VB2020 localhost

30 September - 2 October, 2020 / vblocalhost.com

# SHE SELLS ROOT SHELLS BY THE C(++) SHORE

**Costin Ionescu**

Broadcom, USA

costin.ionescu@broadcom.com

## ABSTRACT

The security industry is well aware of the importance of delivering secure solutions as part of their software offering given that their applications run with elevated privileges. Sometimes, even with the best intentions and carefully thought out precautions, blunders still happen.

This highlights the importance of a systemic approach that minimizes the impact of human (or, soon, AI overlord) mistakes.

In this paper we go over some of the existing approaches for hardening software, looking at hardware features, operating system support, software sandboxing techniques and programming language constructs designed for safety.

We then discuss the approach that the security engines team in the *Symantec Enterprise Division* of *Broadcom* took a few years back to secure engine releases. We'll describe how clang/LLVM instrumentation is used to produce hardened binaries which dramatically reduce the risk of remote code execution, denial of service and other attacks, by severely mitigating the impact of bugs such as buffer overflows, unbounded recursion (stack exhaustion) and infinite loops.

## EXIGENCES OF SECURITY SOFTWARE

When we think of operating environments for security software solutions we usually traverse the spectrum from endpoints to gateways, appliances on premises or in the cloud, and end with backend systems.

In general, the discoverability of vulnerabilities is in direct proportion to the exposed software surface, which makes the endpoint a particularly tough cookie. Of course, a vulnerability for a product targeting any part of the spectrum is just as serious.

A common requirement for security software is the ability to rapidly update the logic used for detection, be it threat detection or event reporting. Regardless of on which side of the 'AV is dead' fence people find themselves, the need for updating the content is seen for a wide range of content – from AI models to blocklists, string signatures or logic updates in algorithmic detections.

The team I'm part of deals with delivering scanning engines to our products for the entire spectrum mentioned above and enables updates to these engines to be accepted several times a day. Quite often these updates are not simple strings or hash signatures added to a database but rather the implementation of an API in the x86 emulator or a built-in method in the script emulators to improve behavioural detections.

Since these updates involve new code to execute on a wide range of systems, there are strict requirements for the quality of the new content. Even with a high bar set for quality, frequent updates mean there are many chances for something to go wrong, thus hardening the runtime environment is of utmost importance in order to limit unwanted negative effects.

## HARDENING

This brings us to a short review of existing approaches used for hardening software.

There are many aspects that can be used in creating a more robust and better contained application, ranging from hardware features to OS facilities, compiler code generation options and even the choice of programming language or language subset to use.

The first line of defence is a concept common to all operating systems – the process. Each process has its own address space thus naturally guarding against accidental changes to other unrelated applications.

### Inter-process hardening

Processes are usually grouped inside security boundaries (sessions, effective user ID, etc.) to allow interactions to take place. Especially in the context of a security solution, we want to protect core parts of the application from external tampering. For an endpoint product the UI needs to be in the context of the user using the desktop but most of the scanning logic does not need to be in that context. The better control we have over this boundary, the better the security.

Hardware virtualization offers a good opportunity to establish a barrier to prevent tampering. Virtual machines are great for isolating multiple services running together but sometimes we need something more suitable for isolating a single application. That led to forms of lighter virtualization, such as the *Windows 10* VSM feature (Virtual Secure Mode), which allows the creation of IUM (Isolated User Mode) processes.

Containers are another great way of keeping the security solution in its own trust zone with less risk of tampering. Hardware virtualization enables many guarantees to be offered by containers, although even without it there are effective mechanisms made available by operating systems (chroot in all Unixes, cgroups in *Linux*, rfork in *BSD*).

### Process level hardening

The overwhelming success of the von Neumann architecture for most computing systems in almost a century is in big part due to its amazing simplicity – simple design, simple rules, simple circuits. That also brings some simple problems to deal with… and, well, some not-so-simple problems too! Code and data are in the same memory which, granted, makes it very easy to keep telling the computer what to do… too bad humans seem to be notoriously poor at telling computers how to do what they really need them to do! Furthermore, even when we identify a bug in code we have the tendency to underestimate the impact it can have – after all, how can writing a single byte over the end of the buffer cause remote code execution? And why is 0x41414141 [1] so funny?

Even though some architectures adopted early restrictions on treating memory blocks as either code or data it took many years to achieve widespread adoption following the introduction in commodity hardware by *AMD* in 2004 with the NX feature [2]. While NX solved the simple 'jump to payload' type of exploits, it led to the rise of ROP (return oriented programming – formerly referred to as ret-to-libc [3] in the Unix world).

Continuing the exploitation mitigation battle, an important step was made by operating systems in adopting ASLR [4] (address space layout randomization). This feature requires the introduction of enough entropy in establishing the layout of memory inside processes to make it hard to achieve reliable exploitation.

Another significant improvement came in the form of compiler support for stack cookies (/GS or -fstack-protector), which significantly raised the bar for exploitation of stack overflows.

*Windows* employs a different exception model compared to other operating systems, namely SEH [5] (Structured Exception Handling) with a post Itanium addition – VEH [6] (Vectored Exception Handling). SEH typically uses the stack (on x86) and is therefore subject to all stack-related types of exploitation, which prompted *Microsoft*'s valuable mitigation in the form of SafeSEH [7] and SEHOP [8] (SEH Overwrite Protection).

In the Unix world, the dominant executable format is ELF which, unlike *Microsoft*'s PE format, does not allow relocations in read-only sections. Coupling this with the use of PLT and GOT [9] creates the opportunity for exploitation by overwriting a single pointer in these tables. GNU's C compiler and dynamic linker introduced a very effective mitigation for this type of exploitation, called RELRO [10] (read-only relocations).

All the features mentioned above are commonly available and it is therefore good practice to enable them through relevant compiler options or runtime opt-in APIs.

Now we can turn our attention to newer technologies that are not so common due to limited hardware or operating system support. As more systems start to support the technologies it is likely we'll see an increase in adoption, although in some cases they also spark the debate of performance vs. safety.

To combat exploitation that relies on out-of-bounds overwrites of function pointers, *Microsoft* introduced CFG [11] (Control Flow Guard), which requires the application to be compiled with the required hardening instrumentation turned on (/guard:cf) and a little help from the operating system to enforce runtime checks. The idea is to have every indirect call instrumented to go to a stub which checks if the call target is in a sparse bitmap of allowed targets. This implies a small performance cost, which constitutes a barrier for adoption. Some important limitations of the technology are around dealing with loaded modules that are not CFG-enabled (which become trees with juicy low-hanging ROP gadgets) and the way functions with unaligned addresses can be treated (due to the need to reduce the bitmap size). Nonetheless, CFG significantly increases the difficulty of exploitation when employed appropriately.

Looking at architectural features meant to lower the performance impact of software mitigations, an interesting attempt was *Intel* MPX [12] (Memory Protection Extensions), which was introduced in 2015 with the Skylake generation of processors. The intent was to address exploitation of buffer overflows simply by recompiling legacy C/C++ applications to use MPX-specific instructions for performing bound checking based on information inferred from the compilation stage. This entailed a massive effort to design and include in silicon such a non-trivial extension and to add support in ICC and GCC. All that came to an abrupt stop in 2018 when GCC removed support for MPX, the reason being a mix of performance, memory consumption and efficacy when compared to other instrumentation techniques, in particular ASan [13].

*Intel* MPK (Memory Protection Keys) is a hardening feature that brings the ability to change permissions for memory ranges without altering page tables (which impose significant performance penalties). This allows interesting uses, such as having components protect the structures they manage from accidental or malicious tampering from other code in the same process. Another use is marking code as XOM (executable-only memory), preventing dynamic gadget lookup for JIT-ROP attacks.

Another technology mostly meant to provide privacy is SGX [14]. It targets components that deal with secrets like cryptographic key manipulation or DRM-type of workloads. However, this opens up the opportunity of using this extension for the purpose of partially sandboxing code since exceptions there are trapped and exposed through a single exit point.

A more recent *Intel* extension tackles the ability to tamper with the call stack where function returns end up being attacker-controlled (most notably in ROP exploits). This feature, named CET [15] (Control-flow Enforcement Technology), requires hardware and operating system support in order to manage a shadow stack containing only the call stack and alters the function of standard opcodes like CALL and RET. The purpose is to maintain the call stack and to fault when encountering discrepancies between the shadow stack and the return addresses from the traditional stack.

### In-process management of multiple security domains

Applications that support plug-ins or extensions have to deal with defining the security restrictions that apply to the core components of the application versus the pluggable modules. This model is often encountered in the context of security solutions where important functionality is enabled as plug-ins delivered through updatable channels. However, even when full trust is assumed between the hosting component and the plug-ins, additional guardrails are desired to prevent one plug-in from taking down the entire application.

A more challenging case is when third-party plug-ins need to be fully isolated from each other and from the core application. This is encountered in the browser world where security boundaries are defined around each domain of each browser tab and fast sandboxed execution of arbitrary third-party code is desired.

One of the efforts that brought a lot of attention to the idea of using statically typed languages to target a sandboxed environment is *Google*'s NaCl project [16] (Native Client). This technology uses a modified GCC toolchain to produce instrumented binaries. The binary format enables easy verification of certain rules such that sandbox escapes are prevented. These instrumented binaries containing native code are then delivered to the target platform. Serving platform-specific binary code was not well received in the context of the 'open web'. This led to the second iteration, dubbed PNaCl (Portable Native Client), where the same portable bitcode executable is delivered and compiled ahead of time for the target by the browser before execution. PNaCl uses the LLVM compiler infrastructure for generating binaries and requires a large hosting component responsible for generating the executable code ahead of time. Aside from the sandboxing capabilities, a great advantage of NaCl/PNaCl is the near-native speed, which makes it very attractive for delivering a safe environment for securely running plug-ins.

PNaCl did not gain traction [17] with *Chrome*'s competitors and we saw the asm.js [18] project gaining a bit more popularity due to its approach of trying to get code compiled into a JavaScript subset guaranteed to be understood by all browsers. The performance of asm.js compiled code, although significantly lower than NaCl, can be accelerated by browser-specific optimizations. Similar to PNaCl, asm.js target code is usually compiled using an LLVM derived toolchain – emscripten in this case.

A more recent effort, coordinated by *Mozilla*, is WebAssembly – a successor in spirit of asm.js. LLVM is used to compile code from C, C++ and Rust to a portable executable format that all major browsers agreed to understand. WebAssembly offers a clear, well defined virtual architecture which allows a relatively straightforward JIT engine implementation to support execution (the use of the word 'straightforward' next to JIT is bound to raise some chuckles).

Given its growing popularity, WASM is a very attractive technology for delivering arbitrarily complex sandboxed functionality.

## SAFETY FEATURES OF PROGRAMMING LANGUAGES

All technologies described above attempt to contain the potential damage a vulnerability can cause but do not tackle the source problem – the presence of vulnerabilities.

While the C language carries the blame for enabling most common types of vulnerabilities it also wears the crown for lifting programming from machine-specific assembly to low-level portable code.

C++, with its promise to stay mostly compatible with C, allows most of the same mistakes to be made. However, its higher abstractions make it easier to use constructs that prevent some of the common C mistakes. Among the C++ idioms and features most often mentioned for helping with code safety are RAII [19] (resource acquisition is initialization) and smart pointers (unique_ptr, shared_ptr; not the first incarnation fiasco auto_ptr). The many benefits of C++ are countered by the sheer magnitude of language surface with many intricacies that sometimes surprise even experienced C++ programmers. C++'s amazing track record of backward compatibility combined with the three-year cadence for new features often translates into challenging maintenance of long-lived projects.

Using such safety-challenged languages like C or C++ nowadays is considered dangerous without the use of modern compiler features and techniques during the development cycle such as:

- Warnings-as-errors and high warning level during compilation
- Static code analysers
- Code coverage
- ASan (address sanitizer) instrumentation
- Refactoring tools like clang-tidy to lift outdated constructs into following current best-practice idioms
- Fuzzing (libFuzzer [20], AFL [21]).

While all the above have the merit of raising the bar, the fact that we can obtain a binary subject to memory corruption and concurrency issues means some of these vulnerabilities will slip through.

A relatively new language that is designed around the idea of preventing memory corruption and concurrency issues is Rust [22]. One of Rust's core ideas is that aliasing and mutability are mutually exclusive. This restriction dramatically changes the way an application is designed and implemented, usually guiding the development through a long set of compilation errors. Coupling this core rule with enforcing 'no undefined behaviour allowed' eliminates the risk of exploitation for a large set of vulnerability classes.

Given its focus on safety, Rust should be at the top of the list when we consider adding new components to our security solutions.

Rewriting an entire application in a different language represents a major barrier for adoption of the new language. This is why Rust offers several features to make it somewhat easier to integrate Rust code in an existing application. Interacting with C interfaces is supported through Rust's FFI [23] constructs. Calling into C++ code must be done by exposing extern 'C' linkage functions.

Equally important is the ability to call Rust from C/C++, which can be done by using the system ABI for interface functions [24].

A common misconception about its safety is that Rust applications do not crash – this is inaccurate. Operations that cannot be proved at compile time to be safe and do not violate the language rules end up being compiled with implicit runtime checks that panic [25] on error. For instance, using an external input to index an array can only be done safely if the index is checked at runtime to be within bounds.

As mentioned earlier, a key difference compared to C/C++ is the very different stance on undefined behaviour. In Rust something that has the potential to cause undefined behaviour gets a runtime check that forces the application to abort before the undefined behaviour can take place. A well written, bug-free application would not trigger these implicit panic calls but if it does contain a bug the risk of exploitation is virtually eliminated by having a controlled takedown.

This brings a valid concern when mixing legacy C/C++ with Rust, as crashes can still happen on insufficiently high-quality Rust code. Luckily, Rust offers a very powerful mechanism to recover from faulty Rust code logic without necessarily bringing the process down through the catch_unwind [26] mechanism.

## CP3: A SAFE EXECUTION ENVIRONMENT

One constant challenge that we face in the static scanning technologies team is around providing algorithmic detection capabilities as updatable content without requiring changes to the engine side. Over the decades we had engines that included various interpreters to allow custom logic but they had a domain-specific language, were very limited, and the performance was far from native. A more flexible in-house solution gave us the benefit of writing code in a subset of C++, but even that was tens of times slower than similar functionality compiled natively. As the threat landscape evolved we found ourselves needing to generate more and more custom detection logic content and the limitations of existing technologies required a careful balancing act between speed and flexibility.

The significant growth of code size in order to effectively tackle emerging threat vectors also brings a growing risk of exploitable vulnerabilities.

All these factors constituted a great motivator for developing a solution that limits the attack surface by allowing the transition of most complex logic under the guardrails of a sandboxing solution.

The innovation spree that engulfed the compiler world in the last decades sparked new opportunities, which led us to a pragmatic solution that combines flexibility, performance and safety.

A great merit for triggering the compiler revolution lies with the LLVM project. As usual when humans are involved, there is also a good chunk of politics that goes into the mix, in this case *Apple*'s strong support for LLVM being a direct result of GNU's change of licence from GPL2 to GLP3 for the GCC project. Nonetheless, LLVM's modular design led to its success, bringing also massive opportunities around tooling and code instrumentation.

To create a safe execution environment for dynamic content while minimizing the risks for the stability of the host process we need mainly two ingredients: a way to instrument code modules for containment of potential vulnerabilities and a hosting library to load, execute and enforce the sandboxing rules.

Another design goal is not to introduce a large amount of new code required for key safety features. That tips the balance against having some form of portable executables that get JIT'ed in the host process due to the inherent complexity of the JIT compiler.

### Instrumentation

CP3 project started as an experiment on playing with LLVM's IR (intermediate representation) [27]. The intent was to create an instrumentation layer that allows the compiling of arbitrary C/C++ code and produces a hardened binary guaranteed not to corrupt anything outside an assigned portion of memory in the host process. This would allow execution of such instrumented modules while minimizing the risk of bringing down the process.

LLVM's pluggable compilation process makes it easy to instrument code at various stages and we opted for performing simple transformations of the IR code. This means that all the complexity of the language (C++ in particular) is already dealt with from LLVM's front-end compiler (clang). Similarly, all particularities of the target platform are handled by LLVM's backend. Injecting our instrumentation before the optimization stages still gives us a free ride for obtaining optimized output.

The approach that LLVM takes to be well suited for optimizing code is by having a virtual architecture for the IR with infinite 'registers' operated in SSA [28] form (Single Statement Assignment).

## *Memory safety*

An interesting quality of LLVM's IR is the very limited set of opcodes that interact with memory (load, store, fence, cmpxchg, atomicrmw). If we instrument this handful of opcodes to constrain what memory is accessed we can set a boundary for the in-process sandbox. There are also variations we can consider depending on the level of trust we have for the instrumented binary. For instance, we can restrict only writes but allow reads from anywhere in the process if we want to reduce the overhead of making data available to the instrumented module, or we can choose a more robust isolation layer where reading can happen only from the same sandboxed area.

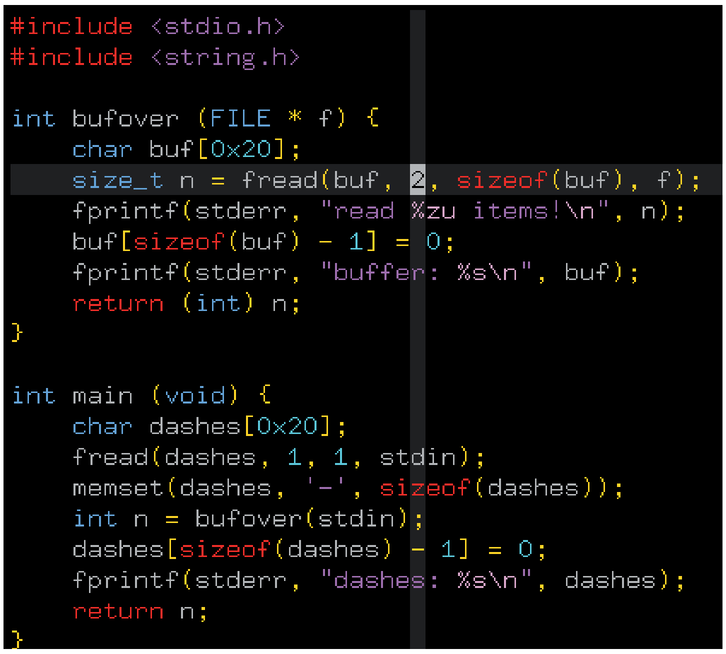
## *Local variables and stack usage*

Since we are merely instrumenting the code in the intermediate representation, we have to abide by the target platform's ABI rules. This implies that we have the real thread stack used during execution. Mutable local variables require writing into the current stack frame and the call stack is just around the stack frame's corner. So if we want to write local variables then it looks like we must move the stack inside the sandboxing zone since all writes are restricted to it. Moving the entire stack would, of course, bring the risk of corrupting the call stack integrity by exposing saved registers and return addresses, defeating the purpose of the sandbox.

Looking again at LLVM's IR language we can observe that all mutable local variables are obtained by using a specific opcode: *alloca*. This property gives us the opportunity to instrument all alloca occurrences for the purpose of selecting items that normally live on the real thread stack and move them somewhere else, namely inside the sandboxing area.

A consequence of this design choice is that typical stack overflows now become overflows in an area separate from where important code flow information is stored (saved registers and return addresses). The native thread stack is never directly manipulated by our instrumented memory writes, but it is still used by the generated code to simulate operating on IR's virtual registers.

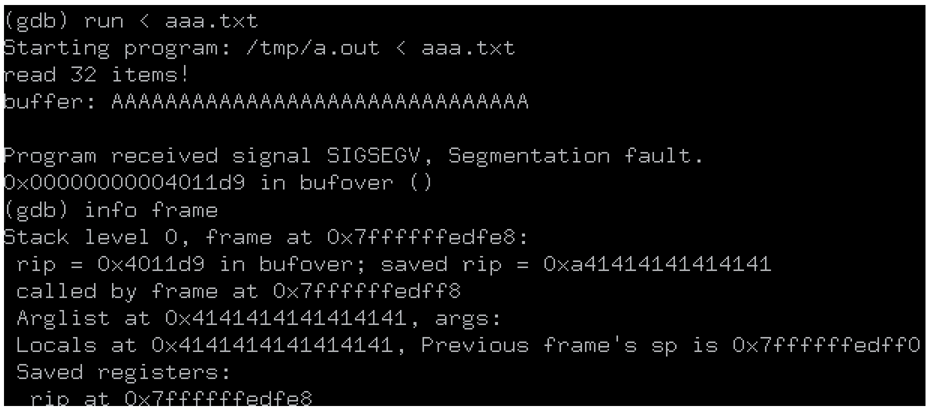
Let's look at a simple example of a program vulnerable to buffer overflow:



```
#include <stdio.h>
#include <string.h>

int bufover (FILE * f) {
    char buf[0x20];
    size_t n = fread(buf, 2, sizeof(buf), f);
    fprintf(stderr, "read %zu items!\n", n);
    buf[sizeof(buf) - 1] = 0;
    fprintf(stderr, "buffer: %s\n", buf);
    return (int) n;
}

int main (void) {
    char dashes[0x20];
    fread(dashes, 1, 1, stdin);
    memset(dashes, '-', sizeof(dashes));
    int n = bufover(stdin);
    dashes[sizeof(dashes) - 1] = 0;
    fprintf(stderr, "dashes: %s\n", dashes);
    return n;
}
```

*Figure 1: Program vulnerable to buffer overflow.*

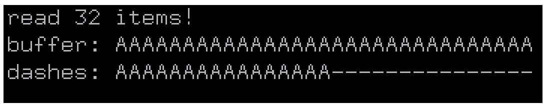
When we debug the program shown in Figure 1 we see what happens on sufficiently large input.



```
(gdb) run < aaa.txt
Starting program: /tmp/a.out < aaa.txt
read 32 items!
buffer: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x00000000004011d9 in bufover ()
(gdb) info frame
Stack level 0, frame at 0x7fffffffedfe8:
 rip = 0x4011d9 in bufover; saved rip = 0xa414141414141414141
 called by frame at 0x7fffffffedff8
 Arglist at 0x4141414141414141, args:
 Locals at 0x4141414141414141, Previous frame's sp is 0x7fffffffedff0
 Saved registers:
  rip at 0x7fffffffedfe8
```

*Figure 2: Debugging shows what happens on large input.*

Figure 3 shows the output when running the unoptimized program under CP3, illustrating how the buffer overflow from one stack frame spills the caller's stack frame without affecting the call stack.



```
read 32 items!
buffer: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
dashes: AAAAAAAAAAAAAAAA---------------
```

*Figure 3: Output when running the unoptimized program under CP3.*

### *Function call restrictions*

In C and C++, from the language point of view, violating ODR [29] (one definition rule) leads to undefined behaviour. In practice, most ABIs allow the passing of a mismatched number of arguments to functions and successful return from those functions and that is allowed mostly from the desire to handle C variadic functions. A notable exception is *Microsoft*'s ABI for x86 where certain calling conventions (stdcall, fastcall) require the callee to clean the stack up, which means a mismatch ends up returning to the caller in an unexpected stack state.

This problem is easier to encounter when using C, as symbols are marginally decorated. One may think that the decorations are enough to guarantee stack consistency (stdcall and fastcall append a number matching the size of a struct containing all function arguments). This is not true, however, if we look at the following example:

```
int __fastcall square (double num) { return num * num; }
int __fastcall mul (int a, int b) { return a * b; }
```

Both functions receive the same decoration but they adjust the stack with a different amount on return. Here is the output using *Compiler Explorer* [30]:

```
_num$ = 8 ; size = 8
@square@8 PROC
movsd xmm0, QWORD PTR _num$[esp-4]
mulsd xmm0, xmm0
cvttsd2si eax, xmm0
ret 8
@square@8 ENDP

@mul@8 PROC
imul ecx, edx
mov eax, ecx
ret 0
@mul@8 ENDP
```

Apart from stability issues, calling functions with mismatched prototypes still represents undefined behaviour, which gives the compiler the right to do anything if it detects it… including calling abort.

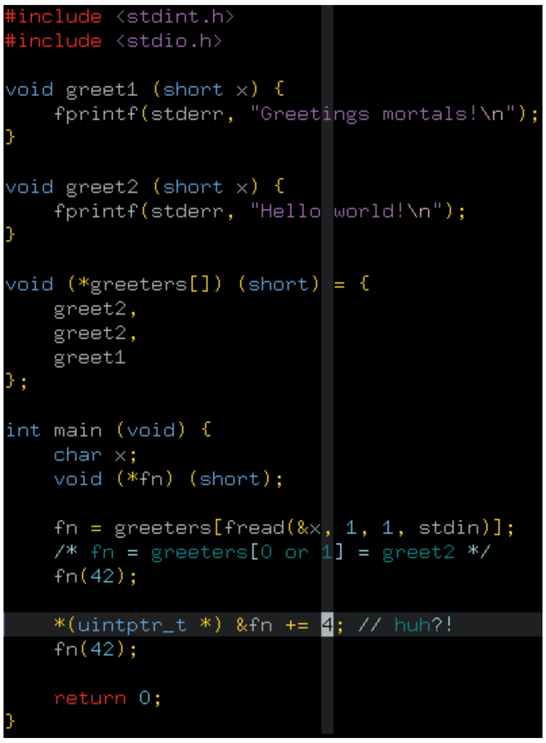
It is therefore desirable to detect such situations. One approach is to mimic C++'s decoration scheme, and detection of mismatched calls takes the form of missing symbols at link time. Another approach is to build a table of function signatures and verify invocations. Since function calls can involve different translation units, the verification needs to happen as a separate step, ideally before linking.

### *Indirect calls*

Ensuring call stack integrity for indirect calls is a trickier problem to overcome as function pointers are commonly used in non-trivial programs. Structures containing function pointers are ordinary in the C world, whereas C++ virtual function tables are essentially the same. Another complicating factor is the language's ability to freely cast between integers, data pointers and function pointers of arbitrary prototypes.

In order to solve this challenge we need to introduce runtime instrumentation for validating that a call target is indeed matching the caller's expectations. This can be achieved by using some form of identifiers to replace function pointers and the call operation becomes a sequence of identifier validation, translation of identifier to real function pointer and ultimately the actual call.

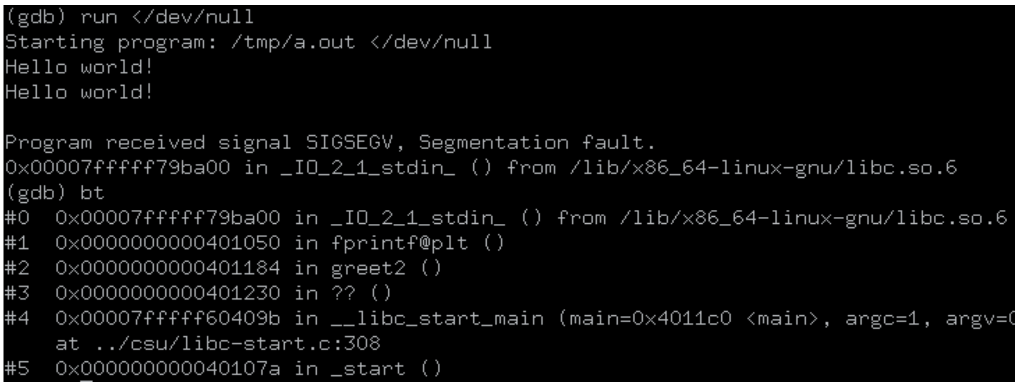
Here is an example where we mutate a function pointer and call it:



```
#include <stdint.h>
#include <stdio.h>

void greet1 (short x) {
    fprintf(stderr, "Greetings mortals!\n");
}

void greet2 (short x) {
    fprintf(stderr, "Hello world!\n");
}

void (*greeters[]) (short) = {
    greet2,
    greet2,
    greet1
};

int main (void) {
    char x;
    void (*fn) (short);

    fn = greeters[fread(&x, 1, 1, stdin)];
    /* fn = greeters[0 or 1] = greet2 */
    fn(42);

    *(uintptr_t *) &fn += 4; // huh?!
    fn(42);

    return 0;
}
```

*Figure 4: Example where we mutate a function pointer and call it.*

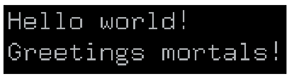
Running a native build seems to allow the second call (starting four bytes into greet2) to run partially then crash in a nested call:



```
(gdb) run </dev/null
Starting program: /tmp/a.out </dev/null
Hello world!
Hello world!

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff79ba00 in _IO_2_1_stdin_ () from /lib/x86_64-linux-gnu/libc.so.6
(gdb) bt
#0  0x00007ffff79ba00 in _IO_2_1_stdin_ () from /lib/x86_64-linux-gnu/libc.so.6
#1  0x0000000000401050 in fprintf@plt ()
#2  0x0000000000401184 in greet2 ()
#3  0x0000000000401230 in ?? ()
#4  0x00007ffff60409b in __libc_start_main (main=0x4011c0 <main>, argc=1, argv=(
    at ../csu/libc-start.c:308
#5  0x000000000040107a in _start ()
```

*Figure 5: Running a native build seems to allow the second call to run partially then crash in a nested call.*

Under CP3, this (carefully contrived) example simply ends up calling a different function of the same prototype due to 'accidental' collision with another valid function identifier. Here is the standard error content:



```
Hello world!
Greetings mortals!
```

*Figure 6: Standard error content.*

Most often, function pointer manipulations end up generating a call to a stub that aborts execution but in the worst case they can end up calling a compatible function. It is important to note that only a function whose address has been explicitly passed as data receives an identifier making it eligible to become the target of an indirect call.

### *Calling outside the sandbox*

It is not hard to imagine a scenario that involves the sandboxed task needing to interact with the host, for the purpose of accepting new input or notifying the host through a callback type of function.

Allowing unguarded external calls would obviously defeat the safety of the sandbox, so special precautions need to be employed.

One possible solution is to treat such external functions in a similar fashion to module imports that have pre-assigned identifiers to allow indirect calls and have the real pointers filled in by the hosting library.

Another option is to allow the static linking of a small helper library that does not go through the same instrumentation and that library is trusted to call anything in a predefined list of interface functions.

### *Timeout protection*

Preventing overly long computational loops is desired to mitigate the risk for denial-of-service vulnerabilities. A robust solution cannot rely only on checks performed when entering functions as illustrated here:

```
for (;;);
```

A simple implementation could inject a check at the end of each code block, whereas a more complex approach can look at the code block graph to reduce the number of locations that need to introduce checks.

Alternatively, if feasible, we can leave the code uninstrumented and rely instead on the hosting library to trigger the forceful termination of the sandbox when desired.

### *Support libraries*

So far we have a way of writing code that has only very limited interaction with functionality not contained in the program itself. This is fine for free-standing code but can be quite limiting if we cannot reuse existing code that assumes a normal hosted environment.

Since we require instrumentation to enforce safety this means we cannot simply link our code with standard C/C++ libraries. However, the benefits of using standard library functionality justify the effort of making the standard libraries available to the sandboxed environment. That can be achieved by compiling some form of a libc and libc++ libraries where lowest level APIs are customized to call outside the sandbox through a controlled interface.

For the standard C library there are several options available with varying degrees of ease-of-integration and licensing requirements: uclibc, dietlibc, musl, newlib, bionic, glibc.

In the case of C++ there is one library in particular that comes to mind after working so much with LLVM and that is LLVM’s own libc++.

## Hosting library

Now that we have a hardened binary all we need is the supporting component to load, unload, execute and enforce the desired runtime guarantees.

To guarantee host process stability we have to deal with several low-level aspects like trapping hardware exceptions, timing out or stack exhaustion.

### *Trapping exceptions*

Hardware exceptions are presented to user-mode code in different ways depending on the host operating system and architecture.

For *Windows*, SEH is the common way of trapping hardware exceptions, however SEH frames can be registered by the sandboxed module as part of try/catch statements. To avoid the risk of adverse effects from sandboxed SEH frames, VEH can be used as it has priority over SEH. To recover from the exception state we can alter the CONTEXT structure to resume execution from a known good state saved at the time of entering the sandbox.

On Unix-like systems hardware exceptions are delivered as signals. Several relevant signals need to be trapped in order to catch memory access violations (SIGSEGV, SIGBUS), aborts (SIGABRT), breakpoints potentially resulting from compiler intrinsics (SIGTRAP), division and floating point errors (SIGFPE) or unsupported instructions (SIGILL). Recovery from the signal can be done using siglongjmp() to a state saved using sigsetjmp() at the time of entering the sandbox.

### *Timeout protection*

If timeout instrumentation is employed then the sandboxing host needs to merely flip a kill-switch variable that is verified by the checks inserted by the instrumentation.

Alternatively, we can come up with creative ways of coercing the thread running the sandbox to trip a fault that we monitor such that the execution returns safely to the host code allowing proper cleanup of all resources held by the sandboxed module.

On *Windows* this involves multithreaded gymnastics with APIs like SuspendThread and SetThreadContext. On POSIX compliant systems pthread_kill can serve the same purpose.

Calling back into the host during sandboxed execution means additional care must be employed to prevent unwanted termination of the sandbox while running non-sandboxed code without leaving the application in a broken state.

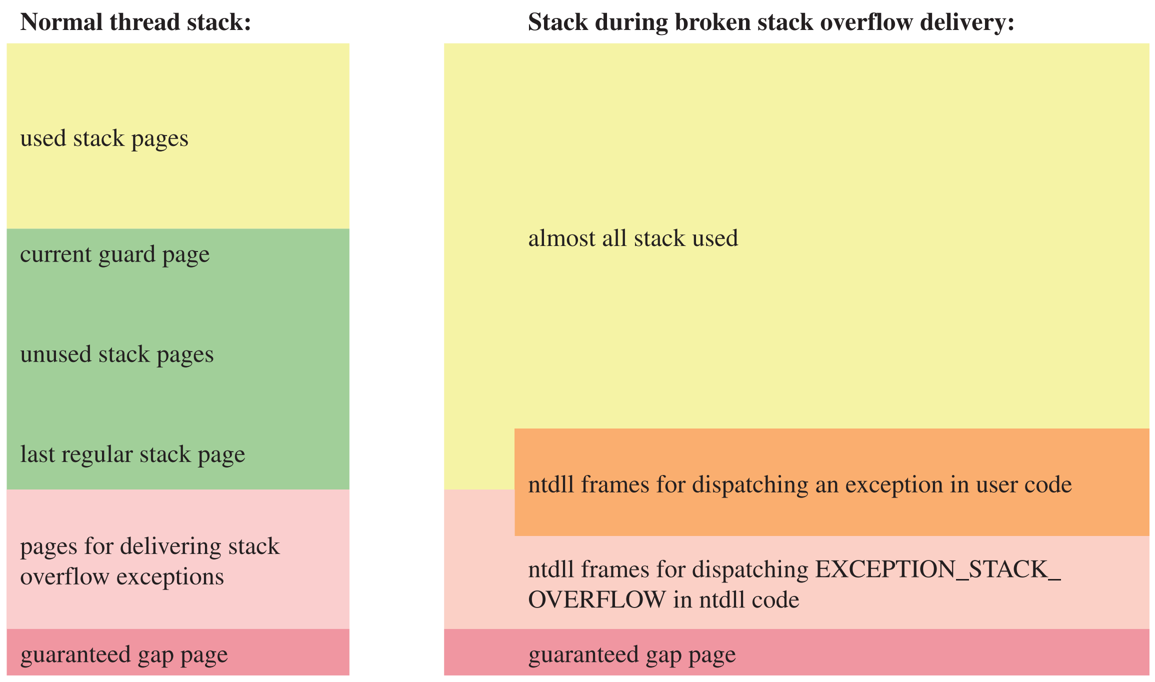
### *Stack exhaustion*

Large stack frames, unbounded recursion are only a few ways to exhaust a thread's stack.

Initially this seemed a simple problem to overcome as stack probes are not a new concept.

On *Windows* there is even a special exception code for stack overflows triggered by probing the bottom pages of the thread's stack: EXCEPTION_STACK_OVERFLOW.

However, this feature turns out to be inadequate for the general handling of stack exhaustion. If the stack gets close enough to the boundary page at the stack bottom and the sandboxed code causes an exception to be delivered, the same stack is used to dispatch the exception. However, at this point the boundary may be reached while in ntdll.dll trying to dispatch the first exception and while holding an internal critical section. This implies that simply resuming execution from a known good state is still subject to deadlocking.


Normal thread stack:
used stack pages
current guard page
unused stack pages
last regular stack page
pages for delivering stack overflow exceptions
guaranteed gap page
Stack during broken stack overflow delivery:
almost all stack used
ntdll frames for dispatching an exception in user code
ntdll frames for dispatching EXCEPTION_STACK_OVERFLOW in ntdll code
guaranteed gap page

The solution we opted for was to instrument function entry blocks to abort if the stack pointer goes under a safe limit.

Unix-like systems do not detect stack exhaustion explicitly, leaving it to be treated like a normal page fault. Also, signal delivery happens normally on the same thread stack, which is at risk of being exhausted. Luckily, the sigaction POSIX mechanism offers the option of delivering these signals on an alternate stack, making handling much easier than on *Windows*. That being said, complications still arise from lack of page probing for large function stack frames. Luckily, the Rust language core team also encountered this problem, prompting the LLVM team to offer an attribute that causes stack probing checks to be inserted during code generation. Even though the '-fstack-check' switch in clang remained a no-op we can use the LLVM instrumentation to tag functions with the much needed stack probing attribute.

### Comparison with similar technologies

According to documentation found on the official project site, *Google*'s PNaCl generated binaries for AMD64 are achieving 25% overhead compared to the same code compiled natively [31]. The overview page for NaCl/PNaCl [32] mentions even better results: 'Running at speeds within 5% to 15% of a native desktop application.'

WebAssembly's performance [33] depends on the browser implementation and research shows that, on average, it runs with an overhead of 45% in *Firefox* and 55% in *Chrome*, with the worst case exceeding twice the native execution time (over 100% overhead).

CP3's observed performance ranges from 10% to 35% overhead with an average of 15%, making it a competitive solution. Aside from the performance aspect an important factor for the adoption of this technology was the fact that the hosting library is relatively small compared to a component that dynamically generates code at run time.

### Applications

Providing standard C/C++ library support enables us to include arbitrarily complex programs and even gives us the option of using other complex libraries as long as we can compile them.

We have tried this technology in the field for several years now, going through a few iterations of feature development and performance optimizations.

Being developed by the team that owns the static scanning engines, we started with moving the more performance-demanding components running orders of magnitude slower under the legacy interpreter, such as the components related to machine learning.

The safety of CP3's in-process sandbox allowed us to deliver the JavaScript emulation engine in a short amount of time while continuously delivering improvements without being constrained to the native engine update cycle that allows three to four iterations per year.

Having the ability to rapidly update sandboxed programs motivated us to transfer even the revamped x86 emulator as a CP3 compiled module. The x86 emulator already had the ability to support content updates for the emulated environment, allowing us to implement new APIs and deliver them through normal definition updates. The functionality introduced that way is emulated, thus runs at a speed significantly slower than native. However, moving the emulator to CP3 allowed us to add new functionality outside the emulated environment, this time running at near-native speed.

We are drastically simplifying the core engines by moving all complexity under the protection offered by the sandbox – be it parsing complex file formats, emulation, command line scanning, etc.

The success we experienced in the case of scanning engines prompted us to expand its usage to other product areas that can benefit from this technology. This allows us to significantly reduce the exploitation risk for in-house or third-party code that we integrate.

## CONCLUSION

Unquestionably, the code safety of any security solution is paramount. It is our responsibility to use best practices when it comes to security from the early stages of development to establishing containment boundaries at runtime.

The emergence of languages focusing on security like Rust should definitely be a top consideration when developing a new component. For existing code, which in the realm of security software is overwhelmingly C/C++, employing modern tooling for testing during development is of great importance.

Given the constant growing complexity of the solutions we offer, moving as much functionality under additional layers of safety can make a huge difference in quality and robustness.

For the security engines team in *Symantec*, CP3 enabled us to rapidly develop, deploy and continuously improve key engine functionality to tackle the challenges of the threat landscape.

## REFERENCES

[1] Why is 0x41414141 associated with security exploits? Stack Exchange. https://security.stackexchange.com/questions/18680/why-is-0x41414141-associated-with-security-exploits.

[2] Executable space protection. Wikipedia. https://en.wikipedia.org/wiki/Executable_space_protection.

[3] Getting around non-executable stack (and fix). Seclits.org. https://seclists.org/bugtraq/1997/Aug/63.

[4] Address space layout randomization. Wikipedia. https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[5] Structured Exception Handling. Microsoft Windows Dev Center. https://docs.microsoft.com/en-us/windows/win32/debug/structured-exception-handling.

[6] Vectored Exception Handling. Microsoft Windows Dev Center. https://docs.microsoft.com/en-us/windows/win32/debug/vectored-exception-handling.

[7] Microsoft-specific exception handling mechanisms. Wikipedia. https://en.wikipedia.org/wiki/Microsoft-specific_exception_handling_mechanisms.

[8] Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP. Microsoft Security Response Center. February 2009. https://msrc-blog.microsoft.com/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/.

[9] PLT and GOT – the key to code sharing and dynamic libraries. Technovelty. May 2011. https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html.

[10] RELRO – A (not so well known) Memory Corruption Mitigation Technique. Trapkit. February 2009. http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html.

[11] Control Flow Guard. Microsoft Windows Dev Center. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[12] Intel MPX Explained. Intel MPX. https://intel-mpx.github.io/.

[13] AddressSanitizer. Clang 11 documentation. https://clang.llvm.org/docs/AddressSanitizer.html.

[14] Intel Software Guard Extensions. Intel. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[15] Control-flow Enforcement Technology Specification. Intel. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[16] Welcome to Native Client. Chrome. https://developer.chrome.com/native-client.

[17] O'Callahan, R. Blink, PNaCl, And Standards. Eyes Above The Waves. May 2013. https://robert.ocallahan.org/2013/05/blink-pnacl-and-standards.html.

[18] asm.js. http://asmjs.org/.

[19] RAII. cppreference.com. https://en.cppreference.com/w/cpp/language/raii.

[20] libFuzzer – a library for coverage-guided fuzz testing. LLVM. https://llvm.org/docs/LibFuzzer.html.

[21] AFL. https://github.com/google/AFL.

[22] Rust. https://www.rust-lang.org/.

[23] Foreign Function Interface. https://doc.rust-lang.org/1.9.0/book/ffi.html.

[24] A little Rust with your C. The embedded Rust book. https://rust-embedded.github.io/book/interoperability/rust-with-c.html.

[25] To panic! or Not to panic! The Rust programming language. https://doc.rust-lang.org/book/ch09-03-to-panic-or-not-to-panic.html.

[26] Function std::panic::catch_unwind. https://doc.rust-lang.org/std/panic/fn.catch_unwind.html.

[27] LLVM Language Reference Manual. LLVM. https://llvm.org/docs/LangRef.html.

[28] Single-Static Assignment Form and PHI. Mapping High Level Constructs to LLVM IR. https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/control-structures/ssa-phi.html.

[29] Definitions and ODR (One Definition Rule). cppreference.com. https://en.cppreference.com/w/cpp/language/definition.

[30] Compile Explorer. https://godbolt.org/.

[31] Native Client FAQ. https://developer.chrome.com/native-client/faq.

[32] Native Client Technical Overview. https://developer.chrome.com/native-client/overview.

[33] Jangda, A.; Powers, B.; Berger, E.D.; Guha, A. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. Usenix 2019. July 2019. https://www.usenix.org/system/files/atc19-jangda.pdf.