The DGA of Pitou - Analyzing a Virtualized Algorithm

Archived: 2026-04-10 03:16:17 UTC

The domain generation algorithm (DGA) of Pitou runs in kernel mode and is protected by a virtual machine, which makes it the hardest DGA I have reverse-engineered so far.

This blog post shows, after an introduction, how the virtual machine of Pitou works. Afterward, reverse engineering of the bytecode is covered showing two approaches. As a result, the DGA was extracted and reimplemented in Python. Documentation of the full virtual instruction set in the Appendix completes the blog post.

Introduction

In my opinion, DGAs are one of the most accessible components of malware. One can usually reverse-engineer the algorithm from an unpacked sample within a few hours. Why is this so?

1. DGAs are usually not explicitly protected.
2. DGAs are relatively easy to localize. Be it via API calls, such as `DnsQuery` and `gethostbyname` or by patterns, e.g., `add al, 61h` .
3. DGAs are usually concise and easy to read. All the more so if decompilers such as Hex Rays are used — they can produce results that are presumably very close to the source code.
4. DGAs are mostly coherent. For example, the letters of the domain name are determined in a loop one after the other, by mapping a random number to the letter a-z.

The DGA of Pitou is different in all four aspects:

1. The entire domain generation algorithm, including seeding, is virtualized. Virtual machines are a particularly effective form of code protection and challenging to analyze, or at the very least incredibly time-consuming.
2. Pitou is a rootkit with dynamically resolved API calls. Furthermore, Pitou uses NDIS hooking to obscure network communication. The usual DGA patterns also do not work for the reason given in point 1.
3. The DGA is pretty long, with a complicated date-based calculation for seeding.
4. The DGA has two major bugs, which make it much more difficult to understand.

These reasons, especially the first, make Pitou by far the hardest domain generation I have ever analyzed.

Previous Work

This blog post focuses exclusively on the analysis of the DGA of Pitou. All other aspects are deliberately not covered.

In August 2014, the F-Secure researchers published an excellent report on Pitou — I highly recommend reading the report to learn more about the characteristics of Pitou. The accompanying blog post briefly answers the most important questions concerning Pitou. The report has a dedicated section on the DGA that mentions a few properties of the algorithm. However, the algorithm itself is not listed, nor are details of the virtual machine that runs the DGA.

In January 2016, Symantec published an entry on their security center. It lists 20 domains that the malware might connect to, without mentioning that these are DGA domains with limited lifetime.

In January 2018, TG Soft's Research Centre (C.R.A.M.) published a blog post on Pitou. The post lists four domains but does not mention, let alone discuss, the domain generation algorithm that produced those domains.

Fourteen days before this blog post was released, Brad Duncan published a diary entry on the SANS Internet Storm Blog titled Rig Exploit Kit send Pitou.B Trojan. He also wrote two entries on his personal blog Malware-Traffic-Analysis.net about two different Pitou samples. This shows that Pitou and its DGA are still relevant — 5 years after F-Secure's report. As the Table at the end of the blog post shows, the DGA with the original seed is still in use, which can probably be attributed to the good protection of the DGA.

Example Domains

The following Wireshark screenshot shows the 20 domains queried on June 20th, 2019:

The domains use some uncommon TLDs such as *.mobi* and *.me*, and stand out as artificially generated, despite consonants and vowels somewhat alternating to give pronounceable names.

Analyzed Sample

I reverse-engineered the following file. It is detected as Pitou by *ESET*, *Ikarus*, and *Microsoft*:

MD5
: 28060e9574f457f267fab2a6e1feec92

SHA1
: 9529d4e33498dd85140e557961c0b2d2be51db95

SHA256
: 43483385f68ad88901031ce226df45b217e8ba555916123ab92f63a97aef1d0e

filesize
: 522K

compile timestamp
: 2017-10-31 10:15:25 UTC

link
: VirusTotal

The file unpacks to this binary, which is now also detected as Pitou by *Avast*, *AVG*, and *Fortinet*.

MD5
: 70d32fd5f467b5206126fca4798a2e98

SHA1
: 6561129fd718db686673f70c5fb931f98625a5f0

SHA256
: f43a59a861c114231ad30d5f9001ebb1b42b0289777c9163f537ac8a7a016038

filesize
: 405K

compile timestamp
: 2017-08-22 10:24:10 UTC

link
: VirusTotal

The above executable then drops the rootkit. Pitou contains a 32-bit and a 64-bit module to support both architectures.

32-bit

MD5
: 9a7632f3abb80ccc5be22e78532b1b10

SHA1
: 2d964bb90f2238f2640cb0e127ce6374eaa2449d

SHA256
: ab3b7ffaa05a6d90a228199294aa6a37a29bb42c4257f499b52f9e4c20995278

filesize
: 431K

compile timestamp

2017-03-22 01:21:01 UTC

link

VirusTotal

64-bit

MD5

264a210bf6bdded5b4e35f93eca980c4

SHA1

8f6ff0dd9b38c633e6f13bde24ff01ab443191f6

SHA256

ddb82094dec1fc7feaa4d987aee9cc0ec0e5d3eb26ba9264bb6ad4aa750ae167

filesize

478K

compile timestamp

2017-02-27 06:13:41 UTC

link

VirusTotal

I only looked at the 64-bit variant.

The Virtual Machine

This section covers the virtual machine that protects the DGA, as well as other functions of Pitou components. The first part shows the main components of the virtual machine. The second part then discusses the properties of the VM and its bytecode.

Components

This video of Tim Blazytko and Moritz Contag gives an excellent introduction to the main components of a virtual machine, which are:

- VM Entry / VM Exit
- VM Dispatcher
- Handler Table

VM Entry

VM entry and exit are responsible for context switching. The VM entry copies the native context (registers and flags) to the virtual context. The VM is entered in the following screenshot:

Four arguments are passed to the virtualized DGA according to the x64 calling convention (in `rcx`, `rdx`, `r8`, and `r9`). I will revisit those arguments and their meanings when discussing the arguments to the DGA. The call to the DGA pushes a return value on the stack, which is later used in VM exit in a `ret` call to jump back to the native code right after the call to the dga. The call leads to this native code island:

The call, which is 5 bytes long, is in the middle of the bytecode that the VM executes. The purpose of the call is to push the address of the following address (marked with `entry_point_bytecode`) on the stack. This address is the

entry point (which is not at the beginning of the bytecode) of the virtual DGA. The call then jumps to the start of the virtual machine. The virtual machine is called from 46 different locations, meaning there are 46 different bytecode starting points, all probably implementing different components of Pitou, for example:

Since native code, i.e., a `call` instruction, can be in the middle of the bytecode, the VM must be able to recognize that instruction and skip it. We will see later how this is implemented.

The VM itself starts as follows:

The native context is copied to the virtual context, which is accessed by the register `rsi`. The following content is copied:

- The **flags**, by using `pushfq`
- The **general purpose registers** `rax`, `rbx`, `rcx`, `rdx`, `rdi`, `rsi` and `r8` through `r15`. The registers `rip`, `rsp` and `rbp` are not copied, as the VM itself uses those.
- The **XMM-registers**, although none of the virtual instructions modifies them.

The last line of the screenshot, `pop rax`, removes the entry point to the virtual code from the stack. This entry point is then also saved in the virtual context:

The above screenshot also shows how VM entry checks if the entry point is inside the bytecode. I named the lowest address of the bytecode `imagebase`, and the highest address `highest_addr`. If the entry point is within this range, the virtual instruction pointer is set to the entry point. This completes the VM entry.

VM Dispatcher

The task of the dispatcher is to fetch and decode the instructions. Usually — and the VM of Pitou is no exception — a handler belonging to the opcode is called. This handler is also responsible for updating the context of the VM, especially the instruction pointer.

The following screenshot shows the VM dispatcher. First, the virtual instruction pointer is read (**1**). If the control register `CR8` is set, the instruction pointer converts to an offset from the entry point, and a software interrupt is triggered (**2**). The dispatcher then reads a byte at the instruction pointer. If it corresponds to `E8` (**4**), the VM is at a native call, as used for passing the entry point to the VM (see section VM Entry). The five bytes of the call are simply skipped (**5**). All other byte values are valid bytecode. The 6 least significant bits correspond to the opcode, which refers to a function from the handler table. The VM then jumps to this function (**6**).

The 6 bit can represent up to 64 different functions, of which `0x28` is excluded because that corresponds to the least significant 6 bits of the reserved byte `0xE8`. However, only the first 29 entries in the handler table point to different routines; I labeled them `instruction_00` to `instruction_1C` according to the primary opcode that invokes them. As of opcode `0x1D`, previous functions are reused, e.g., `0x1D` uses the handler of opcode `0x03`.

Some handler functions are accessed by exactly one opcode, and others have multiple opcodes. For example, `0x07`, `0x47`, `0x87`, `0xc7` all map to the same handler.

VM Exit

The virtual jmp/call/ret instruction (handler `0x02`) also handles VM exit. The appendix discusses this handler in detail. The next screenshot of parts of the handler shows how the native registers are restored from the virtual context. There isn't much more to exiting the VM; the handler at the end just returns to the code following the VM entry, by using the address on the stack.

Properties

The virtual machine of Pitou is stack-based and executes 64-bit code. At least the 64-bit module does. The 32-bit module probably has 32-bit virtual instruction. This section describes the properties of the virtual machine. A complete list of all instructions can be found in the Appendix.

Registers

The VM uses virtual copies of the x64 general purpose registers and flags. As expected, it also has a virtual instruction pointer, a virtual stack pointer, and a virtual base pointer. In addition, two state registers are available, which are used for jumps:

Instruction-Set

The virtual instructions have a variable length from 1 byte to 11 bytes. As section VM Dispatcher has shown, the lowest 6 bits of the first byte are the opcode that determines the handler function. The highest 2 bits can be used to select variants within the handlers. The first byte is the only mandatory one. In fact, many virtual instructions are only 1 byte long. Often determined by one of the two bits of the prefix, one or two optional specifier-bytes follow. The format varies from instruction to instruction (for example, it could contain information about which memory segment the instruction uses, whether the following operand is signed, or which size it has). After the optional specifier, an optional operand might follow.

Operands are stored in little-endian order and can be bytes, words, double-words, or quad-words. Operands and the specifier are XOR-encrypted with the following keys:

- Bytes are encrypted with `0x57`
- Words are encrypted with `0x13F1`
- Double-words are encrypted with `0x69B00B7A`
- Quad-words are encrypted with `0x7EF5142A570C5298`

For example, the operand `AB 01` is encrypted to the value `0x125A` (`0x01AB XOR 0x13F1`). Jump targets are relative to the start of the virtual code. For example, if the virtual code starts at `0xFFFFF87EC582C000` , then a jump with decrypted operand `0x123` would set the virtual instruction pointer to `0xFFFFF87EC582C123` . Instruction 0x01, Instruction 0x04, Instruction 0x06 and Instruction 0x18 can make use of addresses relative to the virtual code. One handler, Instruction 0x18, can also use addresses relative to *the start of the executable*. This allows the handler to access memory outside of the VM. The DGA uses this addressing to read static strings such as the list of top level domains. This is the only instruction that is not position-independent, i.e., which would need to be relocated, if the virtual code were to be placed elsewhere.

Let's look at an example: the virtual instruction `JZ 0xfffff8800586c445` — jump to address `0xfffff8800586c445` if the zero bit is set — is encoded as follows:

- Conditional jumps are in handler `0x06` . This handler uses both prefix bits.
- The first prefix bit determines whether the condition is unary (`0`), or a binary comparison (`1`). In our case, we have a unary condition. The second prefix bit only applies to binary comparisons. All in all the first byte is `0x06`

- The first (most significant) two bits of the specifier determine how the flags are combined. `1 0` means the condition is satisfied if all selected flags are set. The remaining 6 bits are a bitmask for various flags. The `JZ` instruction only has to look at the `ZF` flag, so only this bit is set in the bitmask. The whole specifier is `0xDF`, which is then encrypted with key `0x57` to get the final value `0x88`.
- The jump target is calculated as follows: Subtract the image base (`0xfffff87EC582C000`) from the target (`0xfffff8800586c445`), then XOR encrypt the result (key `0x7EF5142A570C5298`).

Virtual Stack

Since the virtual machine is stack-based, the stack is of particular importance. Although the VM emulates 64-bit code, i.e., it handles values of 64-bit length at most, the stack is 128-bit wide. Each stack entry consists of two 64-bit slots. Most instructions only use the slot stored at the lower address, shown on the left in all stack plots. I refer to this slot as *value* or *regular storage*. The other slot, 64-bits higher than the first one, is often used to store the address of the value in the left slot (*value* slot). I refer to this slot as the *extra* slot.

As usual, the stack follows the “last in first out” principle. It grows towards smaller addresses and shrinks towards greater addresses. The virtual stack register variable points to the last item in the stack (full stack implementation).

Reverse Engineering

This section covers reverse engineering of the virtual machine. At first, I had written a disassembler but was not satisfied with its output; respectively it was still too complicated to read. Especially the lack of tools for the user-defined instructions made analysis difficult.

The second approach was to bring the virtual code back to C code. This worked surprisingly well and resulted in an algorithm that was easy to understand despite complexity and bugs. I use a part of the DGA as an example to illustrate the four steps. The snippet corresponds to a simple mathematical statement, which can be written as a single line in C This should give a sense of how long the results of the individual steps are and how difficult they are to read.

Approach 1: Disassembler

The usual steps to analyze a VM are as follows (see Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation:

1. Explore how the instructions are decoded from the bytecode, especially which bits are the opcode that determines the handler function. As shown in section VM dispatcher, the lowest 6 bits of the first byte

specify the handler that decodes the rest of the instruction and also sets the instruction pointer forward.

2. Understand the architecture of the VM. This has already been described for the post part in section The Virtual Machine.

3. Finally, the handlers have to be analyzed. This is the most time-consuming part of the analysis. In the case of Pitou, 29 different functions have to be reverse-engineered.

The following screenshot shows the first handler for opcode `0x00`. The function is not all that long and fortunately is not obfuscated.

The following illustration shows the instruction encoding and the effect on the stack for opcode `0x00` (and the aliases `0x2F` and `0x35`):

The fields of the encoded instruction have the following meaning:

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x00` , `0x2F` or `0x35`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

The handler uses the first bit of the prefix to determine whether an optional size specification follows. If the bit is not set, the operation is assumed to be 32-bit, and the instruction is only one byte long. If, on the other hand, the prefix bit is set, then another byte must follow, of which the lowest two bits determine the size of the operation. `0` always stands for Byte (8bit), `1` for Word (16bit), `2` for Dword (32-bit, the default), and `3` for Qword (64-bit). The instruction pops two values from the stack, *XOR*s them and pushes the result back on the stack, casting the values to the specified size. It is straightforward to reimplement this handler by the disassembler:

1. check if the `has size` bit is set, if yes, read the next byte and determine the data width based on the last two bits (`0` means *Word*, etc.). Otherwise, default to size *Dword*.
2. output of `XOR <size>` , e.g., `XOR dword` .
3. if the `has size` bit is set, increase the instruction pointer by 2, if not by 1.

After reimplementing all 29 handlers, the bytecode can be disassembled. The following excerpt from the disassembled bytecode shows the virtual instructions and their effect on the stack for the x64 instruction: `XOR dword edx, edx` , i.e., zeroing register `edx` .

```
 FFFFF8800585817C NOP
                                                (empty stack)
FFFFF8800585817D PUSH dword (rdx, addr(rdx))
                                                | rdx          | addr(rdx)        |
FFFFF8800585817F PUSH dword (addr(rdx), addr(rdx))
                                                | addr(rdx)    | addr(rdx)        |
                                                | rdx          | addr(rdx)        |
FFFFF88005858181 DREFH dword
                                                | rdx          | addr(rdx)        |
                                                | rdx          | addr(rdx)        |
FFFFF88005858182 XOR dword
                                                | edx XOR edx  | addr(rdx)        |
```

```
FFFFF88005858183 P2E dword
                                                    (empty stack)
```

1. The first instruction, `NOP` does nothing.
2. The register `rdx` is pushed on the stack with handler 0x12, but using two different settings for the `push address` flag: the first `PUSH` pushes the value of the register `rdx` to the value slot of the stack, the second `PUSH` pushes the address of register `rdx` (in both cases, the address is stored in the *extra* slot of the stack entries).
3. The address of register `rdx` on top of the stack is dereferenced by `DREFH dword` to obtain `rdx`. Both entries on the stack are now equal. Why the compiler didn't push the register in the first place instead of the address is not clear to me.
4. The `XOR dword` instruction takes the values from the two topmost stack entries, xors them and pushes the result back on the stack. The *extra* slot remains the same, i.e., at the value that was 1 entry from the top before the XOR operation.
5. Finally, the `P2E dword` instruction pops entry from the stack, and moves the value slot (`edx XOR edx` to the memory given by the extra slot — `addr(rdx)`. This means, the virtual register `rdx` is set to `0`.

Here is the bytecode for our example snippet, which, as mentioned in the introduction, corresponds to a simple mathematical statement in the DGA:

The disassembler produces the following lines from the above bytecode:

The first number is the address of each line. The number in the blue square is the handler. Next follow the bytes that make up the instruction. Most are 1 or 2 bytes long, but there are also much longer instructions such as the conditional jump with 10 bytes length. The mnemonic of the instruction follows after the bytecode, with a bit of syntax highlighting. Since I quickly abandoned the disassembler approach, not all mnemonics are particularly well chosen and meaningful. They are all listed in the Appendix though.

One could probably extract the DGA from the output of the disassembler with some patience. However, I did not try for long, because:

1. The output of the disassembler is very long, altogether 3681 lines.
2. The virtual machine uses special instructions, like the `P2E` or `DREFH`, as shown above. These are new and would need practice to grasp quickly.
3. There are no tools that display the output nicely, e.g., a graph view would help to see the code flow.

There are multiple ways to make the disassembly more readable, for instance:

1. Add patterns to reduce the number of lines. For example, the 6 lines that make up the `XOR edx, edx` code above are quite common. Clever pattern matching could reduce the number of lines drastically.
2. Implement the disassembler as an IDA Pro processor module. This would give a nice graph view, with all the commenting tools available in IDA to make the code more readable.

I didn't pursue these ideas because I had a more tempting goal in mind: to decompile the virtual code to C. You can run the disassembler yourself with the Python script on Github:

```
python3 main.py disassembly -o pitou.dis
```

Approach 2: Decompiler

This section shows how to convert the virtual code back to C. The plan is to first translate the bytecode to x64 assembly, then assemble that to an x64 binary. IDA Pro can then open the binary, and produce x64 disassembly as well as C code with the Hex Rays plugin. The purpose of those steps is to be able to use existing tools, in particular, the Hex Rays decompiler. This is especially worthwhile because the DGA uses many instances of integer division by multiplication, which are hard to read in assembly but are very well handled by decompilers.

Step 1: Dynamic Binary Translator

The only one of the four steps for which there exist no tools is the translation of the bytecode of the VM to x64 assembly. The task of binary translation is to recompile the sequence of virtual instructions to x64 assembly.

I decided to do dynamic translation by emulating the bytecode and simultaneously outputting the corresponding x64 instructions on the fly. To emulate the virtual instructions, I used the disassembler from step 1. This provides the decoding of the instructions as well as the recursive traversal of the code. Since the VM is stack-based, the stack has to be emulated. Two elements per stack slot are tracked:

1. An immediate or a register name. I labeled this the *value* and *extra* for the two slots of the stack.
2. Only if (1) is a register, also the sequence of assembly instructions that lead to the register value on the stack. This list of instructions can be empty. I labelled the list of instructions *value instructions* and *extra instructions*.

The following sections show how to emulate most categories of virtual instructions. I hope this clarifies the method. Either way, the source code of my binary translator can be referenced on Github for more details.

Unary Operations

The virtual instructions that model unary operations are NEG; INC and DEC; NOT; and SHR, SHL, ROR, ROL, SAR, and SAL. They are all decompiled in the following manner:

1. Pop a value from the stack. In the following example `rcx`.
2. If the value is a register, cast it according to the size of the instruction, e.g., *Dword* would cast `rcx` to `ecx`. The cast register becomes the *value* of the new stack entry.
3. Concatenate all *value instructions* from the popped stack element, if there are any. Then add `<mnemonic> <size> <value (cast)>`, e.g., `NEG DWORD ecx`. Set the sequence of instructions as the *value instructions* of the new stack entry.
4. Copy the *extra* and *extra instructions* from the popped stack entry to the *extra* and *extra instructions* slots of the new entry.
5. Push the new stack entry on the stack.

The decompiler only affects the stack; it does not write any output. The following table shows an example stack before the NEG handler is called (with size flag unset):

value	value instructions	extra	extra instructions
rcx	ADD rcx, 10 SHR rcx, 2	addr(rcx)	

And after the handler is called:

value	value instructions	extra	extra instructions
ecx	ADD rcx, 10 SHR rcx, 2 NEG DWORD ecx	addr(rcx)	

Binary Operations

There are 7 virtual instructions that perform a binary operation: XOR, SBB, SUB, OR, AND, CMP, and ADD, Binary arithmetic instructions are all handled similar to unary operations:

1. Pop two values from the stack. In the following example `[r10 + 1]` and `rax`, with the first popped value being the source of the instruction, and the second being the destination.
2. If the destination value is a register, cast it according to the size of the instruction, e.g., *Word* would cast `rax` to `al`. The cast target register becomes the *value* of the new stack entry.
3. Concatenate all value instructions from the two popped stack elements, if there are any. Then add `<mnemonic> <size> <target value (cast)>, <src value>`, e.g., `XOR byte al, [r10 + 1]`. Set the sequence of instructions as the *value instructions*.
4. Copy the values *extra* and *extra instructions* from the second popped stack entry to the *extra* and *extra instructions* slots of the new entry.
5. Push the new stack entry on the stack.

As for the unary operations, the decompiler only changes the stack and does not write any output. The following table shows the stack before the XOR handler is called, with size flag set and the size byte set to `0` (representing *Byte*):

value	value instructions	extra	extra instructions
[r10 + 1]	SHR r10d, 2	r10	ADD QWORD r10, 1
rax	SHL rax, 2	addr(rax)	

The XOR-operation changes the stack to:

value	value instructions	extra	extra instructions
al	SHR r10d, 2 SHL rax, 2 XOR BYTE al, [r10 + 1]	addr(rax)	

Move Operations

Virtual instructions that write data are M2E, M2V, P2E, and P2V. For example, the M2E instruction moves the data in the *value* slot of the topmost stack to the memory location specified by the *extra* slot. Often, this *extra* slot

contains the address of a virtual register, which results in the *value* being moved to a register. For example, if applied to the following stack, the content of `rsp` would be written to `rax`:

value	value instructions	extra	extra instructions
rsp	SUB QWORD rsp, 8	addr(rax)	SHL QWORD rax, 3 ADD QWORD rax, 1

This move corresponds to the assembly `MOV rax, rsp`. However, `rsp` and `rax` have been changed on the stack, as can be seen in the *value instructions* and *extra instructions* columns: these transformations have been applied on the stack. These operations need to be *realized* by the decompiler by simply writing them out before the `MOV` statement:

```
 SHL QWORD rax, 3
ADD QWORD rax, 1
SUB QWORD rsp, 8
MOV rax, rsp
```

In almost all bytecode sequences of the VM, the destination and source are the same. In these cases, the `MOV` is removed, e.g., `MOV rsp, rsp` would not be written. Now that the instructions have been realized, they are removed from the respective stack slots if the handler did not already pop the stack entirely (like `P2E`, `P2V` do). The *value* and *extra* columns stay the same.

value	value instructions	extra	extra instructions
rsp		addr(rax)	

Popping from the stack work the same way. A `POP dword rax` on this stack

value	value instructions	extra	extra instructions
rdx	SHR rdx, 2	addr(rdx)	

results in an empty stack, and the binary translation writes:

```
 SHR rdx, 2
MOV DWORD eax, edx
```

There is a small problem with this approach. Consider these three lines of instructions that the above move might produce (the first two lines stem from realizing the *value instructions* field, the last line is the actual move).

```
 SHL QWORD rax, 2
ADD QWORD rax, r11
```

```
MOV QWORD rcx, rax
```

The issue here is that the first line modifies `rax`, which is then used again by the second instruction. The original virtual register, however, would not have been changed, as the operations all happen on the stack. In this example, there is a simple fix: The target of the operation, `rcx`, is not used in the previous assembly lines. It can, therefore, serve as a substitute for register `rax`. This leads to an additional line of assembly, `MOV rcx, rax`, to copy the value over. In return, the `MOV rcx, rax` can be omitted, since the calculations already refer to `rcx`:

```
 MOV rcx, rax
SHL QWORD rcx, 2
ADD QWORD rcx, r11
```

Unfortunately, this does not always work, as the following example shows:

```
 SHL QWORD rax, 1
ADD QWORD rax, rax
```

In this case, the calculations operate on the target `rax` of the move instruction, which is therefore eliminated. The final target `rax` is the same as the tainted register. In those cases, the binary translation uses `r15` as the temporary register. This register could, of course, be in use, so it is first stored on the stack, and then restored at the end. I chose `[rsp-1000]` arbitrarily. Since the virtual machine does not use the native stack except for `RET`, this should not cause any problems. After `r15` is saved, it receives the value of the tainted register `rax` with `MOV r15, rax`. Then follow the two original lines with `rax` replaced by `15`. Finally, `r15` is moved back to `rax` and `r15` is restored from the stack:

```
 MOV [rsp-1000], r15
MOV r15, rax
SHL QWORD r15, 1
ADD QWORD r15, rax
MOV rax, r15
MOV r15, [rsp-1000]
```

Jumps and Calls

The JMP, CALL and RET handler is in essence a unary operation with some additional steps. First, the *value* is popped from the stack, so

value	value instructions	extra	extra instructions
0xFFFFF8800588FCBA		0xFFFFF8800588FCBA	

results in an empty stack. If the value is a bytecode address, then decompiler prepends `_addr_` to the hex string to produce a label for the jump target, i.e.:

```
JMP _addr_FFFFF8800588FCBA
```

At the target address, the label is also written:

```
    ADD QWORD rsp, 8
   RET
_addr_FFFFF8800588FCBA:
   MOV DWORD eax, 1
```

The target can also be a symbolic expression. Of these, `[rsp]` is interesting, because `JMP [rsp]` is essentially `RET`. The decompiler snippet above shows an example of a `RET` being used in place of a `JMP [rsp]`.

Conditional Jumps

The conditional jump IF x → y is straightforward to implement: Determine the mnemonic according to the table in the previous section, and convert the destination to an absolute address, if given as a relative offset. The handling of the jump target is the same as for handler 0x02. An example output would be:

```
JNZ _addr_FFFFF8800588FCBA
```

The virtual data type conversions in handler 0x0B just need to be converted to the corresponding mnemonic and output.

Discarding Stack Elements

The handler POPD removes *n* elements from the stack, and outputs all instructions of the value and extra slot, e.g.:

value	value instructions	extra	extra instructions
r10d	AND DWORD r10d, r10d	addr(r10)	
r8d	XOR DWORD r8d, r8d	addr(r8)	
ebx	AND DWORD ebx, ebx	addr(rbx)	

becomes

value	value instructions	extra	extra instructions
ebx	AND DWORD ebx, ebx	addr(rbx)	

with these lines written by the binary translation:

```
 AND DWORD r10d, r10d
XOR DWORD r8d, r8d
```

Swapping Stack Elements

The handler STACKSWP swaps the two topmost value slots of the stack — including value instructions. For example,

value	value instructions	extra	extra instructions
r10d	AND DWORD r10d, r10d	addr(r10)	
r8d	XOR DWORD r8d, r8d	addr(r8)	

becomes

value	value instructions	extra	extra instructions
r8d	XOR DWORD r8d, r8d	addr(r10)	
r10d	AND DWORD r10d, r10d	addr(r8)	

No x64 assembly is output.

Dereferencing

The dereferencing happens exclusively on the stack, and no output is generated. For example:

value	value instructions	extra	extra instructions
rax	ADD QWORD rax, rsp ADD QWORD rax, 56	addr(rax)	

turns into

value	value instructions	extra	extra instructions
[rsp + rax + 56]		rax	ADD QWORD rax, rsp ADD QWORD rax, 56

The translation also replaces the `ADD` instructions with `+` if dereferencing happens. Hence, in many cases, the instructions can be cleared out and moved to the value part.

Virtual Instructions that don't change the stack

Two handlers do nothing: NOP, TRIPLE, and two handlers that change state variables that don't seem to matter: SET1, and STATE. The binary translation does nothing for these four handlers.

Multiplying and Division

Multiplication (MUL) and division (DIV) are special: First, the two virtual registers `rax` and `rdx` are copied to the native registers. Then a value is popped from the stack, e.g.,

value	value instructions	extra	extra instructions
rcx		addr(rcx)	

turns into an empty stack. Then `MUL <size> <popped value>` or `DIV <size> <popped value>` is executed, for instance, `MUL BYTE cl`. The result is not pushed back on the stack; instead, the virtual context stores the two native registers `rax` and `rdx`.

Signed multiplication (IMUL) works differently. The instructions pops two values from the stack. The first is set to `rdx`, the second to `rax`. Then `IMUL Byte dl`, `IMUL Word dx`, `IMUL Dword edx`, or `IMUL Qword rdx` is calculated, depending on the size. Finally, `rdx` then `rax` are pushed back on the stack. For example, the stack before:

value	value instructions	extra	extra instructions
r9		addr(r9)	
rax		addr(rax)	

And the stack after:

value	value instructions	extra	extra instructions
rdx		addr(rax)	
rax		addr(r9)	

The process generates these two lines of assembly:

```
 MOV rdx, r9
IMUL DWORD rdx
```

Pushing

The two PUSH instructions in handler 0x18 and handler 0x12 push an immediate, a register or the address of a register on the stack. The binary translation only changes the stack without writing any output. For instance, pushing register `rbp` would change this stack

value	value instructions	extra	extra instructions
rax		addr(rax)	

to this stack:

value	value instructions	extra	extra instructions
rbp		addr(rbp)	
rax		addr(rax)	

xDIAGy

The strange instructions `MDIAG`, `MDIAGA`, `PDIAG`, `PDIAGA` just affect the stack, and the binary translation only needs to move stack entries around. For example, `PDIAG` on this stack:

value	value instructions	extra	extra instructions
r9	ADD QWORD r9, -1	addr(r9)	
rax		addr(rax)	

leads to:

value	value instructions	extra	extra instructions
rax		r9	ADD QWORD r9, -1

Example: Binary Translation

The following snippet shows the disassembly together with the resulting x64 assembly (marked by `▶`).

```
 FFFFF880058766F6 NOP
FFFFF880058766F7 PUSH dword (addr(r8), addr(r8))
FFFFF880058766F9 DREFH dword
FFFFF880058766FA PUSH dword (r8, addr(r8))
FFFFF880058766FC AND dword
FFFFF880058766FD POPD
               ▶ AND DWORD r8d, r8d
FFFFF880058766FE NOP
FFFFF880058766FF STATE  1
FFFFF88005876700 IF NOT ZF -> JMP 0xFFFFF8800587C8F8
               ▶ JNZ _addr_FFFFF8800587C8F8
FFFFF8800587670A NOP
FFFFF8800587670B PUSH dword (51EB851Fh, 51EB851Fh)
```

```
FFFFF88005876710 POP dword rax
                 ▶ MOV DWORD eax, 1374389535
FFFFF88005876712 NOP
FFFFF88005876713 PUSH dword (addr(r9), addr(r9))
FFFFF88005876715 DREFH dword
FFFFF88005876716 PUSH dword (rax, addr(rax))
FFFFF88005876718 IMUL dword
                 ▶ MOV rdx, r9
                 ▶ IMUL DWORD r9d
FFFFF88005876719 POP dword rdx
FFFFF8800587671B POP dword rax
FFFFF8800587671D NOP
FFFFF8800587671E PUSH dword (addr(rdx), addr(rdx))
FFFFF88005876720 DREFH dword
FFFFF88005876721 PUSH byte (5h, 5h)
FFFFF88005876724 SAR dword
FFFFF88005876726 P2E dword rdx
                 ▶ SAR DWORD edx, 5
FFFFF88005876727 NOP
FFFFF88005876728 PUSH dword (addr(rdx), addr(rdx))
FFFFF8800587672A DREFH dword
FFFFF8800587672B POP dword rax
                 ▶ MOV DWORD eax, edx
FFFFF8800587672D NOP
FFFFF8800587672E PUSH dword (addr(rax), addr(rax))
FFFFF88005876730 DREFH dword
FFFFF88005876731 PUSH byte (1Fh, 1Fh)
FFFFF88005876734 SHR dword
FFFFF88005876735 P2E dword rax
                 ▶ SHR DWORD eax, 31
FFFFF88005876736 NOP
FFFFF88005876737 PUSH dword (rdx, addr(rdx))
FFFFF88005876739 PUSH dword (addr(rax), addr(rax))
FFFFF8800587673B DREFH dword
FFFFF8800587673C ADD dword
FFFFF8800587673D P2E dword rdx
                 ▶ ADD DWORD edx, eax
FFFFF8800587673E NOP
FFFFF8800587673F PUSH dword (addr(rdx), addr(rdx))
FFFFF88005876741 DREFH dword
FFFFF88005876742 PUSH byte (64h, 64h)
FFFFF88005876745 IMUL dword
                 ▶ MOV rax, 100
                 ▶ IMUL DWORD edx
FFFFF88005876746 POPD
FFFFF88005876747 PUSH dword (addr(rdx), addr(rdx))
FFFFF88005876749 PDIAG
```

```
FFFFF8800587674A P2E dword rdx
                ▶ MOV DWORD edx, eax
FFFFF8800587674B NOP
FFFFF8800587674C PUSH dword (r9, addr(r9))
FFFFF8800587674E PUSH dword (addr(rdx), addr(rdx))
FFFFF88005876750 DREFH dword
FFFFF88005876751 CMP dword
FFFFF88005876752 POPD
                ▶ CMP DWORD r9d, edx
FFFFF88005876753 NOP
FFFFF88005876754 STATE  1
FFFFF88005876755 IF NOT ZF -> JMP 0xFFFFF88005867CB8
                ▶ JNZ _addr_FFFFF88005867CB8
FFFFF8800587675F NOP
FFFFF88005876760 PUSH dword (51EB851Fh, 51EB851Fh)
FFFFF88005876765 POP dword rax
                ▶ MOV DWORD eax, 1374389535
FFFFF88005876767 NOP
FFFFF88005876768 PUSH dword (addr(r9), addr(r9))
FFFFF8800587676A DREFH dword
FFFFF8800587676B PUSH dword (rax, addr(rax))
FFFFF8800587676D IMUL dword
                ▶ MOV rdx, r9
                ▶ IMUL DWORD r9d
```

As you can see from the listing above, the binary translation dramatically reduces the number of instructions. The original 3681 lines of disassembly are condensed into 786 x64 instructions, a reduction of about 80%. The progress can also be seen in the code fragment used as an example throughout this post. This disassembly turns into these lines of x64 assembly:

```
_addr_FFFFF880058766F6:
   AND DWORD r8d, r8d
   JNZ _addr_FFFFF8800587C8F8
   MOV DWORD eax, 1374389535
   MOV rdx, r9
   IMUL DWORD r9d
   SAR DWORD edx, 5
   MOV DWORD eax, edx
   SHR DWORD eax, 31
   ADD DWORD edx, eax
   MOV rax, 100
   IMUL DWORD edx
   MOV DWORD edx, eax
   CMP DWORD r9d, edx
   JNZ _addr_FFFFF88005867CB8
   MOV DWORD eax, 1374389535
```

```
    MOV rdx, r9
    IMUL DWORD r9d
    SAR DWORD edx, 7
    MOV DWORD eax, edx
    SHR DWORD eax, 31
    ADD DWORD edx, eax
    MOV rax, 400
    IMUL DWORD edx
    MOV DWORD edx, eax
    CMP DWORD r9d, edx
    JNZ _addr_FFFFF8800587C8F8
    JMP _addr_FFFFF88005867CB8
_addr_FFFFF88005867CB8:
    MOV DWORD eax, 1
    JMP _addr_FFFFF88005852C11
_addr_FFFFF8800587C8F8:
    XOR DWORD eax, eax
    JMP _addr_FFFFF88005852C11
```

This snippet is much more readable already. Mainly still missing is a representation of the code flow in a graph, as well as better handling of the optimized integer division, which make up most of the lines.

Step 2: Assembler

In the first step, x64 assembly was created. To be able to analyze it with IDA Pro, it must first be converted into an executable. For this, I used the Netwide Assembler (NASM). The code from the previous section only needs to be amended with two section headers. The `data` section, where I copied in data that the VM reads from the native context, and the `text` section before the actual code. Although the DGA is a function, I used it directly as an entry point for the binary.

```
 section .data
data_FFFFF8800589E540    dd    31,28,31,30,31,30,31,31,30,31,30,31
data_FFFFF8800589E570    dd    31,29,31,30,31,30,31,31,30,31,30,31
...
section .text
global _start
```

The code on Github already adds those lines if you run:

```
 python3 main.py nasm -o pitou.asm
```

The resulting file can then be compiled with:

```
 nasm -f elf64 pitou.asm
ld pitou.o -o pitou.bin
```

Of course, this makes our code unreadable at first, because it is now in a binary format again. Our example snippet is now:

Step 3: Disassembler

The executable file from the previous step can now be opened and disassembled in IDA. Our excerpt now looks as follows. In contrast to step 1, we have a graph view and the possibility to add comments.

Step 4: Decompiler

Finally, Hex Rays can decompile the disassembly to C code. Our example snippet looks as follow:

The very long disassembly from Approach 1 became a single line of C code, which corresponds to this statement

> Is a year a leap year?

You can run the decompiler yourself with the Python script on Github:

```
python3 main.py nasm -o pitou.nasm
```

The DGA

This section covers reverse engineering of the DGA using the output from the process in the previous section. A Python reimplementation is presented as the result of the reverse engineering. The script can generate the domains for any given date.

DGA Caller

To understand the DGA, you must first look at the native code that calls the VM:

At the top of the picture you can see the call of the virtual DGA with the five arguments that are passed:

- **r8d**: The current day, for instance `2` for the 2nd.
- **edx**: The current month, for instance `3` for March.
- **ecx**: The current year, for instance `2019`.

- **rsi**: Domain number, starting with 0.
- **r9**: Memory address that receives the generates domain.

The domain nr `rsi` is set to `r12d` in the first line of the screenshot, with `r12d` being zero. The loop generates exactly 20 domain, until `rsi` reaches 20.

IDA Pro Graph View

The number of assembler lines that the dynamic binary translation generated is 80% less than the number of virtual instructions. Nevertheless, the DGA is still very long, as the following two images show. They show the DGA itself, as well as a function called from it that does date-based seeding.

DGA Main Function

This is the DGA itself:

DGA Seeding

This is the date-based function for seeding:

The DGA could now easily be analyzed with the disassembly graph of IDA, as it reveals the structure and control flow of the functions. The real advantage of IDA in this case, however, is the Hex Rays decompiler. As mentioned before, the DGA uses a lot of optimized integer divisions, so-called divisions by invariant integers. Those are annoying to read in disassembly, but very nicely handled by Hex Ray's decompiler plugin.

IDA Pro Hex Rays

The DGA is reverse-engineered based completely on the output of the Hex Rays decompiler.

Pseudo Days since Epoch

The date function called by the DGA for seeding is analyzed first. It is based on the following assumptions regarding the arguments that it receives:

- `r8d` : The one-based day, so the first day of month is 1.
- `edx` : The zero-based month. That is. January has the value `0` and December the value `11`.
- `ecx` : The four-digit year.

The assumption that the month is zero-based is wrong. The month is passed one-based, so with January equal to 1. The first part of this section analyzes the function as if the assumption about the arguments were correct. The second part then examines the effects of the unexpected values for the month.

First of all the whole output of Hex Rays, with hidden casting and declerations:

```
 signed __int64 __usercall days_since_epoch@<rax>(int month@<edx>, int year@<ecx>, int day@<r8d>)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  retaddr = v4;
  month_0 = month;
  extra_years = month / 12;
  year_f = extra_years + year;
  month_fixed = (-12 * extra_years + month_0);
  if ( month_fixed < 0 )
  {
    month_fixed = (month_fixed + 12);
    --year_f;
  }
  month_fixed_2 = month_fixed;
  day_0_based = (day - 1);
  if ( day - 1 < 0 )
```

```
{
  year_plus_1900 = year_f + 1900;
  do
  {
    month_fixed = (month_fixed - 1);
    if ( --month_fixed_2 < 0 )
    {
      month_fixed = 11LL;
      --year_f;
      --year_plus_1900;
      month_fixed_2 = 11LL;
    }
    is_leap_year = !(year_plus_1900 % 4)
                && (year_plus_1900 != 100 * (year_plus_1900 / 100) || year_plus_1900 == 400 * (year_plus_1900
    *(&stack_ - 125) = a5;
    a5 = *(&stack_ - 125);
    day_0_based = (month_lengths_common_year[month_fixed_2 + 12 * is_leap_year] + day_0_based);
  }
  while ( day_0_based < 0 );
}
month_f = month_fixed;
year_f_plus_1900 = year_f + 1900;
while ( 1 )
{
  year_div4_ = year_f_plus_1900 % 4;
  leap_year = year_div4_
           || year_f_plus_1900 == 100 * (year_f_plus_1900 / 100) && year_f_plus_1900 != 400 * (year_f_plus_190
  *(&stack_ - 125) = a5;
  tmp = *(&stack_ - 125);
  if ( day_0_based < month_lengths_common_year[month_f + 12 * leap_year] )
    break;
  leap_year_ = !year_div4_
            && (year_f_plus_1900 != 100 * (year_f_plus_1900 / 100)
             || year_f_plus_1900 == 400 * (year_f_plus_1900 / 400));
  *(&stack_ - 125) = tmp;
  a5 = *(&stack_ - 125);
  index = month_f++ + 12 * leap_year_;
  day_0_based = (day_0_based - month_lengths_common_year[index]);
  if ( month_f == 12 )
  {
    month_f = 0LL;
    ++year_f;
    ++year_f_plus_1900;
  }
}
years_since_epoch = year_f - 1970;
day_1_based = day_0_based + 1;
```

```
  year_mod4 = year_f % 4;
  year_mod4_is_1 = year_mod4 && year_mod4 < 2;
  days_beg_year_with_rule_div4 = year_mod4_is_1 + years_since_epoch / 4 + 365 * (year_f - 1970);
  year_mod100 = year_f % 100;
  c1 = year_f % 100 && year_mod100 < 70;
  days_beg_year_rule_div100 = -c1 - years_since_epoch / 100 + days_beg_year_with_rule_div4;
  year_mod400 = year_f % 400;
  c2 = year_mod400 && year_mod400 < 370;
  c3 = (1374389535LL * years_since_epoch) >> 32;
  days_in_months = 0LL;
  days_beg_year_with_rule_div400 = c2 + (c3 >> 31) + (c3 >> 7) + days_beg_year_rule_div100;
  for ( i = 0LL; i < month_f; days_in_months = (month_lengths_common_year[i_] + days_in_months) )
  {
    is_leap_year_1 = !year_mod4 && (year_mod100 || !year_mod400);
    *(&stack_ - 125) = tmp;
    tmp = *(&stack_ - 125);
    i_ = i++ + 12 * is_leap_year_1;
  }
  *(&stack_ - 125) = tmp;
  return days_in_months + days_beg_year_with_rule_div400 + day_1_based - 1;
```

If the arguments were correct

If the date components are within the expected range (again, they aren't, but for now we assume that they are), then the calculations at the beginning are omitted, as is the whole if-block (`if(day -1 < 0)`). The `while(1)` block also has no effect, since the `break` statement always triggers.

The code also contains lines with `*(&stack_ - 125)` , these are relics of the temporary stack variable `[rsp-1000]` which the binary translation writes sometimes, see the Binary Translation Section. All lines with `*(&stack_ - 125)` can be removed. This leads to this simplified code:

```
 signed __int64 __usercall days_since_epoch@<rax>(int month@<edx>, int year@<ecx>, int day@<r8d>)
{
  year_f = year;
  day_0_based = (day - 1);
  month_f = month;
  year_f_plus_1900 = year_f + 1900;
  years_since_epoch = year_f - 1970;
  day_1_based = day_0_based + 1;
  year_mod4 = year_f % 4;
  year_mod4_is_1 = year_mod4 && year_mod4 < 2;
  days_beg_year_with_rule_div4 = year_mod4_is_1 + years_since_epoch / 4 + 365 * (year_f - 1970);
  year_mod100 = year_f % 100;
  c1 = year_f % 100 && year_mod100 < 70;
  days_beg_year_rule_div100 = -c1 - years_since_epoch / 100 + days_beg_year_with_rule_div4;
  year_mod400 = year_f % 400;
```

```
  c2 = year_mod400 && year_mod400 < 370;
  c3 = (1374389535LL * years_since_epoch) >> 32;
  days_in_months = 0LL;
  days_beg_year_with_rule_div400 = c2 + (c3 >> 31) + (c3 >> 7) + days_beg_year_rule_div100;
  for ( i = 0LL; i < month_f; days_in_months = (month_lengths_common_year[i_] + days_in_months) )
  {
    is_leap_year_1 = !year_mod4 && (year_mod100 || !year_mod400);
    i_ = i++ + 12 * is_leap_year_1;
  }
  return days_in_months + days_beg_year_with_rule_div400 + day_1_based - 1;
```

The code calculates the days since epoch, i.e., 1 January 1970. First, it determines the number of years since 1970.

```
 year_f = year;
day_0_based = (day - 1);
month_f = month;
year_f_plus_1900 = year_f + 1900;
years_since_epoch = year_f - 1970;
```

Next, the code determines the number of days to the beginning of the provided year since epoch by considering each year divisible by 4 a leap year. So for instance, if the function is called for November 2nd, 2017, then it calculates the days to January 1, 2017.

```
 day_1_based = day_0_based + 1;
year_mod4 = year_f % 4;
year_mod4_is_1 = year_mod4 && year_mod4 < 2;
days_beg_year_with_rule_div4 = year_mod4_is_1 + years_since_epoch / 4 + 365 * (year_f - 1970);
```

Then the code corrects the fact that every year divisible by 100 is not a leap year:

```
  year_mod100 = year_f % 100;
  c1 = year_f % 100 && year_mod100 < 70;
  days_beg_year_rule_div100 = -c1 - years_since_epoch / 100 + days_beg_year_with_rule_div4;
```

And finally the code accounts for the rule that every year divisible by 400 is again a leap year:

```
  year_mod400 = year_f % 400;
  c2 = year_mod400 && year_mod400 < 370;
  c3 = (1374389535LL * years_since_epoch) >> 32;
  days_in_months = 0LL;
  days_beg_year_with_rule_div400 = c2 + (c3 >> 31) + (c3 >> 7) + days_beg_year_rule_div100;
```

Now the code correctly determined the number of days to the beginning of the year. Finally, it uses a loop to add up the days in each elapsed month:

```
 for ( i = 0LL; i < month_f; days_in_months = (month_lengths_common_year[i_] + days_in_months) )
{
  is_leap_year_1 = !year_mod4 && (year_mod100 || !year_mod400);
  i_ = i++ + 12 * is_leap_year_1;
}
```

The `month_length_common_year` lists the number of days per month in a common year, immediately followed by the list of days in a leap year. The term `12 * is_leap_year_1` switches to the leap year array if necessary.

Finally, the code adds up the days to the beginning of the year, the days in past months, and the current day, subtracting `1` to get the days since epoch:

```
  return days_in_months + days_beg_year_with_rule_div400 + day_1_based - 1;
```

Impact of the actual arguments

The above code works perfectly fine when the months are zero-based. However, the function *RtlTimeToTimeFields* is used by the caller to get the date values, see DGA caller.This function returns months from 1 to 12. What happens to the epoch function, if it runs on these dates?

Case 1: not December, not the end of the month

Dates that are not in December and not after the 28th of the month result in the number of days to the corresponding day in the next month. So for instance (with dates `dd.mm.yyyy`):

real date	treated as	result
28.3.2019	28.4.2019	0x465E
1.9.2017	1.10.2017	0x4420
1.1.2014	1.2.2014	0x30A1
30.11.2019	30.12.2019	0x4754

Case 2: not December, end of the month

Shifting dates to the next month will cause a problem if the day in the next month does not exist. For example, March 31 would shift to April 31, which does not exist. In those instances, the `while(1)` loop that we skipped before comes into effect:

```
while ( 1 )
 {
   year_div4_ = year_f_plus_1900 % 4;
   leap_year = year_div4_
            || year_f_plus_1900 == 100 * (year_f_plus_1900 / 100) && year_f_plus_1900 != 400 * (year_f_plus_190
   if ( day_0_based < month_lengths_common_year[month_f + 12 * leap_year] )
     break;
   leap_year_ = !year_div4_
             && (year_f_plus_1900 != 100 * (year_f_plus_1900 / 100)
              || year_f_plus_1900 == 400 * (year_f_plus_1900 / 400));
   index = month_f++ + 12 * leap_year_;
   day_0_based = (day_0_based - month_lengths_common_year[index]);
   if ( month_f == 12 )
   {
     month_f = 0LL;
     ++year_f;
     ++year_f_plus_1900;
   }
 }
```

It has the test

```
day_0_based < month_lengths_common_year[month_f + 12 * leap_year]
```

which checks, if the day exists in the current month. If it does not, then the date properly overflows to the next month. The code could even shift dates by several months, e.g., April 91 to June 30. Leap years are also correctly handled, see the last to rows in the following table:

real date	treated as	result
31.3.2019	1.5.2019	0x4661
30.11.2019	30.12.2019	0x4754
31.1.2019	3.3.2019	0x4626
31.1.2020	2.3.2020	0x4793

Case 3: December

For dates in December, the start of the function is relevant:

```
  month_o = month;
extra_years = month / 12;
year_f = extra_years + year;
```

```
  month_fixed = (-12 * extra_years + month_o);
  if ( month_fixed < 0 )
  {
    month_fixed = (month_fixed + 12);
    --year_f;
  }
```

The variable `extra_years` is `1`, which increments the year by one. The month value is reduced by 12 (`-12 * extra_years + month_o`), i.e., becomes `0` which stands for January. We therefore get:

real date	treated as	result
6.12.2019	6.1.2020	0x475B

The DGA Function

The DGA looks as follows. Again, the output is very long, although only few lines are superfluous. The components of the algorithm are examined individually afterward.

```
 __int64 __usercall dga@<rax>(__int64 months@<rdx>, __int64 year@<rcx>, __int64 domaint_output@<r9>, int days@<r
{
  domain_out = domain_output;
  vars30 = &vars38;
  vars28 = a4;
  vars20 = a3;
  vars18 = a6;
  vars10 = a7;
  vars8 = a8;
  j = 0LL;
  domain = a5;
  v20 = year;
  random_numbers = 0;
  magic_number = 0xDAFE02C;
  days_since_1970_broken = days_since_epoch(months, year, days);
  consonants = *pConsonants;
  LOBYTE(v20) = v20 - 108;
  retaddr = v20;
  i = 0;
  v25 = &v72;
  seed_value = domain_nr / 3 + days_since_1970_broken;
  if ( !*pConsonants )
  {
    consonants = (ExAllocatePool)(&v72, domain, 23LL, 0LL);
    *pConsonants = consonants;
    if ( decrypt_consonants )
    {
```

```
    key = 0x3365841C;
    key_index = 0LL;
    addr_encrypted_consonants = &encrypted_consonants;
    do
    {
      key_index_1 = key_index;
      ++consonants;
      ++addr_encrypted_consonants;
      key_byte = *(&key + key_index);
      *(consonants - 1) = *(addr_encrypted_consonants - 1) ^ *(&key + key_index);
      key_index = (key_index + 1) & 0x80000003;
      *(&key + key_index_1) = 2 * key_byte ^ (key_byte >> 1);
      if ( key_index < 0 )
        key_index = ((key_index - 1) ^ 0xFFFFFFFC) + 1;
    }
    while ( addr_encrypted_consonants < &encrypted_consonants_end );
    consonants = *pConsonants;
  }
}
vowels = *pVowels;
if ( !*pVowels )
{
  vowels = (ExAllocatePool)(&v72, domain, *pVowels + 7LL, 0LL);
  *pVowels = vowels;
  if ( decrypt_vowels )
  {
    key = -967459448;
    key_index_2 = 0LL;
    addr_encrypted_vowels = &encrypted_vowels;
    do
    {
      key_index_3 = key_index_2;
      ++vowels;
      ++addr_encrypted_vowels;
      key_byte_1 = *(&key + key_index_2);
      *(vowels - 1) = *(addr_encrypted_vowels - 1) ^ *(&key + key_index_2);
      key_index_2 = (key_index_2 + 1) & 0x80000003;
      *(&key + key_index_3) = 2 * key_byte_1 ^ (key_byte_1 >> 1);
      if ( key_index_2 < 0 )
        key_index_2 = ((key_index_2 - 1) ^ 0xFFFFFFFC) + 1;
    }
    while ( addr_encrypted_vowels < &encrypted_vowels_end );
    vowels = *pVowels;
  }
}
tlds = pTLDs;
if ( !pTLDs )
```

```
{
  tlds = (ExAllocatePool)(&v72, domain, pTLDs + 38, 0LL);
  pTLDs = tlds;
  if ( decrypt_tlds )
  {
    key = 2131189013;
    key_index_4 = 0LL;
    addr_encrypted_tlds = &encrypted_tlds;
    do
    {
      v44 = key_index_4;
      ++tlds;
      ++addr_encrypted_tlds;
      key_byte_2 = *(&key + key_index_4);
      *(tlds - 1) = *(addr_encrypted_tlds - 1) ^ *(&key + key_index_4);
      key_index_4 = (key_index_4 + 1) & 0x80000003;
      *(&key + v44) = 2 * key_byte_2 ^ (key_byte_2 >> 1);
      if ( key_index_4 < 0 )
        key_index_4 = ((key_index_4 - 1) ^ 0xFFFFFFFC) + 1;
    }
    while ( addr_encrypted_tlds < &encrypted_tlds_end );
    tlds = pTLDs;
  }
}
tlds_1 = tlds;
v30 = &v72 - tlds;
do
{
  v67 = *tlds_1;
  tlds_1 = (tlds_1 + 1);
  *(tlds_1 + v30 - 1) = v67;
}
while ( v67 );
if ( tlds )
{
  (ExFreePool)(&v72, domain, v30, tlds);
  pTLDs = 0LL;
}
v17 = 1LL;
v73 = &v72;
if ( v72 )
{
  do
  {
    if ( *v25 == 44 )
    {
      *v25 = 0;
```

```
      *&tld_array[8 * v17 - 49] = v25 + 1;
      v17 = (v17 + 1);
    }
    v25 = (v25 + 1);
  }
  while ( *v25 );
}
counter_ = domain_nr;
round_seed_to_nearset_10 = 10 * (seed_value / 0xA);
seed_value = round_seed_to_nearset_10;
HIWORD(v39) = HIWORD(round_seed_to_nearset_10);
LOWORD(v39) = ((0xDAFE02Cu >> domain_nr) * (domain_nr - 1)) * round_seed_to_nearset_10;
LOBYTE(v39) = (v39 & 1) + 8;
domain_length = v39;
if ( v39 > 0 )
{
  v41 = BYTE1(seed_value);
  addr_random_numbers = &the_random_numbers;
  do
  {
    ip1 = i++;
    v50 = (ip1 >> 31) & 3;
    v51 = v50 + ip1;
    v52 = (v51 >> 2);
    v53 = (v51 & 3) - v50;
    if ( v53 )
    {
      v54 = v53 - 1;
      if ( v54 )
      {
        v55 = v54 - 1;
        if ( v55 )
        {
          if ( v55 == 1 )
          {
            ++random_numbers;
            v56 = (round_seed_to_nearset_10 << v52) ^ (v41 >> v52);
            v57 = v52;
            counter_ = domain_nr;
            addr_random_numbers = (addr_random_numbers + 1);
            *(addr_random_numbers - 1) = v56 * (*(&magic_number + v57) & 0xF) * (domain_nr + 1);
          }
          else
          {
            counter_ = domain_nr;
          }
        }
```

```
      else
      {
        ++random_numbers;
        v15 = (v41 << v52) ^ (round_seed_to_nearset_10 >> v52);
        v16 = v52;
        counter_ = domain_nr;
        addr_random_numbers = (addr_random_numbers + 1);
        *(addr_random_numbers - 1) = v15 * (*(&magic_number + v16) >> 4) * (domain_nr + 1);
      }
    }
    else
    {
      v31 = v52;
      v32 = v52;
      counter_ = domain_nr;
      ++random_numbers;
      addr_random_numbers = (addr_random_numbers + 1);
      *(addr_random_numbers - 1) = (*(&magic_number + v31) & 0xF) * (retaddr << v32) * (domain_nr + 1);
    }
  }
  else
  {
    v63 = v52;
    v64 = v52;
    counter_ = domain_nr;
    ++random_numbers;
    addr_random_numbers = (addr_random_numbers + 1);
    *(addr_random_numbers - 1) = (*(&magic_number + v63) >> 4) * (retaddr >> v64) * (domain_nr + 1);
  }
}
while ( random_numbers < domain_length );
domain = domain_out;
if ( domain_length > 0 )
{
  while ( 1 )
  {
    v34 = j;
    j = (j + 1);
    v35 = *(&the_random_numbers + v34);
    if ( (v35 & 0x80u) == 0 )
      break;
    *(++domain - 1) = *(consonants + (v35 % 21));
    if ( j >= domain_length )
      goto append_tld;
    v36 = j;
    j = (j + 1);
    *(++domain - 1) = *(vowels + (*(&the_random_numbers + v36) % 5));
```

```
        if ( j >= domain_length )
          goto append_tld;
        r = *(&the_random_numbers + j);
        LOBYTE(r) = r & 64;
        if ( r )
        {
          *(++domain - 1) = *(vowels + (r % 5));
_addr_FFFFF880058745FC:
          j = (j + 1);
        }
        if ( j >= domain_length )
          goto append_tld;
      }
      *domain = *(vowels + (v35 % 5));
      domain += 2;
      *(domain - 1) = *(consonants + (*(&the_random_numbers + j) % 21));
      goto _addr_FFFFF880058745FC;
    }
  }
append_tld:
  *domain = '.';
  tld = *&tld_array[8 * ((counter_ ^ round_seed_to_nearset_10 ^ 0xDAFE02C) % 9) - 49];
  dmtld = domain - tld;
  do
  {
    result = *tld;
    tld = (tld + 1);
    *(tld + dmtld) = result;
  }
  while ( result );
  return result;
}
```

Seeding

The main part of seeding is in function `days_since_epoch`. This value is combined with the domain counter and rounded to 10 day intervals:

```
  days_since_1970_broken = days_since_epoch(months, year, days);
  ...
  seed_value = domain_nr / 3 + days_since_1970_broken;
  ...
  round_seed_to_nearset_10 = 10 * (seed_value / 10);
  seed_value = round_seed_to_nearset_10;
```

The seed stays the same for ten days, in most cases. It does not really matter that the number of days since epoch are calculated wrong, this value is only used for seeding and required to change daily. This is the case for almost all days, except for a few edge cases, where two days can have the same seed (for example January 29, 2019 and February 1, 2019 return the same value). The broken calculation can also shorten or extend the 10 day window. The longest windows is 13 days at the end of January, e.g., 2019-01-23 – 2019-02-04. Windows at the end of February are shorter, for example the 7 day window 2019-02-25 – 2019-03-03. In rare cases, the window is just 1 day long, happening next 2025-01-31.

Seeding also uses a magic number:

```
magic_number = 0xDAFE02C;
```

According to F-Secure, this means it is Pitou version 33. They list 0xDAFE02D as a second seed for version 31.

Encrypted Strings

The DGA uses three encrypted strings: vowels, consonants, and TLDs. The DGA decrypts these strings the first time it is run. The encryption is a rolling XOR with a four-byte key, which is updated every loop according to `key = (key<<1) ^ (key >>1)`:

```
if ( !*pConsonants )
{
  consonants = (ExAllocatePool)(&v72, domain, 23LL, 0LL);
  *pConsonants = consonants;
  if ( decrypt_consonants )
  {
    key = 0x3365841C;
    key_index = 0LL;
    addr_encrypted_consonants = &encrypted_consonants;
    do
    {
      key_index_1 = key_index;
      ++consonants;
      ++addr_encrypted_consonants;
      key_byte = *(&key + key_index);
      *(consonants - 1) = *(addr_encrypted_consonants - 1) ^ *(&key + key_index);
      key_index = (key_index + 1) & 0x80000003;
      *(&key + key_index_1) = 2 * key_byte ^ (key_byte >> 1);
      if ( key_index < 0 )
        key_index = ((key_index - 1) ^ 0xFFFFFFFC) + 1;
    }
    while ( addr_encrypted_consonants < &encrypted_consonants_end );
    consonants = *pConsonants;
  }
}
```

Second Level Domain Length

The length of the second level domain is calculated as follows:

```
 counter_ = domain_nr;
round_seed_to_nearset_10 = 10 * (seed_value / 0xA);
seed_value = round_seed_to_nearset_10;
HIWORD(v39) = HIWORD(round_seed_to_nearset_10);
LOWORD(v39) = ((0xDAFE02Cu >> domain_nr) * (domain_nr - 1)) * round_seed_to_nearset_10;
LOBYTE(v39) = (v39 & 1) + 8;
domain_length = v39;
```

This leads to lengths of 8 or 9 characters.

Random Numbers

The seed is converted to random numbers. The seed is viewed as a 16bit value, which splits into four 4-bit values. These values are then used to generate the letters of the domain. Because the more significant bits of the seed change slowly, the domains mostly change at letter positions 3,4 and 7,8. For example, these are domains from June 1, June 10, and June 20, 2019:

```
 zuoezaxa�.name
zuopabma.org
zuojabba.mobi
```

Why the `�` character in the domain `zuoezaxa�.name`? Pitou contains a severe bug. Even if the length of the second level domain is picked out to be 9 characters, only 8 random numbers are calculated. The 9th is read from undefined memory. This means that the last character of the sld is undetermined. Domains with only 8 second level domain characters are ok.

Picking Letters

Two arrays provide the letters of the second level domain: a list of vowels (`aeiou`) and a list of consonants (`bcdfghjklmnpqrstvwxyz`). They are picked somewhat alternately to produce more natural-sounding names.

Picking a TLD

The TLD is also picked from a set of hardcoded list: `com` , `org` , `biz` , `net` , `info` , `mobi` , `us` , `name` , `me`

Reimplementation in Python

The DGA is pretty messy, an even the reimplementation in Python is challenging to read.

```
 import argparse
from datetime import date, datetime, timedelta
```

```
from calendar import monthrange

def date2seed(d):
    year_prime = d.year
    month_prime = (d.month + 1)
    day_prime = d.day

    if month_prime > 12:
        month_prime -= 12
        year_prime += 1

    _, monthdays = monthrange(year_prime, month_prime)
    if day_prime > monthdays:
        month_prime += 1
        day_prime -= monthdays

    if month_prime > 12:
        month_prime -= 12
        year_prime += 1

    date_prime = date(year_prime, month_prime, day_prime)
    epoch = datetime.strptime("1970-01-01", "%Y-%m-%d").date()
    return (date_prime - epoch).days

def dga(year, seed, counter, magic):
    seed_value = 10*( (counter//3 + seed) // 10)
    year_since = year - 1900
    random_numbers = []

    a = (magic >> counter)
    b = (counter - 1) & 0xFF
    d = a*b & 0xFF
    e = d*seed_value
    sld_length = 8 + (e & 1)

    magic_list = []
    for i in range(4):
        magic_list.append((magic >> (i*8)) & 0xFF)
    for i in range(8):
        imod = i % 4
        idiv = i // 4
        b1 = (seed_value >> 8) & 0xFF
        b0 = seed_value & 0xFF
        if imod == 0:
            m = magic_list[idiv] >> 4
            f = (year_since >> idiv)
        elif imod == 1:
```

```
        m = magic_list[idiv] & 0xF
        f = (year_since << idiv)
    elif imod == 2:
        m = magic_list[idiv] >> 4
        f = (b1 <<  idiv) ^ (b0 >> idiv)
    elif imod == 3:
        m = magic_list[idiv] & 0xF
        f = (b0 <<  idiv) ^ (b1 >> idiv)
    cp = (counter + 1)
    r = (m*f & 0xFF) *cp
    random_numbers.append(r & 0xFF)
random_numbers.append(0xE0)
r = random_numbers

vowels = "aeiou"
consonants = "bcdfghjklmnpqrstvwxyz"
sld = ""

while True:
    x = r.pop(0)
    if x & 0x80:
        sld += consonants[x % len(consonants)]
        if len(sld) >= sld_length:
            break
        x = r.pop(0)
        sld += vowels[x % len(vowels)]
        if len(sld) >= sld_length:
            break

        x = r[0]
        if x & 0x40:
            r.pop(0)
            sld += vowels[x % len(vowels)]
            if len(sld) >= sld_length:
                break
    else:
        sld += vowels[x % len(vowels)]
        x = r.pop(0)
        sld += consonants[x % len(consonants)]
        if len(sld) >= sld_length:
            break

tlds = ['com', 'org', 'biz', 'net', 'info', 'mobi', 'us', 'name', 'me']


q = (counter ^ seed_value ^ magic)  & 0xFFFFFFFF
tld = tlds[q % len(tlds)]
```

```
    if len(sld) > 8:
        lc = sld[-1]
        sld = sld[:-1]
        if lc in consonants:
            sld_c = [sld + c for c in consonants]
        else:
            sld_c = [sld + c for c in vowels]
        return [s + "." + tld for s in sld_c]
    else:
        return sld + "." + tld

if __name__=="__main__":
    parser = argparse.ArgumentParser(description="DGA of Pitou")
    parser.add_argument("-d", "--date",
        help="date for which to generate domains, e.g., 2019-04-09")

    parser.add_argument("-m", "--magic", choices=["0xDAFE02D", "0xDAFE02C"],
            default="0xDAFE02C", help="magic seed")
    args = parser.parse_args()

    if args.date:
        d = datetime.strptime(args.date, "%Y-%m-%d")
    else:
        d = datetime.now()

    for c in range(20):
        seed = date2seed(d)
        domains = dga(d.year, seed, c, int(args.magic, 16))
        if type(domains) == str:
            print(domains)
        else:
            l = len(domains[0]) + 1
            print(l*"-" + "+")
            for i, domain in enumerate(domains):
                if i == len(domains)//2:
                    label = "one of these"
                    print("{} +--{}".format(domain, label))
                else:
                    print("{} |".format(domain))
            print(l*"-" + "+")
```

For all domains with sld length 9, the code print all possible domain names (see the bug in Random Number):

```
 ▶ python3 dga.py -d 2019-06-10
-------------+
```

```
koupoalab.me |
koupoalac.me |
koupoalad.me |
koupoalaf.me |
koupoalag.me |
koupoalah.me |
koupoalaj.me |
koupoalak.me |
koupoalal.me |
koupoalam.me |
koupoalan.me +--one of these
koupoalap.me |
koupoalaq.me |
koupoalar.me |
koupoalas.me |
koupoalat.me |
koupoalav.me |
koupoalaw.me |
koupoalax.me |
koupoalay.me |
koupoalaz.me |
-------------+
```

Characteristics

The following table summarizes the properties of Pitou's DGA.

property	**value**
type	TDD (time-dependent-deterministic), to some extend TDN (time-dependent non-deterministic)
generation scheme	bit-shifting the seed
seed	magic value + current date
domain change frequency	mostly every 10 days, very rarly after 1 day, sometimes after 13 days
domains per day	20
sequence	sequential
wait time between domains	None
top level domains	`com`, `org`, `biz`, `net`, `info`, `mobi`, `us`, `name`, `me`

property	value
second level characters	a-z
second level domain length	8 or 9

Comparison with Public Reports.

For all the reports listed under Previous Work, I checked that any domains mentioned are actually covered by the DGA in this blog post. You can find the list of domains from 2015 - 2021 for the two seeds here: 0xdafe02c and 0xdafe02d. All domains from the reports are covered by the DGA as I reimplemented it.

Pitou - The “silent” resurrection of the notorious Srizbi kernel spambot

The report by f-Secure does not list any Pitou DGA domains.

Bootkits are not dead. Pitou is back!

The report by C.R.A.M from Januar 15, 2018 lists four domains:

domain	seed	first generated	last generated
unpeoavax.mobi	0xDAFE02C	2017-10-04	2017-10-13
ilsuiapay.us	0xDAFE02C	2017-10-04	2017-10-13
ivbaibja.net	0xDAFE02C	2017-10-08	2017-10-17
asfoeacak.info	0xDAFE02C	2017-10-08	2017-10-17

Rig Exploit Kit sends Pitou.B Trojan

The SANS Internet Storm Center diary entry from Brad Duncan, published 25 June 2019, links to a Pitou PCAP on the excellent malware-traffic-analysis blog by the same author.

domain	seed	first generated	last generated
rogojaob.info	0xDAFE02C	2019-06-23	2019-07-01
wiejlauas.info	0xDAFE02C	2019-06-18	2019-06-27
yoevuajas.us	0xDAFE02C	2019-06-22	2019-06-30
ijcaiatas.name	0xDAFE02C	2019-06-19	2019-06-28
piiaxasas.com	0xDAFE02C	2019-06-19	2019-06-28

domain	seed	first generated	last generated
caoelasas.name	0xDAFE02C	2019-06-22	2019-06-30
naaleazas.net	0xDAFE02C	2019-06-23	2019-07-01
epcioalas.info	0xDAFE02C	2019-06-20	2019-06-29
oltaeazas.mobi	0xDAFE02C	2019-06-20	2019-06-29
suudaacas.org	0xDAFE02C	2019-06-18	2019-06-27
giazfaeas.me	0xDAFE02C	2019-06-21	2019-06-30
zuojabba.mobi	0xDAFE02C	2019-06-18	2019-06-27
unufabub.net	0xDAFE02C	2019-06-21	2019-06-30
ufayubja.me	0xDAFE02C	2019-06-19	2019-06-28
huoseavas.name	0xDAFE02C	2019-06-17	2019-06-26
irifyara.com	0xDAFE02C	2019-06-21	2019-06-30
vaxeiayas.mobi	0xDAFE02C	2019-06-22	2019-06-30
kooovaqas.biz	0xDAFE02C	2019-06-23	2019-07-01
dienoalas.us	0xDAFE02C	2019-06-17	2019-06-26
amlivaias.us	0xDAFE02C	2019-06-20	2019-06-29

Brad Duncan also wrote a second blog post on another Pitou sample, again he provides a PCAP with these Pitou domains:

domain	seed	first generated	last generated
amlivaias.us	0xDAFE02C	2019-06-20	2019-06-29
piiaxasas.com	0xDAFE02C	2019-06-19	2019-06-28
zuojabba.mobi	0xDAFE02C	2019-06-18	2019-06-27
vaxeiayas.mobi	0xDAFE02C	2019-06-22	2019-06-30
giazfaeas.me	0xDAFE02C	2019-06-21	2019-06-30
oltaeazas.mobi	0xDAFE02C	2019-06-20	2019-06-29
rogojaob.info	0xDAFE02C	2019-06-23	2019-07-01

domain	seed	first generated	last generated
irifyara.com	0xDAFE02C	2019-06-21	2019-06-30
ufayubja.me	0xDAFE02C	2019-06-19	2019-06-28
naaleazas.net	0xDAFE02C	2019-06-23	2019-07-01
dienoalas.us	0xDAFE02C	2019-06-17	2019-06-26
kooovaqas.biz	0xDAFE02C	2019-06-23	2019-07-01
suudaacas.org	0xDAFE02C	2019-06-18	2019-06-27
wiejlauas.info	0xDAFE02C	2019-06-18	2019-06-27
unufabub.net	0xDAFE02C	2019-06-21	2019-06-30
yoevuajas.us	0xDAFE02C	2019-06-22	2019-06-30
epcioalas.info	0xDAFE02C	2019-06-20	2019-06-29
huoseavas.name	0xDAFE02C	2019-06-17	2019-06-26
caoelasas.name	0xDAFE02C	2019-06-22	2019-06-30
ijcaiatas.name	0xDAFE02C	2019-06-19	2019-06-28

Trojan.Pitou.B

The technical description of Pitou by Symantec lists 20 domains. Note that these use these stem from the other seed `0xDAFE02D`:

domain	seed	first generated	last generated
ecqevaaam.net	0xDAFE02D	2016-01-06	2016-01-15
yaefobab.info	0xDAFE02D	2016-01-09	2016-01-18
alguubub.mobi	0xDAFE02D	2016-01-14	2016-01-23
dueifarat.name	0xDAFE02D	2016-01-14	2016-01-23
ehbooagax.info	0xDAFE02D	2016-01-13	2016-01-22
igocobab.com	0xDAFE02D	2016-01-08	2016-01-17
utleeawav.us	0xDAFE02D	2016-01-14	2016-01-23
wuomoalan.us	0xDAFE02D	2016-01-06	2016-01-15

domain	seed	first generated	last generated
coosubca.mobi	0xDAFE02D	2016-01-09	2016-01-18
seeuvamap.mobi	0xDAFE02D	2016-01-06	2016-01-15
hioxcaoas.me	0xDAFE02D	2016-01-15	2016-01-24
upxoearak.biz	0xDAFE02D	2016-01-07	2016-01-16
oxepibib.net	0xDAFE02D	2016-01-07	2016-01-16
ruideawaf.us	0xDAFE02D	2016-01-08	2016-01-17
agtisaib.info	0xDAFE02D	2016-01-07	2016-01-16
neaqaaxag.org	0xDAFE02D	2016-01-08	2016-01-17
pooexaxaq.org	0xDAFE02D	2016-01-15	2016-01-24
iyweialay.net	0xDAFE02D	2016-01-13	2016-01-22
laagubha.com	0xDAFE02D	2016-01-15	2016-01-24
viurjaza.name	0xDAFE02D	2016-01-09	2016-01-18

Appendix - Virtual ISA

Not all variants are used by Pitou, and not all instructions are fully implemented. I created the illustrations with a Python script created for this purpose.

0x00 - XOR

Pops two values from the stack, **XOR**s them, and pushes the result back on the stack.

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x00` , `0x2F` or `0x35`

field	bits	description
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x01 - M2E and M2V

Gets a value (and extra value) from the stack, and moves the value to the offset given by the extra slot from the stack (`M2E` - move to extra) or to the offset from the instruction (`M2V` - move to value).

field	bits	description
has size	1	`0` : no size and segment field, defaults to Dword, `1` : size and segment value follows (1 byte)
move to offset	1	`0` : move to address from stack, `1` : move to relative offset given by `offset`
opcode	6	must be `0x01`
segment	6	(optional) `0` or `6` : data segment, `4` : fs, `5` : gs
size	2	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord
offset	64	(optional) destination of the mov instruction

0x02 - JMP, CALL and RET

Pop an address from the stack. Then **jump** or **call** this address. The flag `stay in vm` determines, whether the call is to bytecode or to native code. The difference between **jump** or **call** affects the native stack: the **call** pushes the (native) instruction pointer on the native stack. This allows calls to native code to return to the VM handler. With the help of the decompiler, some `JMP` statements are renamed to `RET` (see *Decompiler* section).

field	bits	description
call or jump	1	`0` : jump, `1` : call
stay in vm	1	`0` : external (native) address, `1` : vm address
opcode	6	must be `0x02` , `0x24` or `0x25`

0x03 - POPD

Increase the stack pointer, i.e., **discard** stack elements.

field	bits	description
multiple	1	`0` : discard 1 element, `1` : discard *n* elements given as operand (1 byte)
opcode	6	must be `0x03` , `0x1D` , `0x2A` or `0x3A`
nr	8	(optional) number of stack elements to discard

0x04 - DREFH

Dereference an address from the stack or given as an argument. The resulting address and the address itself are stored on the stack (as value and extra data respectively).

field	bits	description
has size	1	`0` : no size and segment field, defaults to Dword, `1` : size and segment value follows (1 byte)
dref from stack or immediate	1	`0` : dereference from stack, `1` : dereference from argument
opcode	6	must be `0x04` , `0x1E` , `0x23` or `0x33`
segment	3	(optional) `0` or `6` : data segment, `4` : fs, `5` : gs
signed	1	(optional) `0` : dereferenced value is unsigned, `1` : dereferenced value is signed
size	2	(optional) size of the address, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord
address	64	(optional) address of the value

0x05 - SBB

Pops two values from the stack, **subtracts** the first from the second and pushes the result back on the stack. Also subtracts the carry flag, i.e., simulates assembly instruction `SBB` .

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x05` , `0x2E` or `0x32`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x06 - IF x → y

Conditional jump to a target address (relative or absolute). Can consider one or two sets of flags from the FLAGS register.

field	bits	description
binary comparison	1	`0` : use one flag, `1` : use two flags
and / or	1	how to treat two flags, `0` : OR operation, `1` : AND operation
opcode	6	must be `0x06`
operation 1	2	`0` : true if at least one but not all flags set, `1` : true if no flag set, `2` : true if all or no flags set, `3` : true if all flags set
bitmask 1	3	`0` : CF, `1` : PF, `2` : AF, `3` : ZF, `4` : SF, `5` : OF
operation 2	2	(optional) *see operation 1*
bitmask 2	3	(optional) *see bitmask 1*
destination	64	(optional) relative destination address

For example, the conditional jump `JLE` is taken when

```
IF ZF OR NOT (SF & OF) AND (SF | OF)
```

Which can be split into two conditions

1. `ZF` : `bitmask 1` set to `001000` , with `operation 1` set to `3` .
2. `NOT (SF & OF) AND (SF | OF)` : `bitmask 2` set to `110000` , with `operation 2` set to `0` .

The two conditions are combined with `OR`, i.e., `and / or` is set to `0`.

The handler could check very complex combinations of flags. At least for the DGA, however, only the following checks are used which all can be mapped to an x86/x64 assembly instruction:

comparison	and / or	op 1	bm 1	op 2	bm 2	result	assembly
unary	-	1	010000	-	-	`NOT SF`	`JNS`
unary	-	3	010000	-	-	`SF`	`JS`
unary	-	2	110000	-	-	``(SF & OF) OR NOT (SF	OF)``
unary	-	1	001000	-	-	`NOT ZF`	`JNZ`
unary	-	0	110000	-	-	``NOT (SF & OF) AND (SF	OF)``
binary	or	3	010000	0	110000	``ZF OR NOT (SF & OF) AND (SF	OF)``
unary	-	3	010000	-	-	`ZF`	`JZ`
unary	-	3	000001	-	-	`CF`	`JB`

0x07 - NOP

Does not affect the VM.

field	bits	description
opcode	6	must be `0x07` or `0x37`

0x08 - MUL

Copy the two virtual registers `rax` and `rdx` to the native counterpart. Pop a value from the stack and run **MUL**, which multiplies the stack value with the native `rax`. Then the two native registers `rax` and `rdx` are copied back to their respective virtual registers.

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x08` , `0x22` or `0x2D`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x09 - DIV

Same as MUL, except **DIV** is run instead of MUL.

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x09`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x0A - STACKSWP

Swap the value of the two topmost stack elements. The *extra* field is not affected by this instruction, so probably the instruction will only be applied to stack elements where the extra field is unused.

field	bits	description
opcode	6	must be `0x0A` or `0x3C`

0x0B - CWD, CDW …

Extends the sign bit of `al`, `ax`, `eax` or `rax`.

field	bits	description
has size	1	`0`: no size field, defaults to `CWDE` for 32-bit and `CDQ` for 64-bit, `1`: size value follows (1 byte)
64-bit	1	`0`: use 32 bit conversions, `1`: use 64-bit conversions
opcode	6	must be `0x0B`, `0x20` or `0x36`
size	8	(optional) size of the operation, see table below

The 8 combinations of flags `has size`, `64-bit` and `size` map to these `x64` instructions. The colum *extension* shows which register is sign extended to what register.

has size	64-bit	size	mnemonic	extension
0	0	x	`CWDE`	`ax` to `eax`
1	0	1	`CBW`	`al` to `ax`
1	0	2	`CWDE`	`ax` to `eax`
1	0	3	`CDQE`	`eax` to `rax`
0	1	x	`CDQ`	`eax` to `edx:eax`
1	1	1	`CWD`	`ax` to `dx:ax`
1	1	2	`CDQ`	`eax` to `edx:eax`
1	1	3	`CQO`	`rax` to `rdx:rax`

0x0C - NOP

Not implemented, extracts three values but does nothing

Encoding

field	bits	description
opcode	6	must be `0x0C` or `0x29`
c	3	an unused 3 bit value
b	3	an unused 3 bit value
a	2	an unused 2 bit value

0x0D - IMUL

Pop two values from the stack, **multiply** them as **signed** values, and push the result back on the stack.

Encoding

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x0D` or `0x3E`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x0E - NEG

Pop a value from the stack and negate (**NEG**) it, then push the result back on the stack.

Encoding

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x0E`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

Decompiler

See unary

0x0F - SUB

Pop two values from the stack, **subtract** them (without borrow) and push the result back on the stack.

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x0F` , `0x34` or `0x39`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x10 - OR

Pops two values from the stack, **OR**s them and pushes the result back on the stack.

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x10` , `0x21` , `0x26` or `0x30`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x11 - INC and DEC

Pops a value from the stack and increments (**INC**) or decrements (**DEC**) it, then pushes the result back on the stack.

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
dec or inc	1	`0` : decrement, `1` : increment
opcode	6	must be `0x11` or `0x38`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x12 - PUSH and POP

Push or **pop** a register to or from the stack. The push can either push the value or the address of the register to the stack, while the extra field is always set to the address of the register.

field	bits	description
push or pop	1	`0` : pop from stack to register, `1` : push register to stack
push address	1	(optional) `0` : push register value, `1` : push register address
opcode	6	must be `0x12` , `0x27` or `0x2B`
not used	1	(unused)
not used	2	(unused)
register	5	register index, see below

The register is referenced by an index: 0: `rax` , 1: `rcx` , 2: `rdx` , 3: `rbx` , 4: `rsp` , 5: `rbp` , 6: `rsi` , 7: `rdi` , 8: `r8` , 9: `r9` , 10: `r10` , 11: `r11` , 12: `r12` , 13: `r13` , 14: `r14` , 15: `r15` , 16: `eflags` , 17: `field_88` , 18: virtual instruction pointer, 19: virtual stack base, 20: virtual stack pointer

0x13 - CMP

Pop two values from the stack, **compare** them, and set the virtual `RFLAGS` (`CF` , `PF` …) accordingly. Push the last value back on the stack, which is then often discarded in the next virtual instruction. The following snippet from the disassembly shows how `r11` is compared to `Ch` . Note that the address of `r11` is pushed and then dereferenced in the following instruction to get `r11` on the stack, altough there is also a version of `PUSH` that pushes the value itself, without requiring the `DRERH` instruction.

```
 E7 BC            PUSH qword (addr(r11), addr(r11))
9E 54            DREFH qword
98 57 5B         PUSH byte Ch
BB 54            CMP
3A               POPD
```

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x13` or `0x3B`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x14 - SET1

Set a virtual flag to `1` . I did not examine how this flag is used, it had no influence on reversing the DGA.

field	bits	description
opcode	6	must be `0x14` or `0x3D`

0x15 - xDIAGy

Get (`MDIAG` / `MDIAGA`) or pop (`PDIAG` / `PDIAGA`) a value from the stack. Then either move the value (`MDIAG` / `PDIAG`) or the address of the value (`MDIAGA` / `PDIAGA`) to the extra part on top of the stack. Hopefully the following picture can help to illustrate the complicated instructions:

field	bits	description
pop or get	1	`0` : variants `P*` that pop the value from the stack, `1` : variants `M*` that get the value from stack
use address	1	`0` : move the value to the extra field, `1` : move the address of the value to the extra field.
opcode	6	must be `0x15`

0x16 - STATE

Set the flag `state1` , and either set or unset `state2` . The two states together define the addressing mode.

field	bits	description
value	1	`0` : set `state2` to `0` , `1` : set `state2` to `1`
opcode	6	must be `0x16` or `0x2C`

0x17 - ADD

Pop two values from the stack, **add** them and push the result back on the stack.

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x17`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x18 - PUSH

Push the value given as the operand on the stack. The operand can be 1, 2, 4, or 8 bytes long and can be signed or unsigned.

field	bits	description
has size	1	`0` : no size and signed field, defaults to Dword and *unsigned* , `1` : 1 sign flag and 2 bit size value follow
opcode	6	must be `0x18` or `0x31`
signed	1	(optional) `0` : the `value` is unsigned, `1` : the `value` is signed, i.e., the value needs to be sign extended if not 64 bit already
size	2	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

field	bits	description
value	8, 16, 32 or 64	immediate value to push on the stack

0x19 - P2E and P2V

Pop a value (and extra value) from the stack, and **move** the value to the extra value from the stack (`P2E` - pop to extra) or to the offset from the instruction (`P2V` - pop to value).

field	bits	description
has size	1	`0` : no size and segment field, defaults to Dword, `1` : size and segment value follows (1 byte)
move to offset	1	`0` : move to address from stack, `1` : move to relative offset given in `offset`
opcode	6	must be `0x19`
segment	6	(optional) `0` or `6` : data segment, `4` : fs, `5` : gs
size	2	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord
offset	64	(optional) destination of the mov instruction

0x1A - NOT

Pop a value from the stack, perform a bitwise **NOT** operation, i.e., reverses all bits, and push the result back on the stack.

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x1A` or `0x1F`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x1B - AND

Pop two values from the stack, **AND** them and push the result back on the stack.

field	bits	description
has size	1	`0` : no size field, defaults to Dword, `1` : size value follows (1 byte)
opcode	6	must be `0x1B`
size	8	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

0x1C - SHR, SHL …

Pop a value from the stack, **shift** the value, and push the result back on the stack.

field	bits	description
has size	1	`0` : no size and type field, defaults to Dword and *SHL/SHR*, `1` : 6 bit type and 2 bit size value follows
direction	1	direction of shift, `0` : shift right, `1` : shift left
opcode	6	must be `0x1C`
type	6	(optional) type of shift, `0` : SHL or SHR shift, `2` : ROL or ROR, `3` : SAL or SAR; if `has size` is not set, then this field does not exist and the type of shift defaults to SHL and SHR respectively.
size	2	(optional) size of the operation, `0` : Byte, `1` : Word, `2` : DWord, `3` : QWord

---

Source: https://johannesbader.ch/2019/07/the-dga-of-pitou/