the stack similar to how it does for kernel-mode protection, pairing the return address with the PID communicated from the guest kernel
4) Upon function exit, the same process is followed syscalling into the kernel, passing the PID into the hypervisor who then performs the check based on the PID
5) If the return address differs, the OS can be informed to kill the process or simply alert on the overflow

IX. ENVIRONMENT

To implement the proposed solution, we will be working with a custom 32-bit kernel running inside of our modified hypervisor, KVM. Since KVM is part of the Linux kernel, we will be running Debian 9.8.0 with the Linux kernel 5.0.8 (latest stable at time of research). In order to prevent bugs in our KVM patch from kernel panicking our development machine, the Debian environment was run as a virtual machine itself using VMWare and enabling the VT-x pass-through option which allows the guest operating system (Debian) to act as a hypervisor using VT-x/nested virtualization. The hardware we were using was an enterprise Intel Xeon E5606 CPU @ 2.13 GHz.

As for the hypervisor, KVM was chosen due to the project being open-source as well as being easy to use and modify. Xen was another contender for the hypervisor, however it was found to not be as reliable and easy to use as KVM.

The 32-bit operating system being run was a very minimal kernel that could write characters to the video buffer and only had a couple functions inside of it. StackSupervisor was also tested on a minimal 16-bit kernel to make sure it worked outside of 32-bit mode.

X. EXPLOITATION PREVENTION

In order to evaluate the effectiveness of the StackSupervisor protection mechanism against return address corruption, we tested a classic buffer overflow bug in our 32-bit kernel. Taking a local variable inside of a StackSupervisor function, we overran the buffer with a *memcpy(3)*, corrupting the return address with *\xAA\xAA\xAA\xAA*. Upon returning from the function, the *RET* instruction attempted to set *EIP* to *0xAAAAAAAA* which was an invalid region of code, and we can observe that QEMU has crashed.

```
KVM internal error. Suberror: 1
emulation failure
EAX=00000000 EBX=00000002 ECX=000b8000
EDX=00000007 ESI=00000000 EDI=0010a000
EBP=aaaaaaaa ESP=00104ff0 EIP=aaaaaaaa
EFL=00010006 [-----P-] CPL=0 II=0 A20=1
SMM=0 HLT=0
```

Looking at the StackSupervisor output, we observe the following messages:

```
Stackguard Function Enter
RIP=0x100313 RSP=0x104fd0 RBP=0x104fe8
Return address: 0x100389

Stack Dump:
20 00 00 00 00 00 00 00
07 00 00 00 CF 07 00 00
50 00 00 00 19 00 00 00
08 50 10 00 89 03 10 00

Stackguard Function Exit
RIP=0x100343 RSP=0x104fd0 RBP=0x104fe8
Return address: 0x100389
Ret addr OK!

Stack Dump:
FC 03 10 00 15 00 00 00
07 00 00 00 CF 07 00 00
FC 03 10 00 00 00 00 00
08 50 10 00 89 03 10 00

Stackguard Function Enter
RIP=0x1003c0 RSP=0x104fe0 RBP=0x104fe8
Return address: 0x100396

Stack Dump:
3C 19 36 FE 07 00 00 00
08 50 10 00 96 03 10 00
FC 03 10 00 00 00 00 00
00 00 00 00 00 00 00 00

Stackguard Function Exit
RIP=0x1003f4 RSP=0x104fe0 RBP=0x104fe8
Return address: 0xffffffffaaaaaaaa
Return address not equal!!
0x100396 != 0xffffffffaaaaaaaa

Stack Dump:
14 00 00 00 07 00 00 AA
AA AA AA AA AA AA AA AA
AA AA AA AA AA AA AA AA
AA AA AA 00 00 00 00 00
```

We can see that there first is a function that is called that returns properly (return address is checked and remains the same), followed by a function that has our buffer overflow bug. With the function that gets overflowed, we can see that the lower 32 bits of return address has been corrupted as intended earlier and StackSupervisor detects this, telling us it is not the expected return address which is *0x100396*. Looking further at the "Stack Dump" snapshot, we can clearly see that the buffer has been overrun with *0xAA* bytes, and the return address corrupted.

XI. PERFORMANCE

A. Measuring Performance

On Intel x86 CPUs, there is an instruction called *RTDSC* which allows you to read the current value of the CPU's time-stamp counter [18]. This is useful because we do not have to rely on any operating system code to get the current time. In addition, the timer is extremely precise, giving us the number of ticks that have elapsed since boot. In order to make sure that the value has been calculated properly after data buffers have been flushed in in the CPU, as well as all preceding instructions have been executed, the instructions *LFENCE* and *MFENCE* were used [18]. Using this timer, we can grab the current number of ticks elapsed, run our function we wish to benchmark, and then calculate the difference and get the execution time of the code in between.

```
inline unsigned long get_ticks(void) {
        unsigned long hi, lo;

        asm volatile ("mfence\n"
                      "lfence\n"
                      "rdtsc" :
                      "=a" (lo),
                      "=d" (hi));
        return lo; //Only need low
         ↪ 32-bits
}

unsigned long time = get_ticks();
```

```
prot_function_test(); //StackSupervisor
 ↪ instrumented
unsigned long total = get_ticks() -
 ↪ time;
```

In addition to using the CPU tick counter, the Linux *perf(3)* [19] utility was used with the KVM backend to measure the time the *VMCALL* instruction took.

B. Data

Following the code above, the following data was collected for measuring the time to execute a StackSupervisor guarded function. The units are in CPU ticks.

StackSupervisor	Without StackSupervisor	Overhead
191932	380	191552
176032	400	175632
174476	280	174196
175204	284	174920
175912	400	175512
	Average (ticks)	
178711	348	178363

The results from the Linux *perf(3)* utility are as follows:

Samples	Min Time	Max Time	Avg Time
55	7.10us	35.65us	22.49us

I expect there to be some mild level of performance impact due to the addition of instructions. Furthermore, when a hypercall is invoked, the CPU has to perform a *VMEXIT* into the hypervisor which runs the hypercall, then switches back with a *VMENTER* call. This is costly as many caches get invalidated due to this switch, and such *VMENTER/VMEXIT* calls have a measurable impact on performance. As I finish up the project I intend to benchmark the finished solution and provide graphs and metrics for performance here.

XII. ISSUES

While StackSupervisor is effective in protecting stack-based attacks, it does not protect from exploits which corrupt local variables, such as function pointers, to gain arbitrary code execution. Since an attacker could carefully overflow a buffer just enough to corrupt a function pointer but not overwrite the return address. This type of overflow would not be detected since the return address would not be overrun which is what StackSupervisor is checking. In order to protect from this, some alternative method such as placing canaries on the boundaries of buffers would have to be used.

XIII. CONCLUSION

In conclusion, StackSupervisor has proven to be a successful tool in detecting stack overflows and preventing arbitrary code execution. In comparison to other existing stack protection technologies, StackSupervisor offers more rigorous protection to programs by checking their return address directly - a significant limitation amongst the other solutions.

With regard to performance, StackSupervisor does have overhead of, on average, 178363 CPU ticks on an Intel Xeon E5606 CPU. Although this does amount to a somewhat large number, the fact that protection can be turned on/off via a compiler flag allows a programmer to select the functions they are interested in protecting, reducing the overall overhead of StackSupervisor. Furthermore, it is worth nothing that StackSupervisor included lots of debugging information such as logging to DMESG as part of the hypercall routine. Should this have been stripped in a production setting, it is quite likely that the overhead from the protection instrumentation would significantly decrease. The reason the instrumentation incurs as performance penalty in the first place is due to the addition of instructions. In addition to this, when a hypercall is invoked, the CPU has to perform a *VMEXIT* into the hypervisor which runs the hypercall, then switches back with a *VMENTER* call. This is costly as many caches get invalidated due to this switch, and such *VMENTER/VMEXIT* calls have a consequently measurable impact on performance.

XIV. SOURCE CODE

The source code for this project is available at https://github.com/gfoudree/HypervisorStackGuard under the GPLv3 license.

REFERENCES

[1] "X86 calling conventions," Mar 2019. [Online]. Available: https://en.wikipedia.org/wiki/X86_calling_conventions

[2] "Stack frame layout on x86-64." [Online]. Available: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/

[3] "A detailed description of the data execution prevention (dep) feature in windows." [Online]. Available: https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in

[4] "Linux syscall reference." [Online]. Available: https://syscalls.kernelgrok.com/

[5] InVoLuNTaRy, "Performing a ret2libc attack." [Online]. Available: http://shellblade.net/docs/ret2libc.pdf

[6] "Lab 08 - exploit protection mechanisms," Nov 2018. [Online]. Available: https://ocw.cs.pub.ro/courses/cns/labs/lab-08

[7] "Address space layout randomization," Mar 2019. [Online]. Available: https://en.wikipedia.org/wiki/Address_space_layout_randomization

[8] "Exceptions." [Online]. Available: https://wiki.osdev.org/Exceptions#General_Protection_Fault

[9] "Introduction to virtual machines." [Online]. Available: http://www.cs.columbia.edu/~nieh/teaching/s4118_09/lectures/Columbia08.pptx

[10] Shahrir and Sina, "Hypervisor from scratch part 2: Entering vmx operation," Sep 2018. [Online]. Available: https://rayanfam.com/topics/hypervisor-from-scratch-part-2/

[11] S. Maresca, "Vm security." [Online]. Available: http://www.kiayias.com/compsec/CSE4707_Computer_Security/Reading_files/VM-security.pdf

[12] LordNoteworthy, "Lordnoteworthy/cpu-internals," Apr 2019. [Online]. Available: https://github.com/LordNoteworthy/cpu-internals

[13] Torvalds, "torvalds/linux." [Online]. Available: https://github.com/torvalds/linux/blob/master/Documentation/virtual/kvm/hypercalls.txt

[14] XENProject. [Online]. Available: http://xenbits.xen.org/docs/unstable/hypercall/x86_64/index.html

[15] Microsoft, "Hypervisor specifications." [Online]. Available: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/tlfs

[16] F. Cloutier. [Online]. Available: https://www.felixcloutier.com/x86/vmcall

[17] J. D. Gelas, "Hardware virtualization: the nuts and bolts," Mar 2008. [Online]. Available: https://www.anandtech.com/show/2480/9

[18] F. Cloutier, “Rdtsc read time-stamp counter.” [Online]. Available: https://www.felixcloutier.com/x86/rdtsc
[19] “Perf events.” [Online]. Available: https://www.linux-kvm.org/page/Perf_events