полиморфный генератор — своим руками

крис касперски ака мыщъх, no e-mail

технологии полиморфизма можно встретить не только в вирусах, но и, например, в защитных механизмах. это передовой путь хакерской мысли, притягивающий новичков как магнитом, это настоящий омут ассемблерных конструкций, в который легко попасть, но трудно выбраться. полиморфные вирусы достаточно сложная штука, но настоящие хакеры прямых путей не ищут!

введение

Существуют тысячи готовых полиморфных движков, вобравших в себя множество блестящих идей и чтобы переплюнуть их придется очень сильно постараться. Конечно, профессиональный хакер может (и должен!) бросить вызов своим коллегам, удавить мир очередным шедевром (или сначала удивить, а потом удавить). Но для этого требуется опыт. А где его взять? Правильно, начать программировать простенькие полиморфные генераторы, повторяя давно пройденный путь. И вот так маленькими шажками, хакерское сообщество движется путем прогресса из золотого века во тьму.

классификация полиморфных генераторов

Принято выделять шесть ступеней полиморфизма — от простейших "одноклеточных", до развитых организмов. Будем следовать этой классификации и мы. Рассмотрим как она выглядит с точки зрения хакера и антивируса.

ступень 0: permutating или простейшие перестановки

Идея: обрабатываемый код делится на блоки постоянного или переменного размера, которые в каждом поколении вируса переставляются в случайном порядке. Это еще не настоящая полиморфия, но и обычным такой код уже не назовешь. Он легко программируется, но и легко обнаруживается, ведь содержимое блоков остается неизменным, поэтому с ними справляется даже сигнатурный поиск.

Поскольку блоки "нарезаются" еще на стадии проектирования, проблемы случайного "расщепления" машинных команд границами блоков не возникает и сочинять собственный дизассемблер длин нам не нужно. Тем не менее, при программировании возникают следующие проблемы: поскольку, адреса блоков в каждом поколении меняются, машинный код должен быть полностью перемещаемым, то есть сохранять работоспособность независимо от своего местоположения. Это достигается путем отказа от непосредственных межблочных вызовов. Совершать переходы, вызывать функции, обращаться к переменным можно только в пределах "своего" блока. В практическом плане это значит, что вместе с кодом каждый блок несет и свои переменные.

Но все-таки делать межблочные вызовы иногда приходится. Как? Зависит от фантазии. Проще всего создать таблицу с базовыми адресами всех блоков и разместить ее по фиксированному смещению, например, положить в первый блок. Она может выглядеть, например, так:

```
base_table:
        block_1 DD      offset block_1
        block_2 DD      offset block_2
        ...
        block_N DD      offset block_N
```

Листинг 1 таблица косвенных вызовов с базовыми адресами всех блоков

А вызов блока может выглядеть так:

```
        SHR     ESI,2                           ; умножаем номер блока на 4
        ADD     ESI, offset base_table + 4      ; переводим в смещение (4 понадобилось
                                                ; затем, что блоки нумеруются с 1)
        LODSD                                   ; считываем адрес блока
        ADD     EAX,EBX                         ; добавляем смещение функции
        CALL    EAX                             ; вызываем блок
```

Листинг 2 косвенный межблочный вызов, номер блока передается в регистре ESI, а смещение функции от начала блока — в регистре EBX, аргументы функции можно передавать через стек

Как вариант, можно разместить перед функцией ASCIIZ-строку с ее именем (например, "my_func"), а затем осуществлять его хэш-поиск, что позволяет обстрагивается от смещений, а, значит, упростить кодирование, но в этом случае содержимое всех блоков должно быть зашифровано, чтобы текстовые строки не сразу бросались в глаза. Впрочем, шифруй - не шифруй, антивирус все равно сможет нас обнаружить, а обнаружив — поиметь. Или отыметь? А! Не важно!

Процедуру опознания можно существенно затруднить, если сократить размер блоков до нескольких машинных команд, "размазав" их по телу файла-жертвы. Внедряться лучше всего в пустые места (например, последовательности нулей или команд NOP /* 90h */ образующиеся при выравнивании). В противном случае нам придется где-то сохранять оригинальное содержимое файла, а затем восстанавливать его, а это геморрой.

Нарезка блоков может происходить как статически на стадии разработки вируса, так и динамически — в процессе его внедрения, но тогда нам потребуется дизассемблер длин, сложность реализации которого намного превышает "технологичность" всего пермутирующего движка. Так что здесь он будет смотреться как золотая цепь на шее у бомжа. Ладно, прекратим отвлекаться на бомжей и рассмотрим общую стратегию внедрения.

Вирус сканирует файл на предмет поиска более или менее длинной последовательности команд NOP или цепочек нулей, записывает в них кусочек своего тела и добавляет команду CALL для перехода на следующий фрагмент. Так продолжается до тех пор, пока вирус полностью не окажется в файле.

Различные программы содержат различное количество свободного места, расходующегося на выравнивание. В программы, откомпилированные с выравниванием на величину 4'х байт втиснутся практически нереально (поскольку даже команда перехода, не говоря уже о команде CALL, занимает по меньшей мере два байта). С программами, откомпилированными на величину выравнивания от 08h до 10h байт, все намного проще и они вполне пригодны для внедрения.

Ниже в качестве примера приведен фрагмент одного из таких вирусов

```
.text:08000BD9         xor     eax, eax
.text:08000BDB         xor     ebx, ebx
.text:08000BDD         call loc_8000C01
…
.text:08000C01 loc_8000C01:                            ; CODE XREF: .text:0800BDD↑j
.text:08000C01         mov     ebx, esp
.text:08000C03         mov     eax, 90h
.text:08000C08         int     80h                     ; LINUX - sys_msync
.text:08000C0A         add     esp, 18h
.text:08000C0D         call loc_8000D18
…
.text:08000D18 loc_8000D18:                            ; CODE XREF: .text:08000C0D↑j
.text:08000D18         dec     eax
.text:08000D19         call    short loc_8000D53
.text:08000D1B         call    short loc_8000D2B
…
.text:08000D53 loc_8000D53:                            ; CODE XREF: .text:08000D19↑j
.text:08000D53         inc     eax
.text:08000D54         mov     [ebp+8000466h], eax
.text:08000D5A         mov     edx, eax
.text:08000D5C         call    short loc_8000D6C
```

Листинг 3 фрагмент файла, зараженного пермутирующим вирусом "размазывающим" себя по кодовой секции

Естественно, фрагменты вируса не обязательно должны следовать линейно друг за другом. Напротив, если только создатель вируса не даун, CALL'ы будут блохой скакать по всему файлу, используя "левые" эпилоги и прологи для слияния с окружающими функциями.

В машинном представлении CALL target является относительным адресом. Как правильно вычислить относительный адрес перехода? Определяем смещение команды перехода от физического начала секции, добавляем к нему три или пять байт (в зависимости от длины команды). Полученную величину складываем в виртуальным адресом секции и кладем полученный результат в переменную a1. Затем определяем смещение следующей цепочки, отсчитываемое от начала той секции, к которой она принадлежит и складываем его с

виртуальным адресом, записывая полученный результат в переменную a2. Разность a2 и a1 и представляет собой операнд инструкции CALL.

Теперь необходимо как-то запомнить начальные адреса, длины и исходное содержимое всех цепочек. Если этого не сделать, тогда вирус не сможет извлечь свое тело из файла для внедрения в остальные файлы. Вот поэтому-то для перехода между блоками мы использовали команду CALL, а не JMP! При каждом переходе на стек забрасывается адрес возврата, представляющий собой смещение конца текущего блока. Как нетрудно сообразить, совокупность адресов возврата представляет собой локализацию "хвостов" всех используемых цепочек, а адреса "голов" хранятся… в операнде команды CALL! Извлекаем очередной адрес возврата, уменьшаем его на четыре и – относительный стартовый адрес следующей цепочки перед нами! Так что "сборка" вирусного тела не будет большой проблемой!

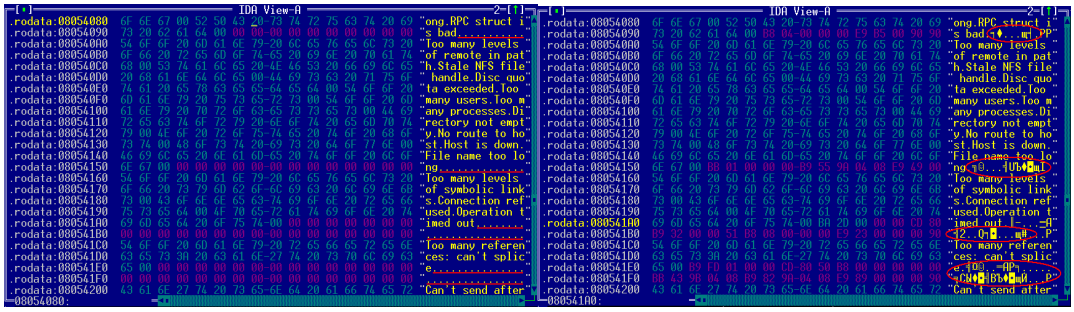

Рисунок 1 так выглядел файл cat до (слева) и после (справа) его заражения перкутирующим вирусом

ступень 1 выбор случайного расшифровщика из списка

Идея: создаем некоторое количество шифровщиков/расшифровщиков, выбираем случайный шифровщик и зашифровываем тело вируса вместе с остальными шифровщиками/расшифровщиками. В каждом последующем поклонении расшифровщик расшифровывает вирус, передает управление основному телу, а перед внедрением выбирает случайный шифровщик, которым и зашифровывает его. Код вируса меняется на 100% и "визуально" опознать зараженный файл уже не представляется возможным.

Правда, поскольку количество расшифровщиков всегда конечно (даже если оно очень-очень велико), сигнатурный поиск остается достаточно эффективной мерой противодействия. Антивирус просто заносит в базу "фотороботы" всех расшифровщиков и… хана вирусу. Правда, тут есть одно небольшое "но". Расшифровщики различных вирусов (и даже честных программ с навесной защитой) обычно похожи как две капли воды и потому в их детектировании нет никакой пользы. Антивирус вынужден привлекать эмулятор, имитирующий выполнение расшифровщика и раскриптовывающий основной вирусный код. Если распаковщик содержит антиотладочные команды или использует машинные инструкции, не поддерживаемые эмулятором, антивирус обломается по полной программе. Здесь, правда, появляется проблема типа "грабли", на которые наступают те, кто борется с эмулятором. Чем больше антиотладочных приемов содержит расшифровщик, тем выше его уникальность и, следовательно, надежнее детектирование. Строго говоря, первая ступень — это еще не полиморфизм. Такие вирусы называют псевдо-полиморфными или олигоморфиками, но все-таки это уже большой шаг вперед.

Какие проблемы возникают при кодировании? Во-первых, код расшифровщика должен быть полностью перемещаемым. Тут базара нет! Написать такой расшифровщик несложно. Во-вторых, расшифровщик должен как-то определять начало и конец зашифрованного фрагмента. Это тоже решаемо. Шифровка не изменяет длину данных, и размер шифроблока будет постоянен для всех расшифровщиков, так что его можно вычислить еще на стадии ассемблирования. В-третьих, для каждого шифровщика должен существовать парный расшифровщик или же выбранный криптоаглгоритм должен быть симметричен, т. е. повторная шифровка зашифрованного текста расшифровывает его (не путать с симметричной криптографией — это совсем из другой области!)

Свойством "симметрии" обладают операции NOT; XOR X, любая константа; ROL/ROR X, 4 и некоторые другие арифметическо-логические операции, например, ADD byte,100h/2.

Простейший симметричный шифровщик может выглядеть так (предварительно необходимо открыть сегмент кода на запись, что можно сделать функцией VirtualProtect):

```
        MOV     ESI, offset body_begin
        MOV     EDI, ESI
        MOV     ECX, offset body_end - body_begin
my_begin:
        LODSB
        XOR     AL, 66h
        STOSB
LOOP my_begin
body_begin:

; // тело вируса со всеми остальными шифровщиками

body_end:
```

Листинг 4 симметричный расшифровщик на основе XOR

А вот другой расшифровщик:

```
        LEA EAX, body_end
my_begin:
        NOT     byte ptr DS:[EAX]
        SUB     EAX, offset body_begin +1
        PUSHF
        ADD     EAX, offset body_begin
        POPF
        JNZ my_begin
body_begin:

; // тело вируса со всеми остальными шифровщиками

body_end:
```

Листинг 5 симметричный расшифровщик на основе NOT

Несимметричные шифровщики/расшифровщики устроены сложнее, зато здесь наблюдается гораздо большее разнообразие. Можно использовать практически любую комбинацию арифметическо-логических команд, только никакого смысла в этом нет. Все равно, если антивирус доберется до файла, он его схватает.

ступень 2: расшифровщик из кирпичиков

Идея: расшифровщик вируса имеет постоянный алгоритм, но состоит из произвольно выбираемой последовательности команд. Звучит страшновато, но реализуется тривиально. Возьмем в качестве наглядно-агитационного пособия расшифровщик, изображенный на листинге 4. А теперь попробуем подобрать синонимы для каждой из слагающих его машинных команд (заботится об оптимизации необязательно). Получится примерно так:

```
MOV ESI,offset body_begin → LEA ESI,body_begin;MOV ESI,offset body_begin-1/INC ESI;
MOV EDI, ESI → PUSH ESI/POP EDI; XOR EDI,EDI/ADD EDI,ESI;
LODSB → MOV AL,[ESI]/INC ESI; SUB AL,AL/SUB AL,[ESI]/NOT AL/INC ESI/ADD AL,1
XOR AL,66h →  XOR AL, 6/XOR AL, 60h
…
```

Листинг 6 инструкции и их синонимы

Для каждой из команд шифровщика мы будем выбирать случайную последовательность "синонимов", как бы собирая шифровщик из кирпичиков. Результат нашей работы может выглядеть, например, так:

```
        LEA ESI, body_begin
        PUSH ESI
        POP EDI
        MOV ECX, offset body_end - body_begin + 2
        DEC ECX
        DEC ECX
my_begin:
        MOV AL,[ESI]
        INC ESI
        XOR AL, 6
```

```
        XOR AL, 66
        MOV [EDI], AL
        SUB EDI,-1
        ADD ECX,-1
        JNZ my_begin
```

Листинг 7 случайно сгенерированный расшифровщик

Поскольку, "синонимы" команд подготавливаются вручную еще на этапе ассемблирования, автоматически определять их длины совсем необязательно! Реализация получается простая как два пальца. Единственную проблему представляют метки. Как видно, и абсолютное, и относительное смещение my_begin изменилось. Мы не можем, не имеет права на этапе ассемблирования подставлять в команду JNZ смещение my_begin, поскольку оно будет указывать в космос. Что делать? Вот одно из решений проблемы. Вместо метки подставляем команду сохранения текущего EIP в один из свободных регистров общего назначения, а затем прыгаем на него:

```
my_begin:
        CALL $+5        ; прыгаем на следующую команду, занося EIP в стек
        POP EBX         ; выталкиваем EIP из стека
        PUSH EBX        ; (как вариант, можно дать INC EBX)
        ...
        JNZ $+4         ; выход из расшифровщика
        JMP EBX         ; переход на my_begin с динамич
```

Листинг 8 автоматическое определение смещений меток

На первый взгляд, каждое последующее поколение вируса выглядит полностью непохожим на предыдущее и по сигнатурам он уже не детектируется. Ведь если подобрать побольше команд-синонимов, количество возможных комбинаций может составить не один миллион. Тресни моя антивирусная база! Тем не менее, в каждой позиции расшифровщика встречается строго определенная комбинаций команд, поэтому специальный алгоритм отождествления сможет надежно ее распознать. Естественно, для этого антивирусникам придется потрудиться и запастись хорошей травой. Был как-то у одной такой компании мешок отменной травы, так за месяц скурили! Ну это ладно.

Не стоит забывать и про эмулятор. Расшифровав основной код вируса, разработчики запросто отождествят его по сигнатуре. Поэтому, антиотладочные приемы должны быть! Здесь за уникальность можно уже не опасаться (команды-синонимы делают свое), оттянувшись на полную катушку со всем набором SSE и прочих векторных команд, до которых антивирусным эмуляторам еще ползти и ползти…

ступень 3: самый мусорный

Идея: внедряем в расшифровщик мусорные команды, не имеющие никакого побочного эффекта, но преображающие код до неузнаваемости. Чаще всего используются различные вариации NOP, которых насчитываются десятки (например, XCHG EBX, EBX/MOV ECX,ECX/Jx $+2/XCHG EAX, EBX; XCHG EBX, EAX и т.д. — главное фантазию иметь).

Обработанный расшифровщик из листинга 4 после "экзекуции" может выглядеть, например, так:

```
        XCHG    ECX, ECX
        MOV     ESI, offset body_begin
        JO      $+2
        MOV     EDI, ESI
        MOV     EAX, EAX
        MOV     ECX, offset body_end - body_begin
        NOP
my_begin:
        LODSB
        JNZ     $+2
        XOR     AL, 66h
        MOV     EDI,EDI
        STOSB
        JZ      $+2
LOOP my_begin
```

Листинг 9 расшифровщик, разбавленный ничего не значащими мусорными командами

Какие проблемы возникают при кодировании? Проблем много. Рассмотрим лишь две из них, как самые важные. Если не предпринять никаких усилий, уже через несколько поколений размер вируса вырастет до облаков и продолжит расти до тех пор, пока не упрется в конец адресного пространства (или отведенной приложению памяти). Чтобы этого избежать, необходимо либо вычищать имеющийся мусор перед добавлением новой порции (что сложно), либо включить в зашифрованное тело копию расшифровщика и всегда "издеваться" только над ней.

А вот вторая проблема. Чтобы добавить мусорную инструкцию между двумя другими, необходимо как-то определить где кончается одна и начинается другая. Вот для этого нам и нужен дизассемблер длин. Дизассемблер длин представляет собой до предела упрощенный x86 дизассемблер, декодирующий инструкции и определяющий их границы. Это достаточно сложная задача, которая не имеет простых и красивых решений. Приходится зарываться в формат кодирования машинных команд, а это целый исторический пласт с кучей наслоений. Когда-то мы дойдем и до него, но нельзя ли пока обойтись методом грубой силы? Можно! Некоторые вирусы определяют границы команд с помощью трассировки, так сказать, руками самого процессора. Но это все равно утомительно. Лучше (и быстрее!) определить границы вручную и занести в специальную таблицу. Уродливо, зато эффективно.

Как антивирусы справляются с такими файлами? Да очень просто! Вычищают весь мусор и натягивают на отстоявшийся "осадок" старую добрую сигнатуру. Мусорные команды легко генерировать, но легко и распознавать! Так что данная методика крайне ненадежна и годится разве что для тренировки.

ступень 4: еще более мусорный

Идея: усложнить генерацию мусорных инструкций, сделав ее менее очевидной. Например, если мы видим: MOV EAX,EBX, то перед этим в EAX можно писать все, что угодно. Все равно он будет заново проинициализирован.

Более сложная задача: отслеживать обращения к регистрам, выявлять неиспользуемые регистры (как локально, так и глобально), и пихать в них всякую глупость. Для этого нужен не только дизассемблер длин, но и полноценный дизассемблер команд, определяющий, что это именно MOV EAX,EBX, а не что-то другое. Причем, необходимо специальным образом обрабатывать флаги — встретив команду, зависимую от флагов (например, Jxx), необходимо найти ближайшую к ней инструкцию, воздействующую на этот флаг и между ними вставлять только тот "мусор", который не оказывает на флаги никакого влияния. Разумеется, это сложно. Очень сложно. Зато такой эффект!

```
MOV     ESI, EAX
SUB     ESI, ECX
XCHG    EDX, EBP
LEA     EBX, [ESI+EDI]
ROR     AL,8
MOV     ESI, offset body_begin
XOR     ECX, ECX
SUB     EDI, EAX
ADD     EBP, ESI
NOT     EDI
MOV     EDI, ESI
MOV     ECX, offset body_end - body_begin
CALL    $+5
SUB     EAX, ECX
NOT     EBP
POP     EBX
XOR     EAX, EAX
PUSH    EBX
SUB     EBP,ECX
LODSB
MOV     EAX, EAX
ADD     EDX, ESI
XCHG    EBX, EBX
XOR     AL, 66h
XOR     EDX, EDX
ADD     EAX, EDX
STOSB
MOV     EBP, EAX
DEC     ECX
XCHG    EBX,EBX
MOV     EBP,ESI
```

```
        JNZ     $+4
        JMP     EBX
body_begin:

; // тело вируса со всеми остальными шифровщиками

body_end:
```

Листинг 10 расшифровщик, замусоренный значащими командами

Антивирусу, чтобы распознать этот код потребуется реализовать сложную систему графов, анализирующих зависимости по данным и отбрасывающих инструкции, замыкающие граф. Например, MOV EAX, EBX --> ADD EAX, ECX –> MOV EAX, ECX. Последняя команда перекрывает результат деятельности первых двух, выдавая их галимую мусорную природу. Реализовать полиморфный генератор четвертого уровня намного проще чем разработать "лекарство", так что антивирусники тут находятся в проигрыше.

Кстати говоря, без дизассемблера можно в принципе и обойтись, воспользовавшись псевдокодом. В этом случае код шифровщика будет выглядеть так:

```
MOV $R1, offset body_begin
MOV $R3, offset body_end - body_begin
MOV $R3, $IP
XOR [$R1], 66h
ADD $R1, 1
SUB $R2, 1
JNZ $R3
```

Листинг 11 псеводокод простейшего шифровщика

Тогда наша задача сведется к написанию кодогенератора для x86, что намного проще. Наш псведокод может быть построен по самым демократичным принципам и иметь фиксированные длины инструкций. В псевдокоде допустимо напрямую адресовать регистр EIP, переложив заботу на кодогенератор, который, кстати говоря, можно выдрать из любого Open Source компилятора. Это тем более замечательно, что кодогенератор имеет множество различных опций, порождающих различный код (например, код для 8088 и 386 процессоров). Антивирусы идут в глухой отруб!

ступень 5: дальше только небо

Пятый уровень — это наивысший уровень полиморфизма. Он комбинирует методы всех нижележащих уровней, добавляя к ним множество новых фич. Во-первых, это динамическая генерация шифровщика/расшифровщика. Вместо того, чтобы модифицировать фиксированный набор заранее подготовленных расшифровщиков, вирус их создает самостоятельно по случайному закону. Это очень сложная задача, требующая боевого опыта и длительной математической подготовки.

Полиморфный генератор решает с каким типом данных он будет работать, в каком направлении будет происходить шифровка (от начала к концу или от конца к началу), сколько делать проходов, какие арифметические или логические операции выбрать и т. д. Прямая трансляция в x86 код используется редко. Обычно вирус генерирует промежуточный псведокод, структура которого заточена под нужды данной задачи, а потом уже переводит его в "родные" машинные команды.

Более продвинутые полиморфики вообще отказываются от расшифровщика и модифицируют непосредственно само вирусное тело. Это архисложная задача, которую в полном виде до сих пор никто не решил. Усилий тут требуется просто море. Зато какой куш нас ждет! Адекватных методик детектирования подобных вирусов до сих пор не разработано и вряд ли они появятся в дальнейшем. Программистам здесь делать пока нечего. Сначала тут должны поработать математики. Необходимо научить компьютер распознавать "смысл" совокупности машинных команд, а это уже из области AI, работы над котором были свернуты еще в конце семидесятых. Автоматическая декомпиляция невозможна! А вот компиляция — очень даже вполне! Так что появление очередных полиморфных монстров уже не за горами.

заключение или прокалывая небо

Вот уже статья подошла к своему концу, который как ни оттягивай, а он все равно наступает, а готовых примеров полиморфных генераторов в ней не наблюдалось. Почему? Да потому, что даже самый простейший пседвополиморфный вирус в статью просто не влезет. Даже если сосредоточиться на ключевых фрагментах генератора, нам потребуется стопка листов формата A1. Лучше взять готовый исходник (благо недостатка в них не наблюдается, сходите хотя бы на vx.netlux.org и скачайте все вирусные журналы) и проанализировать его. А эта статья поможет понять назначение отдельных вирусных частей, определив степень его полиморфизма и выбранный алгоритм.

Что ж! Как говориться, дорогу осилит идущий. Кстати говоря, что-то до сих пор не слышно про вирусы под платформу x86-64 (она же AMD-64). Неправильно это! В следующей статье мы покажем какие преимущества дает переход на новые процессоры от AMD и как начать писать ассемблерные программы под нее. Как-никак, это последнее радикальное изменение архитектуры со времен 386, у которого есть неплохие шансы потеснить Intel и занять свое место под солнцем. Немного забегая вперед скажем, что иметь для этого 64-битный процессор хоть и желательно, но необязательно. На первых порах можно воспользоваться и эмулятором, а потом уже решать — нужна ли нам эта технология или нет.