Mutation Engine For Fun And Profit

ADHOKSHAJ MISHRA

Agenda

- Legal disclaimer
- Who am I?
- History of mutation
- Basic introduction to code mutation
- Common mutation techniques
- Pros and cons of various techniques
- Writing polymorphic engine
- Writing oligomorphic engine

Agenda

- Writing metamorphic engine
- Compile time mutation: what and why?
- Programming the programming language
- Compile time evaluation
- Obfuscating data
- Obfuscating code
- Detection and analysis of mutation engines

Legal Disclaimer

The opinions expressed in the presentation and on the following slides are solely of mine, and not of my employer.

The technique(s) presented hereafter are offensive in nature; and are generally considered a criminal offence if practiced without proper authorization in place. It is presented here for educational purpose only. In other words, if you come to me saying that you are in trouble due to these techniques, I won’t feel responsible at all.

Who am I?

- Independent security researcher, with deep interest in offensive malware techniques and cryptography.
- I love to attend and speak at various security conferences and meet-ups.
- You can contact me on LinkedIn(AdhokshajMishra), or via e-mail(me@adhokshajmishraonline.in)

History of mutation

- 1948: IBM Selective Sequence Electronic Calculator appears with ability to treat instructions as data, and as a consequence to run self modifying code.
- 1981: Self-modifying code based copy protection appears in some disk-based software targeting IBM PC and Apple II
- 1989: Mark Washburn writes 1260 virus with polymorphic engine. It targeted 16 bit COM files.
- 1991: Dark Avenger Mutation Engine, written by DarkAvenger. It created 900000 mutations by late 1992.

History of mutation

- 1997: GriYo writes Marburg virus targeting EXE and SCR files. It was first 32 bit polymorphic virus.
- 1997: GriYo writes Gollum virus with a very basic polymorphism. First virus targeting DOS and Windows both.
- 1999: GriYo writes CTX virus, which is advanced version of Marburg, targeting Windows NT.
- 2000: GriYo writes Dengue virus which was considered one of the most sophisticated virus of its time.

Why mutation?

- Can be used to make life of malware analysts really difficult
- Can be used as an anti-piracy measure
- Can be used for evasion from certain security systems
- Gives you boasting rights among your hacker peers

Where is mutation used today?

- Products like Themida and VMProtect use mutation to protect intellectual property, and as an anti-piracy countermeasure. These are also seen in malware samples.
- Tools like Obfuscator-LLVM provide mutation at compiler level, which is then used by various companies. (Snapchat acquired whole team behind Obfuscator-LLVM).

Introduction to mutation

- A mutation engine is a computer program that can be used to transform a program into a subsequent version that consists of different code yet operates with the same functionality.
- In context of computer science, mutation generally refers to ability of codes to modify themselves. Mutation may be local(affecting a small part of code) or global (affecting whole code), persistent (patching copy on disk) or non-persistent(monkey patching).
- A mutation engine may work at compile time, link time, post-processing stage or runtime. It may exist as a separate entity, or as a part of target code itself.

Classification of mutation techniques

- Generally it is classified into three classes
  - Polymorphic mutation
  - Oligomorphic mutation
  - Metamorphic mutation

Polymorphic Mutation

- Common stub is used with all variants. Only payload varies.
- Common methods are compression (packers), encryption(crypters), or a combination of them.
- Easiest to program and debug.
- Easiest to get detected due to static stub.

Oligomorphic Mutation

- Stub and payload both vary.
- Commonly, whole code is broken into independent chunks. Now each chunk can be replaced with another chunk with equivalent behavior. Similarly, stub is changed accordingly.
- Common methods are: using different cryptographic and/or compression algorithms in different variants, reordering chunks of code as long as functionality is not affected.
- Easy to program and debug.
- Automated detection is harder than polymorphism

Metamorphic Mutation

- Kind of “oligomorphic mutation on steroid” or “oligomorphic mutation taken too far”
- It works at instruction level. Instruction swapping, register swapping, junk insertion/removal, instruction reordering are some of the common techniques.
- Pretty hard to program. Even harder to debug.
- With sufficient mutation, automatic detection is theoretically impossible.

Writing polymorphic engine

- Simplest case: whole payload is self-sufficient and position independent.
- Simple case: the payload is statically linked ELF binary with position independent code.
- Hard case: the payload is dynamically linked ELF binary.
- Worst case: the payload is dynamically linked ELF binary and utilizes stuff like thread local storage callbacks…

Polymorphic engine: simplest case

- Running the payload from a buffer is trivial
  - Find page boundaries
  - Set memory permission to PROT_READ|PROT_EXEC
  - JMP to first byte (or CALL, if you want).
- Payload can be kept encoded / encrypted / compressed
- Payload can even come from network, or database, or registry (Windows®)

Polymorphic engine: simplest case

- Finding page boundaries

```
long pagesize = sysconf(_SC_PAGESIZE);
unsigned long page_start = (unsigned long)payload &
~(pagesize - 1)
unsigned long page_end = ((unsigned long)(payload +
payload_size) & ~(pagesize -1)) + pagesize;
```

Polymorphic engine: simplest case

- Setting memory permissions

```
int error = mprotect((void*)page_start, page_end -
page_start + 1, PROT_READ|PROT_WRITE|PROT_EXEC);
```

Polymorphic engine: simplest case

- Executing the buffer

```
if (!error)
{
     int (*call_payload)() = (int(*)())payload;
     call_payload();
}
```

Polymorphic engine: simple case

- Parse the ELF header, map the sections, copy stuff to memory, set permissions, and you are pretty much done.
- No need to worry about relocation table.
- No need to worry about symbol resolution.

Polymorphic engine: other cases

- Everything in first point from previous case.
- Parse import table, make dependency graph of all dependencies.
- Sort them topologically.
- Load them one by one in sorted order.
- Relocate them as required.

Polymorphic engine: other cases

- Resolve symbols and patch GOT/PLT
- Deal with TLS callbacks, and other dark corners.
- Pray to your favorite Deity for success.
- Weep in case of failure.

Do I have to put THAT much work?


Pathetic.

Writing own ELF loader to load ELF from buffer

memfd_create

Polymorphic engine: the lazy way

- **memfd**

  memfd_create() creates an anonymous file and returns a file descriptor that refers to it. The file behaves like a regular file, and so can be modified, truncated, memory-mapped, and so on. However, unlike a regular file, it lives in RAM and has a volatile backing storage. Once all references to the file are dropped, it is automatically released.

  - shamelessly stolen from Linux man pages.

Polymorphic engine: the lazy way

- Create file descriptor for memory buffer using memfd_create
- Copy ELF contents using write() system call
- Run ELF from /proc/<pid>/fd/<file descriptor>

Polymorphic engine: disadvantages

- The stub is static.
  - Malware analysts love static things.
- Number of variants of payload is extremely limited.
  - Malware analysts will analyze them all.
- Recovering the payload is easy.
  - Malware analysts will automate the recovery.


ONE SIMPLY DOES NOT
GIVE UP
imgflip.com

Writing oligomorphic engine

- Basically polymorphic code on steroid
- Instead of treating whole payload as one chunk, payload is broken into modules
- Modules are shuffled and swapped while preserving overall behaviour
- Different chunks can be loaded in memory for execution on “need to execute” basis, and can be removed after that

Writing oligomorphic engine

- Simplest case: all chunks are self sufficient position independent codes
- Simple case: chunks are ELF shared objects
- Not-so-simple case: chunks are mix of ELF and self-sufficient position independent codes

Writing oligomorphic engine

- Self sufficient position independent codes can be handled just like we did in polymorphic engine.
- ELF shared objects can be loaded from memfd
- Shared objects can be run in multiple ways.

Writing oligomorphic engine

- Dealing with shared objects
  - Some function can be executed automatically when shared object is loaded.
  - Other functions can be “searched” for, within the object, and can be called by any other code from same process space.
  - Both of the above can be combined together.

Writing oligomorphic engine

- Every chunk will have one or more equivalent implementations.
- Whenever a chuck with certain functionality is needed, one can be selected randomly from pool of other implementations.
- Selection and merging of chunks can be done either at post-build stage, or at runtime.

Writing oligomorphic engine

- In some cases equivalent implementations can be auto-generated.
  - Equivalence in Boolean algebra comes handy
    - For example, sum of two numbers can be re-written as:
      - Sum = (A XOR B) XOR Carry-in
      - Carry-out = A AND B OR Carry-in(A XOR B)
    - Every Boolean expression can be converted to equivalent NAND only, and NOR only expressions.
      - Other implementations can be generated by selectively replacing some expressions with equivalent expressions in terms of NAND and/or NOR.

Writing oligomorphic engine

- In some cases equivalent implementations can be auto-generated.
  - Turing equivalence comes handy.
    - All known Turing complete systems are Turing equivalent.
    - The set of assignment and conditional jump is Turing complete.
      - In other words, anything can be done with sufficiently convoluted lookup table and jumps.
    - Brainfuck is Turing complete. Run some parts of code on Brainfuck interpreter.

Writing oligomorphic engine

- In some cases equivalent implementations can be auto-generated.
  - Turing equivalence comes handy.
    - On x86, MOV and XOR are Turing complete (due to so many addressing modes, and their side effects)
      - Replace some parts of code with equivalent code in terms of MOV and/or XOR

Writing oligomorphic engine

- In some cases equivalent implementations can be auto-generated.
  - Single instruction set computers are magical.
    - Writing emulators for such computers is ridiculously simple.
      - There is only one instruction to run.
    - Most of them can be “programmed” to interpret Brainfuck programs.
      - Compile parts of your program to Brainfuck.
      - Run the compiled program on these magical computers.

Writing oligomorphic engine

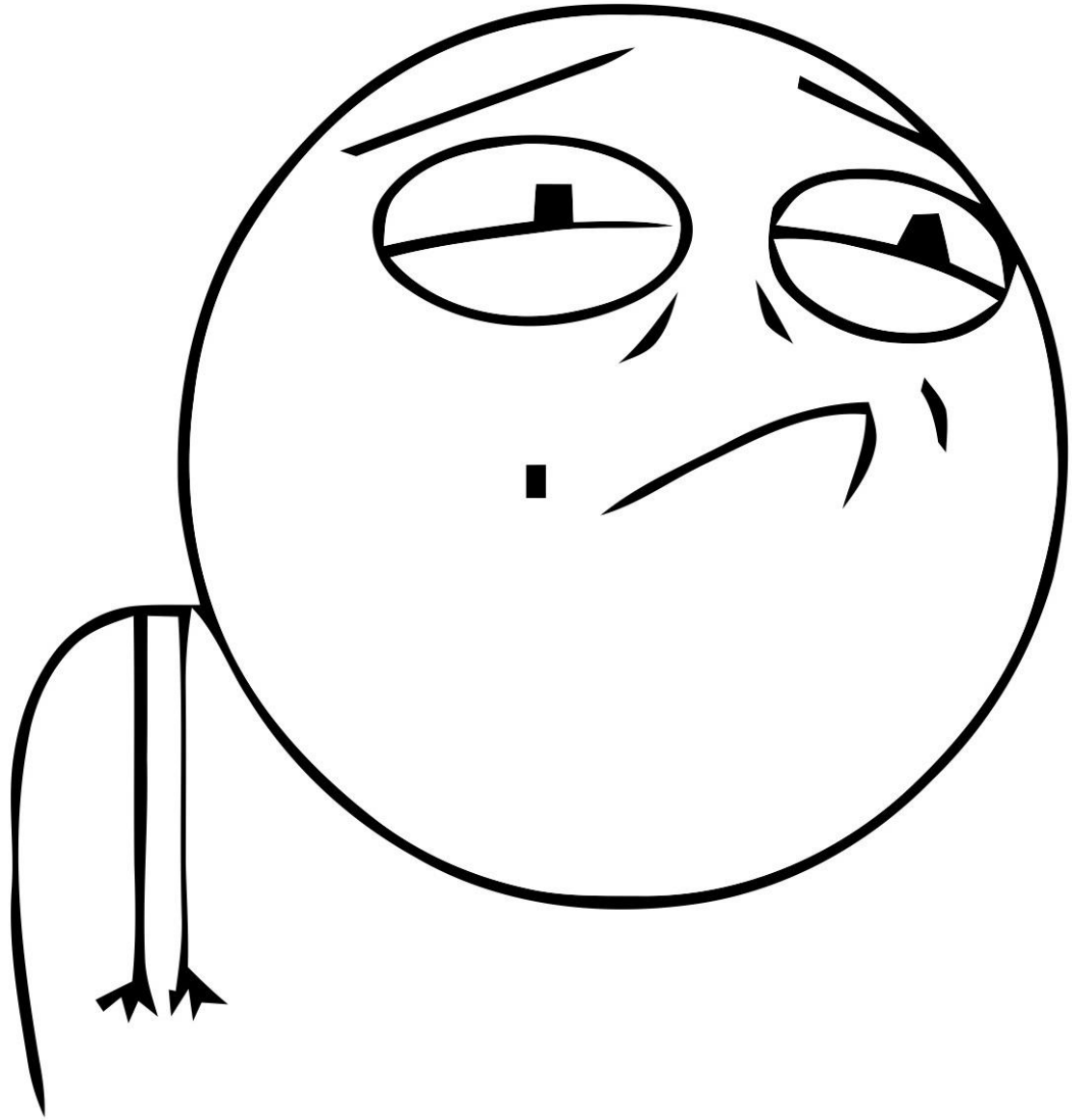
- In some cases equivalent implementations can be auto-generated.
  - Single instruction set computers are magical.
    - TOGA(A, B)
      - Toggle A and branch to B; if result of toggle operation is TRUE.
    - SUBLEQ (A, B, C)
      - Subtract A from B, and jump to C if (A-B) is less than or equal to 0.
    - ADDLEQ(A, B, C)
      - Add A and B, and jump to C if (A+B) is less than or equal to 0.
    - There are many more…

Writing oligomorphic engine

- Tools to try:
  - https://github.com/elikaski/BF-it
    - Compiles C like language to Brainfuck.
  - https://github.com/arthaud/c2bf
    - Compiles C to Brainfuck.

Oligomorphic engine: disadvantages

- Even though stub is not static, there are not too many varieties either.
  - Malware analysts are stubborn people. They will analyze them all.
- Most of the variants are generated using handful of building blocks.
  - Malware analysts will analyze those building blocks, and automate the rest.



Writing metamorphic engine

- Basically oligomorphic code on steroids
- Instead of working at “block of code” level, we deal at instruction level directly
- Opens gate to some more interesting approaches

Writing metamorphic engine: register swapping

- One general purpose register is swapped with another general purpose register.
- Easiest thing to do, and automate.
- Common problems: you have to deal with restrictions imposed by ABI, architecture limitations imposed by underlying system, and limitations due to special uses of certain registers.
  - For example, not all uses of RAX can be swapped out. RSP/RBP, registers to pass arguments to functions or system calls etc. also cannot be swapped out.

Register swapping: example code

```
push   rbp
mov    rbp, rsp
sub    rsp, 16
movabs rax, 8022916924116329800
mov    QWORD PTR -13[rbp], rax
mov    QWORD PTR -5[rbp], 174353522
mov    QWORD PTR -1[rbp], 0
lea    rax, -13[rbp]
mov    edx, 12
mov    rsi, rax
mov    edi, 1
call   write@PLT
mov    eax, 0
leave
ret
```

Register swapping: mutated code

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
movabs rcx, 8022916924116329800
mov     QWORD PTR -13[rbp], rcx
mov     QWORD PTR -5[rbp], 174353522
mov     QWORD PTR -1[rbp], 0
lea     rbx, -13[rbp]
mov     edx, 12
mov     rsi, rbx
mov     edi, 1
call    write@PLT
mov     eax, 0
leave
ret
```

Writing metamorphic engine: instruction re-ordering

- Instructions are written in different order at every run. An instruction that is at one place now, may appear elsewhere in another run.
- Much harder to pull off in automated way, than register swapping.
- Two instructions can be reordered if and only if there is no dependency between them.

Instruction re-ordering: example code

```
push   rbp
mov    rbp, rsp
sub    rsp, 16
movabs rax, 8022916924116329800
mov    QWORD PTR -13[rbp], rax
mov    QWORD PTR -5[rbp], 174353522
mov    QWORD PTR -1[rbp], 0
lea    rax, -13[rbp]
mov    edx, 12
mov    rsi, rax
mov    edi, 1
call   write@PLT
mov    eax, 0
leave
ret
```

Instruction re-ordering: mutated code

```
push   rbp
mov    rbp, rsp
sub    rsp, 16
movabs rax, 8022916924116329800
mov    QWORD PTR -13[rbp], rax
lea    rax, -13[rbp]
mov    QWORD PTR -5[rbp], 174353522
mov    rsi, rax
mov    QWORD PTR -1[rbp], 0
mov    edi, 1
mov    edx, 12
call   write@PLT
mov    eax, 0
leave
ret
```

Writing metamorphic engine: instruction swapping

- Set of one or more instructions is swapped with another set of one or more instructions where both set have exact same output on same input.
- Problem: size and location of code changes, and therefore you may (and will!) have to fix jump offsets, references etc.

Instruction swapping: example code

```
push   rbp
mov    rbp, rsp
sub    rsp, 16
movabs rax, 8022916924116329800
mov    QWORD PTR -13[rbp], rax
mov    QWORD PTR -5[rbp], 174353522
```

```
mov    QWORD PTR -1[rbp], 0
lea    rax, -13[rbp]
mov    edx, 12
mov    rsi, rax
mov    edi, 1
call   write@PLT
mov    eax, 0
leave
ret
```

Instruction swapping: mutated code

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
movabs  rax, 8022916924116329800
mov     QWORD PTR -13[rbp], rax
mov     QWORD PTR -5[rbp], 1744353522
mov     QWORD PTR -1[rbp], 0
lea     rax, -13[rbp]
```

```
mov     edx, 6
sal     edx, 1
mov     rsi, rax
xor     edi, edi
inc     edi
call    write@PLT
xor     eax, eax
leave
ret
```

Writing metamorphic engine: junk insertion

- A set of one or more instructions, which is effectively an elaborate NOP, is inserted at random places in code.
- Much easier to inject
- Problem: detection of junk is a bit hard. Without detection and removal, code size balloons to infinity.

Junk insertion: example code

```
push   rbp
mov    rbp, rsp
sub    rsp, 16
movabs rax, 8022916924116329800
mov    QWORD PTR -13[rbp], rax
mov    QWORD PTR -5[rbp], 174353522
mov    QWORD PTR -1[rbp], 0
lea    rax, -13[rbp]
mov    edx, 12
mov    rsi, rax
mov    edi, 1
call   write@PLT
mov    eax, 0
leave
ret
```

Junk insertion: mutated code

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
xor     rcx, rcx
movabs  rax, 8022916924116329800
push    rcx
mov     QWORD PTR -13[rbp], rax
mov     QWORD PTR -5[rbp], 1743535222
mov     QWORD PTR -1[rbp], 0
inc     rcx
lea     rax, -13[rbp]
mov     edx, 12
mov     rsi, rax
mov     edi, 1
dec     rcx
call    write@PLT
pop     rcx
mov     rax, rcx
leave
ret
```

Writing metamorphic engine: control flow obfuscation

- Straightforward code is broken into chunks, and control jumps between them irregularly.
- Even before one starts automating, control flow graph of existing code needs to be analyzed. Best bet is to obfuscate control flow at function level and move from there.
- Problem: as soon as you cross function boundary, things get ridiculously difficult pretty quickly.

Control Flow Obfuscation: example code

```
push   rbp
mov    rbp, rsp
sub    rsp, 16
movabs rax, 8022916924116329800
mov    QWORD PTR -13[rbp], rax
mov    QWORD PTR -5[rbp], 174353522
mov    QWORD PTR -1[rbp], 0
lea    rax, -13[rbp]
mov    edx, 12
mov    rsi, rax
mov    edi, 1
call   write@PLT
mov    eax, 0
leave
ret
```

Control Flow Obfuscation: mutated code

```
    push   rbp
    mov    rbp, rsp
    jmp l0
14: call   write@PLT
    mov    eax, 0
    jmp l5
l0: sub    rsp, 16
    jmp l1
l2: mov    QWORD PTR -5[rbp], 174353522
    mov    QWORD PTR -1[rbp], 0
    jmp l3
```

```
l5: leave
    ret
l1: movabs rax, 8022916924116329800
    mov    QWORD PTR -13[rbp], rax
    jmp l2
l3: lea    rax, -13[rbp]
    mov    edx, 12
    mov    rsi, rax
    mov    edi, 1
    jmp l4
```

Writing metamorphic engine

- A good metamorphic engine has to employ all the techniques to be effective.
- Metamorphic engines are generally tightly coupled with one architecture.
  - Supporting multiple architectures is **HUGE** pain in some special part of body.
  - Different devices will use different ISA; and therefore, we have to deal with different architectures.

Writing metamorphic engine

- Metamorphic code comes with lots of warts:
  - Junk code analysis is very difficult to get right.
    - Incorrect analysis will lead to some mutations showing erratic behavior.
    - Common approach is to start from same base state, and mutate them by transforming randomly selected chunks of code by randomly selected techniques.

Writing metamorphic engine

- Metamorphic code comes with lots of warts:
  - Control flow analysis, and dependency analysis on optimized assembly is non-trivial.
    - Many instructions have extra side effects
      - For example: **AND** will clear the CF and OF flags, sets PF if the low byte of the result has odd parity, sets SF if the result is negative, and sets ZF if the result is zero.
    - Not everything will be in defined state every time
      - For example: **AND** instruction leaves the AF flag undefined.
    - Assembly is full of surprises
      - For example: **NOT** instruction does not set flags!

Writing metamorphic engine

- Metamorphic code comes with lots of warts:
  - We cannot apply any transformation on any function either…
    - If some function is not exported, it can be transformed into another function which takes parameters in non-standard way; and all call sites for said function can be modified.
    - All of this goes right out of window if function is exported. If we don’t follow the calling convention, said function cannot be called from outside.


I DON'T WANNA!
IT'S TOO HARD!
memegenerator.net

Write metamorphic engine by parsing assembly output of compiler

Write a compiler to generate different mutated code every time

Force your compiler to mutate code without having to patch it

Solution: Chuck Norris Mode

Why compile time mutation?

- There is no need to increase one more step in build (mutation from specific tool)
- There is no need to embed entire mutation engine in target code (mutation is as easy as recompiling everything)
- There is no need to emit assembly directly. We can leave that to compiler, and focus on mutation itself.

Why compile time mutation?

- Good part: we can leave fancy analysis of control flow and data dependency graphs to compiler.
- Better part: we can either limit to oligomorphic mutation, or go up to metamorphic mutation.
- Best part: we don’t have to work as hard to support various architectures.
  - In fact, we have to do very little work on that front. Most of the heavy lifting is done by compiler.

Compile-time mutation engine

A compile-time mutation engine is a program written in programmable language, where you program the programming language itself (in an embedded domain-specific language which just happens to be Turing-complete) to write a program which re-programs parts of anything it attaches to.

In easy terms, a program-ception.

Compile time mutation

- Tools of trade
  - Metaprogramming
    - Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running.
    - In simple words, metaprogramming refers to a variety of ways a program has knowledge of itself or can manipulate itself.

Compile time mutation

- Tools of trade
  - Metaprogramming
    - In languages like C#, reflection is a form of metaprogramming since the program can examine information about itself. For example returning a list of all the properties of an object.
    - In languages like ActionScript, you can evaluate functions at runtime to create new programs such as eval("x" + i). DoSomething() would affect an object called x1 when i is 1 and x2 when i is 2.

Compile time mutation

- Tools of trade
  - Metaprogramming
    - Finally, another common form of metaprogramming is when the program can change itself in non-trivial fashions.
      - LISP is a famous example of such functionality.
      - Pre-processor macros in C are a very weak version of this.
      - C++ templates provide pretty strong metaprogramming capability (not at level of LISP though)

Compile time mutation

- Tools of trade
  - C++ templates
    - Provide a way to generate code at compile time, when generated codes are going to be similar (but not same).
    - Both functions and classes can be templated
    - Are Turing complete!
    - Some templates can be evaluated completely at compile time
      - Others will generate code that will be executed at run time

Compile time mutation

- Tools of trade
  - `constexpr`
    - This specifier declares that it is possible to evaluate the value of the function or variable at compile time. Such variables and functions can then be used where only compile time constant expressions are allowed.

Compile time mutation

- Tools of trade
  - Standard Macros
    - __TIME__ expands to current time.
    - __DATE__ expands to current date.
  - Compile time pseudo-random number generator
    - We have to write one.

Compile time PRNG

- Tools of trade
  - constexpr to perform compile-time evaluations without templates
  - C++ templates to accept arbitrary input and return random number
  - __TIME__ as seed. It is only thing that we can rely upon to be unique on each compilation.
  - Park-Miller PRNG. It is easy enough to implement, and does not produce too bad output.

Compile time PRNG

Generating seed at compile time:

```
constexpr char time[] = __TIME__;
constexpr int DigitToInt(char c) { return c - '0'; }
const int seed = DigitToInt(time[7]) +
DigitToInt(time[6]) * 10 + DigitToInt(time[4]) * 60
+ DigitToInt(time[3]) * 600 + DigitToInt(time[1]) *
3600 + DigitToInt(time[0]) * 36000;
```

Compile time PRNG

- Generating pseudo-random numbers:
  - Setting up base case:

```
template<> struct RandomNumberGenerator<0>
{
        static constexpr unsigned value = seed;
};
```

Park-Miller PRNG at Compile Time

```
template<int N> struct
RandomNumberGenerator {

private:

static constexpr unsigned a =
16807;

static constexpr unsigned m =
2147483647;

static constexpr unsigned s =
RandomNumberGenerator<N -

1>::value;

static constexpr unsigned lo = a *
(s & 0xFFFF);

static constexpr unsigned hi = a *
(s >> 16);

static constexpr unsigned lo2 = lo
+ ((hi & 0x7FFF) <<

16);

static constexpr unsigned hi2 = hi
>> 15;

static constexpr unsigned lo3 = lo2
+ hi;

public:

static constexpr unsigned max = m;

static constexpr unsigned value =
lo3 > m ? lo3 - m :

lo3; };
```

Park-Miller PRNG at Compile Time

Generating random number within a range:

```
template<int N, int M> struct RandomNumber
{
	static const int value = RandomNumberGenerator<N + 1>::value % M;
};
```

Code Obfuscation

- Tools of trade
  - Templated classes and functions
    - Forward declaration
    - Partial initialization
  - Operator overloading
  - Inline assembly (optional)
  - Compiler flags to disable certain optimization passes on certain areas of code (optional)

Code Obfuscation

- Example case: adding two numbers
  - Define a wrapper class MyInt<int N, typename T>
    - Value of N will determine which version of code will get triggered
    - Type of T will determine actual value type (the same class can handle various integer types in one template)

Base case:

```
template<int N, typename T> class MyInt;
```

Code Obfuscation

Basic implementation:

```
template<int N, typename T> class MyInt
{
private: T value;
public: MyInt<N,T> operator+(MyInt param) {…}
…
}
```

Code Obfuscation

- Actual code will contain various partial initialized version of MyInt.

```
template<typename T> class MyInt<0, T> { … }
template<typename T> class MyInt<1, T> { … }
template<typename T> class MyInt<2, T> { … }
```

- Different partially initialized versions will implement various operators differently.
  - Pro-tip 1: automate this step.
  - Pro-tip 2: Generate assembly directly. Use techniques from previous slides.

Code Obfuscation

- Different partially initialized versions will implement various operators differently.
  - Pro-tip 1: automate this step.
  - Pro-tip 2: Generate assembly directly. Use techniques from previous slides.
  - Pro-tip 3: Don’t be ashamed in using inline assembly. Or maybe drop a call, and link with generated code using a linker.

Code Obfuscation

- At time of use, set different values for N using compile time random number generator.
  - Don’t forget the range. Compiler will cry if partial initialized version of template does not exist for value of N at hand.
- If you are feeling adventurous, you can implement automation pro-tip from previous slides entirely within templates.
  - Remember, templates in C++ are **Turing Complete**.

Code Obfuscation

- Functions too can be templated just like data types.
  - Again, you can automate some code generation directly in assembly.
  - Or, you can use LLVM and play with IR
  - Or, you can use LLVM, and generate extra code directly in C (or C++)

Data Obfuscation

- Tools of trade
  - Compile time indexes (for strings)
  - Templated classes and functions
  - Inline assembly (optional)
  - Compiler flags to disable certain optimization passes on certain areas of code (optional)

Data Obfuscation

- Example case: strings
  - Define a wrapper class MyString<int N, char Key, typename Indexes>
    - Value of N controls which version of obfuscation will be used
    - Key is randomly chosen encryption key
    - Indexes are where encryption is to be applied
  - Define a wrapper class Indexes<int… I>
    - This is a variadic template, as we don’t know how many indices we will need

Data Obfuscation

- Example case: strings

Base case:

```
template<int N, char Key, typename Indexes>
struct MyString;
```

Data Obfuscation

Basic Implementation:

```
template<int N, char Key, typename Indexes>
struct MyString {

public:

constexpr __attribute__((always_inline))

MyString(const char* str) : buffer_ {K,

encrypt(str[I])...} { }

inline const char* decrypt() { ... }
```

Data Obfuscation

Basic Implementation:

```
private:
constexpr char key() const { return buffer_[0];
}
constexpr char encrypt(char c) const { ... }
constexpr char decrypt(char c) const { ... }
char buffer_[sizeof...(I) + 2];
}
```

Data Obfuscation

Generating indices:

```
template<int... I> struct Indexes { using type =
Indexes<I..., sizeof...(I)>; };
template<int N> struct Make_Indexes { using type =
typename Make_Indexes<N-1>::type::type; };
template<> struct Make_Indexes<0> { using type =
Indexes<>; };
```

Data Obfuscation

Now indices can be generated trivially like this :

```
Make_Indexes<sizeof(str) - 1>::type
```

Data Obfuscation

- Different encrypt() and decrypt() functions can be chosen trivially
  - Implement partially initialized copies of MyString meta-class, for different values of N
  - Each implementation has its own overloaded version of encrypt() and decrypt()
- Finally, wrap the invocation in macro to make life easier (optional step)

Data Obfuscation: limitations

- Only literal values can be obfuscated at compile time. The good news is, any data type can be obfuscated just like strings.
- It does not hide the de-obfuscation mechanism, which means any half-decent reverse engineer can get the actual data easily.
  - Unless you apply mutation on decryption routines.
    - Make the rabbit hole as deep as you want.
      - As long as performance penalty is not absurdly high…


maal kidhar hai ?

Malware to Maal Where?

COUNTERMEASURES

Countermeasures

- Mutation only saves you from static detection.
  - Pro tip: monitor behavior.
- Process launched from memfd will look different from other processes
  - Path will be **/memfd:<some text> (deleted)**
- Linux Audit System is your friend. Use it
  - Or use whatever equivalent your OS provides

Countermeasures

- Use event based monitoring.
  - Osquery rocks here
- Static stubs can be identified by looking for known artifacts in process memory
  - YARA will do it for you
  - Osquery supports YARA
    - And, Osquery will get you event when Yara flags something.

Countermeasures

- Such processes are rather short lived. Keep past event data to correlate in future.
  - By the time you will look into osquery event table, the process will be long gone.

Countermeasures

- Deciphering mutations
  - If you know the tactics used for mutation, you can use them in reverse
  - You can use some solvers to possibly minimize code
    - Or to find various equivalent chunks in the code
  - Programs for single instruction computers can be partially recovered by using solvers

Countermeasures

- Tools of trade
  - Yara: https://github.com/VirusTotal/yara
  - Osquery: https://osquery.io/
  - Linux Audit System: at bare minimum, configure auditd, and process logs