

ⓘ Этот общий диалог создан искусственным интеллектом, предназначен только для ознакомления.

Пройдем по алгоритмам под комментариями.

Отлично! Пройдемся по всем алгоритмам, которые законментированы в коде DYE. Это самая ценная часть — здесь описана **логика работы движка** на псевдокоде.

1. Алгоритм TcPush (сохранение вызова в CMS)

Комментарий в коде:

```
asm

comment '
M TCM{}
P Ip
T TST
A Nt-args
I Nt-id
i TCM-index

i = TC.i
Next:
if i
if C{i-1}.NP | (C{i-1}.Stack <= T.Esp)
if C{i-1}.NT
NT.i = C{i-1}.i
fi
TC.i --
> Next
fi
fi
C{i}.P = M{i}
C{i}.Stack = T.Esp
C{i}.i = NT.i + 1
TC.i ++
end P
'
```

Разбор алгоритма:

Входные данные:

- $M\{\}$ — таблица CM (стабы)
- P — оригинальный адрес возврата (IP)
- T — состояние задачи (TST)
- A — аргументы системного вызова
- I — ID системного вызова
- i — текущий индекс в TCM

Шаги:

1. $i = TC.i$ — берем текущий индекс (свободная запись)
2. Next: — начало цикла очистки IDLE-записей

text

if i

Если есть записи в стеке ($i > 0$)

3. if $C\{i-1\}.NP \mid (C\{i-1\}.Stack \leq T.Esp)$
 - Проверяем предыдущую запись (индекс $i-1$)
 - NP (Not Present) — маркер IDLE (паразитная запись)
 - ИЛИ стек этой записи ($Stack$) $\leq$ текущий ESP
 - **Условие:** запись "мертвая" (либо помечена IDLE, либо стек ушел выше)
4. if $C\{i-1\}.NT$ — если у записи есть NT-сервис
 - $NT.i = C\{i-1\}.i$ — сохраняем ID сервиса
 - (для последующей обработки возврата из сервиса)
5. $TC.i--$ — удаляем запись (сжимаем стек)
 - Переход на Next (повторяем проверку)
6. $C\{i\}.P = M\{i\}$ — сохраняем оригинальный адрес возврата
 - $M\{i\}$ — это адрес возврата из CAPI (который лежит в стеке)

- `mpj` — это адрес возврата из `CALL` (который лежит в стеке)

7. `C{i}.Stack = T.Esp` — сохраняем текущий ESP

- Нужен для проверки валидности записи при возврате

8. `C{i}.i = NT.i + 1` — сохраняем ID системного вызова

- `NT.i` — глобальный счетчик вызовов
- `+1` для уникальности

9. `TC.i ++` — расширяем стек (инкремент индекса)

10. `end P` — возврат в вызывающий код

Аналог на C++:

cpp

```
void TcPush(bool cleanup) {
    int i = Tcm.I; // Текущий индекс

    if (cleanup) {
        while (i > 0) {
            CM* prev = &Tcm.Cm[i - 1];

            // Проверяем, мертвая ли запись
            if (prev->Nid & NID_NP || prev->Stack <= State->Rgp.rEsp) {
                // Если это системный вызов — сохраняем ID
                if (prev->Nid & NID_NT) {
                    Service.I = prev->Nid & NID_ID;
                }
                i--; // Удаляем запись
                continue;
            }
            break;
        }
    }

    // Сохраняем новую запись
    CM* curr = &Tcm.Cm[i];
    curr->Ip = GetReturnAddress(); // Оригинальный RET
    curr->Stack = State->Rgp.rEsp; // Текущий ESP
    curr->Nid = Service.I + 1;     // ID сервиса
}
```

```

    Tcm.l = i + 1;          // Расширяем стек
}

```

2. Алгоритм TrTcmGate (возврат через TCM)

Комментарий в коде:

```

asm

comment '
M TCM{}
P Ip
T TST
A Nt-args
I Nt-id
i TCM-index

i = (M{} - [T.Esp]) >> 3
Next:
if i
j = i - 1
if C{j}.NP | (C{j}.Stack <= T.Esp)
if C{j}.NT
NT.i = C{j}.i
!N{NT.i}.i
F.NR
fi
j --
> Next
fi
fi
TC.i = i
F.CM
'

```

Разбор алгоритма:

Входные данные:

- $M\{\}$ — таблица CM (стабы)
- P — адрес возврата из CALL (в стабе)
- T — состояние задачи (TST)
- i — вычисляемый индекс

Шаги:

1. $i = (M\{\} - [T.Esp]) \gg 3$ — вычисляем индекс
 - $M\{\}$ — адрес возврата из CALL (в стеке)
 - $[T.Esp]$ — текущий ESP (указывает на стек)
 - $\gg 3$ — деление на 8 (выравнивание)
 - **Суть:** Адрес возврата указывает на стаб в таблице CM, вычисляем его индекс
2. Next: — цикл очистки IDLE-записей при возврате

text

if i

Если есть записи ($i > 0$)

3. $j = i - 1$ — берем предыдущую запись
4. if $C\{j\}.NP \mid (C\{j\}.Stack \leq T.Esp)$
 - Проверяем, мертвая ли запись
 - Если да — удаляем
5. if $C\{j\}.NT$ — если это системный вызов
 - $NT.i = C\{j\}.i$ — восстанавливаем ID
 - $!N\{NT.i\}.i$ — очищаем запись в таблице NT
 - $F.NR$ — устанавливаем флаг TF_NR (NT Return)
6. $i--$ — удаляем запись
 - Переход на Next
7. $TC.i = i$ — устанавливаем новый индекс (сжатый стек)
8. $F.CM$ — устанавливаем флаг TF_CM (возврат через CM)

Аналог на C++:

cpp

```
void TrTcmGate() {
    // Получаем адрес возврата из стека (это адрес стаба в CM)
    uint32_t stubAddr = Pop(); // Адрес возврата из CALL

    // Вычисляем индекс в таблице CM
    uint32_t i = (stubAddr - Tcm.Map) / 8;

    // Очищаем мертвые записи
    while (i > 0) {
        CM* prev = &Tcm.Cm[i - 1];

        if (prev->Nid & NID_NP || prev->Stack <= State->Rgp.rEsp) {
            if (prev->Nid & NID_NT) {
                // Восстанавливаем ID системного вызова
                Service.I = prev->Nid & NID_ID;
                // Очищаем запись в таблице NT
                Service.Nt[Service.I].Cid = 0;
                Flags |= TF_NR; // NT Return
            }
            i--; // Удаляем запись
            continue;
        }
        break;
    }

    // Восстанавливаем оригинальный адрес возврата
    CM* curr = &Tcm.Cm[i];
    uint32_t originalIp = curr->Ip;

    // Сжимаем стек TCM
    Tcm.I = i;
    Flags |= TF_CM;

    // Передаем управление на оригинальный адрес
    jmp originalIp;
}
```

3. Алгоритм обработки системного вызова (TrService)

Комментарий в коде:

```
asm

comment '
M TCM{}
P Ip
T TST
A Nt-args
I Nt-id
i TCM-index

i = TcPush()
C{i}.NT
if INTERRUPT
A = T.Edx
C{i}.P -2
C{i}.P <=> T.Eip
if SYSENTER
T.Esp = T.Edx
A = T.Edx +8
C{i}.P <=> [T.Esp]
if CMSWITCH
A = T.Edx +8
C{i}.P <=> [T.Esp]
fi
n = NT.i
S{n}.A = A
S{n}.I = T.Eax
S{n}.i = i
C{i}.i = NT.i +1
NT.i ++
'
```

Разбор алгоритма:

Входные данные:

- i — индекс, полученный из `TcPush()`
- Тип системного вызова (`INTERRUPT` / `SYSENTER` / `CMSWITCH`)

Шаги:

1. $i = \text{TcPush}()$ — сохраняем контекст вызова в TCM
2. $C\{i\}.NT$ — помечаем запись как NT-вызов
3. **Тип 1: INTERRUPT (INT 0x2E)**

text

```
A = T.Edx      ; Аргументы по адресу EDX
C{i}.P - 2      ; IP' = текущий IP - 2 (пропустить INT 0x2E)
C{i}.P <=> T.Eip ; Сохраняем оригинальный IP
```

4. Тип 2: SYSENTER

text

```
T.Esp = T.Edx   ; Новый ESP из EDX
A = T.Edx + 8    ; Аргументы по адресу EDX+8
C{i}.P <=> [T.Esp] ; IP' из стека (KiFastSystemCallRet)
```

5. Тип 3: CMSWITCH (WOW64)

text

```
A = T.Edx + 8    ; Аргументы по адресу EDX+8
C{i}.P <=> [T.Esp] ; IP' из стека
```

6. $n = NT.i$ — берем текущий индекс в таблице NT
7. $S\{n\}.A = A$ — сохраняем указатель на аргументы
8. $S\{n\}.I = T.Eax$ — сохраняем ID системного вызова (из EAX)
9. $S\{n\}.i = i$ — сохраняем индекс в TCM
10. $C\{i\}.i = NT.i + 1$ — сохраняем ID в записи CMS
11. $NT.i ++$ — инкрементируем счетчик NT-вызовов

Аналог на C++:

cpp

```
void TrService(uint32_t serviceType) {
    // 1. Сохраняем контекст в TCM
    int i = TcPush(true);
    Tcm.Cm[i].Nid |= NID_NT;

    uint32_t argsAddr;
    uint32_t ipPrime;

    switch (serviceType) {
    case VM_SINT: // INT 0x2E
        argsAddr = State->Rgp.rEdx;
        ipPrime = State->Control.rEip - 2; // Пропускаем INT 0x2E
        State->Control.rEip = ipPrime;
        break;

    case VM_SENT: // SYSENTER
        State->Rgp.rEsp = State->Rgp.rEdx;
        argsAddr = State->Rgp.rEdx + 8;
        ipPrime = *(uint32_t*)State->Rgp.rEsp; // KiFastSystemCallRet
        break;

    case VM_SWOW: // WOW64
        argsAddr = State->Rgp.rEdx + 8;
        ipPrime = *(uint32_t*)State->Rgp.rEsp;
        break;
    }

    // 2. Сохраняем в таблице NT
    int n = Service.I;
    Service.Nt[n].pArgs = (void*)argsAddr;
    Service.Nt[n].Sid = State->Rgp.rEax; // ID системного вызова
    Service.Nt[n].Cid = i; // Индекс в TCM
    Tcm.Cm[i].Nid |= (n + 1); // Сохраняем ID
    Service.I++;
}
```

4. Алгоритм Sequencer (инструментация)

Комментарий в коде:

```
asm

comment '
if !Carry?
    btr esi,0
.if !Carry?
    ; Инструментация
.endif
endif
test ebp,ebp
Exit:
'
```

Разбор алгоритма:

Что делает Sequencer?

- Это callback, который вызывается **перед выполнением каждой инструкции**
- Позволяет встраивать произвольный код (инструментацию)

Шаги:

1. `btr esi,0` — проверяем бит 0 в `ia` (инструкция уже обработана?)
 - Если бит сброшен (`!Carry?`) — инструкция не инструментируется
2. **Инструментация:**
 - Можно добавить код до/после инструкции
 - Например: логирование, профилирование, подсчет инструкций
3. `test ebp,ebp` — проверяем флаг ZF
 - Используется для пропуска инструментации при необходимости

Пример Sequencer для логирования:

cpp

```
bool Sequencer(uint32_t ip, IA* ia, TST* state) {  
    // Логируем каждую инструкцию  
    printf("IP: 0x%X, Opcode: 0x%X, Size: %d\n",  
        ip, ia->Opcode, ia->Size);  
  
    // Если это CALL — логируем особенно  
    if (ia->Class == I_CALL) {  
        printf(" CALL to 0x%X\n", ia->Mem.Vx.La);  
    }  
  
    return true; // Продолжаем выполнение  
}
```

5. Алгоритм Filter (фильтрация NT-вызовов)

Комментарий в коде:

asm

```
comment '  
2st(kiret) will not be called in scylla.  
,
```

Разбор:

Что делает Filter?

- Обработывает возвраты из системных вызовов
- Позволяет фильтровать/модифицировать поведение

Особенность:

- 2st(kiret) — второй вызов KiRet (возврат из исключения)
- В Scylla (инструменте для дампа процессов) это не вызывается

Шаги:

1. Проверяем бит 0 в Nt (флаг)
2. Если установлен — это особый случай (например, NtContinue)
3. Возвращаем ZF для пропуска/обработки

6. Алгоритм обработки NtContinue (переключение контекста)**Комментарий в коде:**

asm

comment '

if ([Ebp].Rgp.rEsp < Ecx) && (fs:[TEB.Tib.StackBase] > Ecx)

mov edi, NT_CONTINUE.Context[ecx]

if ([Ebp].Rgp.rEsp < Edi) && (fs:[TEB.Tib.StackBase] > Edi)

test [edi].ContextFlags, CONTEXT_I386

if !Zero?

test [edi].ContextFlags, CONTEXT_CONTROL

if !Zero?

mov ecx,cs

mov edx,ss

if (W[Edi].rCs == Cx) && (W[Edi].rSs == Dx)

bts [eax].Nid, NID_NP ; Opt

invoke TcPush, FALSE

push [ebp].Rgp.rEsp

add edx,2

pop [eax].Stack

xchg CONTEXT.rEip[edi], edx

mov [eax].Ip, edx

endif

endif

endif

endif

endif

'

Debug:

газур.

Что делает NtContinue?

- Изменяет контекст выполнения (переключает поток)
- Управление может перейти на **совершенно другой адрес**

Проблема для TCM:

- Оригинальный RET-адрес теряется
- В стеке TCM остается "висячая" запись

Решение:

1. Проверяем, что новый контекст находится в том же сегменте (CS/SS)
2. Если да — создаем **дополнительную запись** в TCM
3. Помечаем оригинальную запись как NID_NP (IDLE)
4. При возврате через TrTcmGate IDLE-запись будет удалена

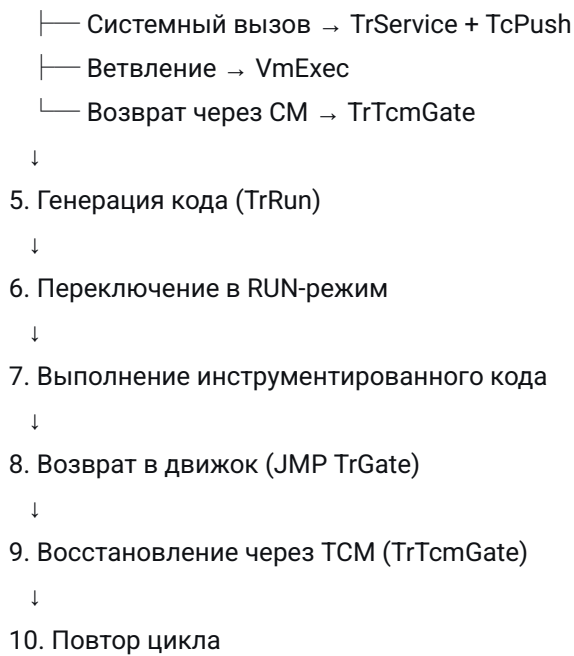
Ключевой момент:

- `bts [eax].Nid, NID_NP` — устанавливаем маркер IDLE
- `invoke TcPush, FALSE` — создаем новую запись для нового контекста
- `xchg CONTEXT.rEip[edi], edx` — подменяем IP в контексте

Связь всех алгоритмов вместе

text

1. Вход в движок (TrGate)
 - ↓
2. Декодирование инструкции (DiDecode)
 - ↓
3. Sequencer (инструментация)
 - ↓
4. Обработка класса инструкции (TrCase)
 - ↓
 - Простая инструкция → TrRun (генерация кода)



Итог: Алгоритмы TCM в DYE

Алгоритм	Назначение	Ключевая операция
TcPush	Сохранение вызова в CMS	Индексный CALL + очистка IDLE
TrTcmGate	Возврат через TCM	Восстановление индекса + удаление IDLE
TrService	Обработка NT-вызовов	Сохранение аргументов + ID сервиса
Sequencer	Инструментация	Callback перед выполнением
Filter	Фильтрация NT-возвратов	Обработка особых случаев
NtContinue	Переключение контекста	Двойная запись в CMS + IDLE

Ключевая идея: Все алгоритмы работают вокруг **индексного CALL** и **стека CMS** в TLS, обеспечивая:

- Быструю трассировку ($O(1)$)

- Поддержку рекурсии
- Обработку системных вызовов
- Очистку "мертвых" записей (IDLE)

Хочешь:

1. Разобрать конкретный алгоритм глубже (например, как работает `VmExeс` для ветвлений)?
2. Посмотреть, как DYE обрабатывает **SEH** (Structured Exception Handling)?
3. Или как движок работает в **многопоточном** окружении?

Собери алгоритмы ткм в одно сообщение.

Отлично, давай соберем **все алгоритмы TCM (Thread Call Map)** из кода DYE в единую, целостную картину.

Ниже — **полная спецификация TCM**, собранная из комментариев, ассемблерных вставок и логики их связей. Это не просто обфускация, а **инфраструктура трассировки вызовов**, построенная на индексных CALL-ах, TLS-стеке и пассивной очистке мусора.

1. Глобальная архитектура TCM

Суть: Каждый вызов в программе трансформируется в **индексный CALL** к таблице стабов (`CmMap`). В стеке потока вместо реального адреса возврата лежит адрес стаба. Реальные адреса возврата сохраняются в TLS-стеке `CMS` .

Структуры:

```
cpp
```

```
// Таблица стабов (выровнена на 8 байт)
```

```
CmMap:
```

```
[0]: INT 0x2E + CALL TrTcmGate + NOP
```

```
[1]: INT 0x2E + CALL TrTcmGate + NOP
```

...

```
// Стек вызовов в TLS (CM Stack)
struct CM {
    PVOID Ip;    // Оригинальный адрес возврата
    PVOID Stack; // Значение ESP в момент вызова
    ULONG Nid;   // ID (сервиса) + флаги (NT/NP/INT/SEN/WOW)
};

// Управляющая структура TCM в TLS
struct TCM {
    ULONG I;    // Текущий индекс (свободная вершина стека)
    PCM Cm;     // Указатель на массив CM (CM_LIM=64)
    PVOID Map;  // База таблицы стабов (CmMap)
};
```

2. Алгоритм входа в вызов (маршрутизация / TcPush)

Вызывается: При перехвате CALL, системного вызова (NT), или при входе в движок.

Цель: Сохранить оригинальный RET-адрес в CMS, подменить его на стаб.

Алгоритм (псевдокод из комментариев):

cpp

```
void TcPush(BOOLEAN Cleanup) {
    // 1. Берем текущую свободную вершину
    int i = Tcm.I;

    // 2. [ОПЦИОНАЛЬНО] Цикл очистки "мертвых" записей (IDLE-GC)
    if (Cleanup) {
        while (i > 0) {
            CM* prev = &Tcm.Cm[i - 1];

            // Если запись помечена как IDLE (NP) ИЛИ ее стек ушел выше текущего ESP
            if ((prev->Nid & NID_NP) || (prev->Stack <= State->rEsp)) {

                // Если это был системный вызов — сохраняем его ID для обработчика
                if (prev->Nid & NID_NT) {
                    Service.I = (prev->Nid & NID_ID);
                }
            }
        }
    }
}
```



```

        i--; // Удаляем (сжимаем стек)
        continue;
    }
    break;
}
}

// 3. Сохраняем новую запись в CMS[i]
CM* curr = &Tcm.Cm[i];
curr->Ip = [ESP];          // Оригинальный RET-адрес (со стека)
curr->Stack = State->rEsp;   // Текущий ESP потока
curr->Nid = 0;              // Очищаем флаги

// 4. Подмена адреса возврата на стеке потока
[ESP] = Tcm.Map + (i * 8);  // Адрес стаба (индексный CALL)

// 5. Расширяем стек TCM
Tcm.I = i + 1;
}

```

3. Алгоритм возврата через стаб (TrTcmGate)

Вызывается: Когда выполнение доходит до RET в функции, управление возвращается на стаб (в CALL TrTcmGate внутри стаба).

Цель: Вычислить индекс по адресу стаба, восстановить оригинальный RET-адрес, сжать стек TCM.

Алгоритм (из комментариев):

```

cpp

void TrTcmGate() {
    // 1. Получаем адрес возврата из стека (это адрес внутри стаба: Map + i*8 + 6)
    uint32_t StubRetAddr = POP();

    // 2. Вычисляем индекс (деление на 8, отбрасываем выравнивание)
    uint32_t i = ((StubRetAddr - Tcm.Map) & ~7) / 8;

    // 3. [ЦИКЛ ОЧИСТКИ] Удаляем IDI E-записи, которые оказались "выше" текущего стека
}

```

```

// Сдвинуть стек на один байт, который указывает на начало текущего стека
while (i > 0) {
    CM* prev = &Tcm.Cm[i - 1];

    if ((prev->Nid & NID_NP) || (prev->Stack <= State->rEsp)) {
        if (prev->Nid & NID_NT) {
            Service.I = (prev->Nid & NID_ID); // Восстанавливаем ID
            Flags |= TF_NR;                // Вводим флаг NT-Return
        }
        i--; // Сжимаем стек
        continue;
    }
    break;
}

// 4. Восстанавливаем оригинальный вектор из текущей записи
uint32_t Originallp = Tcm.Cm[i].lp;

// 5. Устанавливаем новую вершину стека TCM (сжатие)
Tcm.I = i;
Flags |= TF_CM; // Вводим флаг возврата через CM

// 6. Передаем управление
jmp Originallp;
}

```

4. Алгоритм обработки системных вызовов (NT / TrService)

Вызывается: При выполнении INT 0x2E , SYSENTER или WOW64-перехода.

Цель: Сохранить контекст системного вызова, аргументы, ID сервиса, и связать его с записью в CMS.

Алгоритм (из комментариев):

```

cpp

void TrService(uint32_t ServiceType) {
    // 1. Сохраняем RET в CMS и получаем индекс i
    int i = TcPush(TRUE);
    Tcm.Cm[i].Nid |= NID_NT; // Помечаем как NT-вызов
}

```

```
uint32_t ArgsPtr;
uint32_t NewIp;

// 2. Определяем тип шлюза и берем аргументы
switch (ServiceType) {
    case VM_SINT: // INT 0x2E (старый KiSystemService)
        ArgsPtr = State->rEdx;           // Аргументы по EDX
        NewIp = State->rEip + 2;         // Пропускаем INT 0x2E
        Tcm.Cm[i].Nid |= NID_INT;
        break;

    case VM_SENT: // SYSENTER (fastcall)
        State->rEsp = State->rEdx;       // Новый ESP от ядра
        ArgsPtr = State->rEdx + 8;       // Аргументы со смещением
        NewIp = *(uint32_t*)State->rEsp; // Кидаем на KiFastSystemCallRet
        Tcm.Cm[i].Nid |= NID_SEN;
        break;

    case VM_SWOW: // WOW64 (32-bit on 64-bit)
        ArgsPtr = State->rEdx + 8;
        NewIp = *(uint32_t*)State->rEsp; // Снимаем со стека
        Tcm.Cm[i].Nid |= NID_WOW;
        break;
}

// 3. Сохраняем IP возврата из сервиса
Tcm.Cm[i].Ip = NewIp;

// 4. Записываем в таблицу NT (глобальный массив сервисов)
int n = Service.I; // Текущий ID сервиса
Service.Nt[n].pArgs = ArgsPtr; // Указатель на стек с аргументами
Service.Nt[n].Sid = State->rEax; // ID системного вызова (из EAX)
Service.Nt[n].Cid = i; // Индекс в CMS (для обратной связи)

// 5. Связываем запись CMS с ID сервиса
Tcm.Cm[i].Nid |= (n + 1); // Сохраняем ID (сдвиг на 1 для флагов)

// 6. Инкрементируем глобальный счетчик сервисов
Service.I++;
}
```

5. Алгоритм обработки NtContinue (переключение контекста)

Особенность: NtContinue не возвращается по обычному RET-адресу, а прыгает на CONTEXT.Eip .

Цель: Не дать "замусорить" CMS мертвой записью, создав две записи и пометив первую как IDLE.

Алгоритм (из комментариев):

cpp

```
void HandleNtContinue(PCONTEXT Ctx) {
    // 1. Проверяем, что новый контекст находится в том же сегменте (CS:IP)
    if (Ctx->SegCs == CurrentCS && Ctx->SegSs == CurrentSS) {

        // 2. Берем текущую вершину стека (запись, которая создалась при вызове NtContinue)
        int i = Tcm.I - 1;

        // 3. Помечаем ЭТУ запись как IDLE (Not Present) — она станет "паразитной"
        Tcm.Cm[i].Nid |= NID_NP;

        // 4. Пушим НОВУЮ запись для адреса из CONTEXT.Eip (чтобы туда корректно вернуться)
        TcPush(FALSE);

        // 5. Берем свежесозданную запись (вершину)
        CM* curr = &Tcm.Cm[Tcm.I - 1];

        // 6. Сохраняем в нее адрес из CONTEXT (как будто это RET-адрес)
        curr->Ip = Ctx->Eip;

        // 7. Подменяем Eip в контексте на адрес стаба (чтобы вернуться в движок)
        Ctx->Eip = Tcm.Map + (Tcm.I - 1) * 8;
    }
}
```

Результат:

- В CMS лежат две записи:
 - [i] — старая (IDLE), указывает на возврат из NtContinue .
 - [i+1] — новая, указывает на CONTEXT.Eip .

Цель: Обработать особые случаи (например, `NtCallbackReturn`).

Алгоритм (из комментариев):

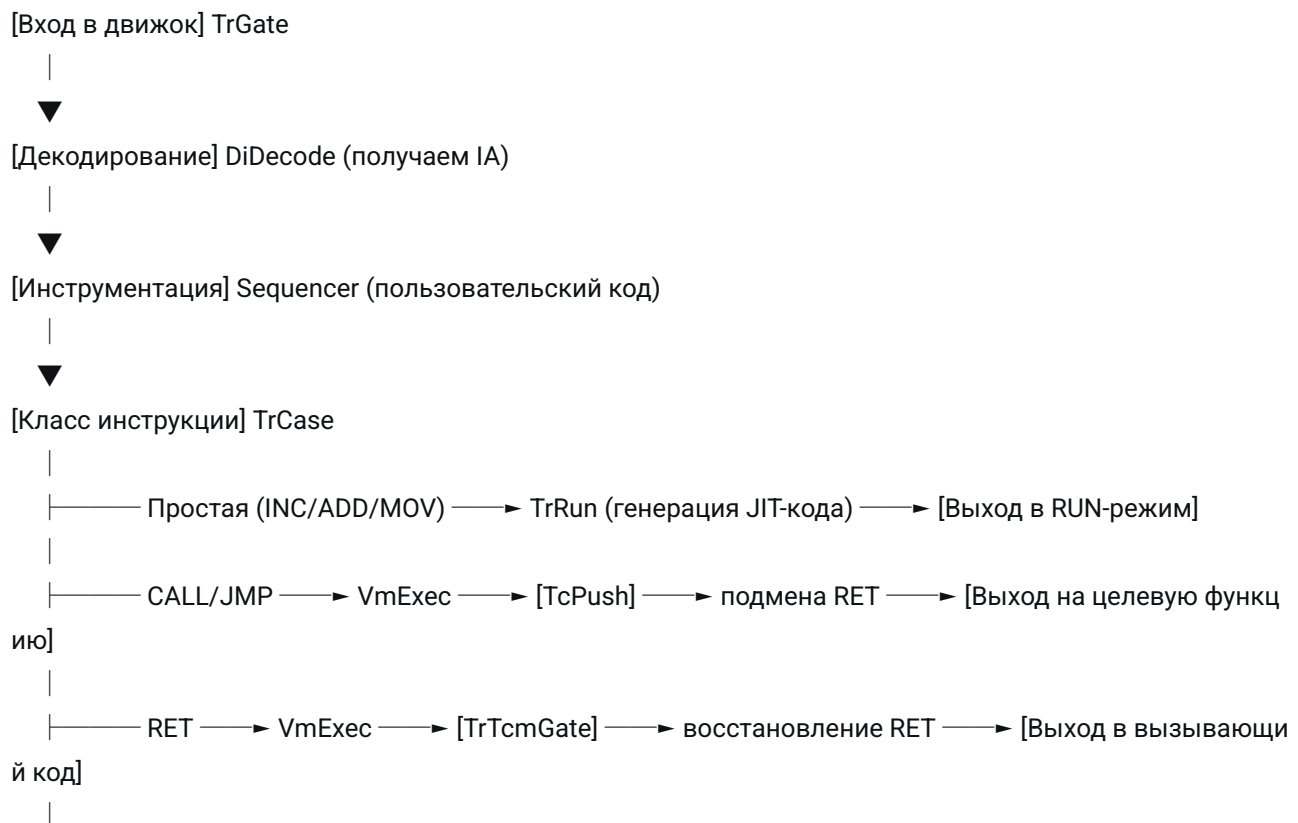
cpp

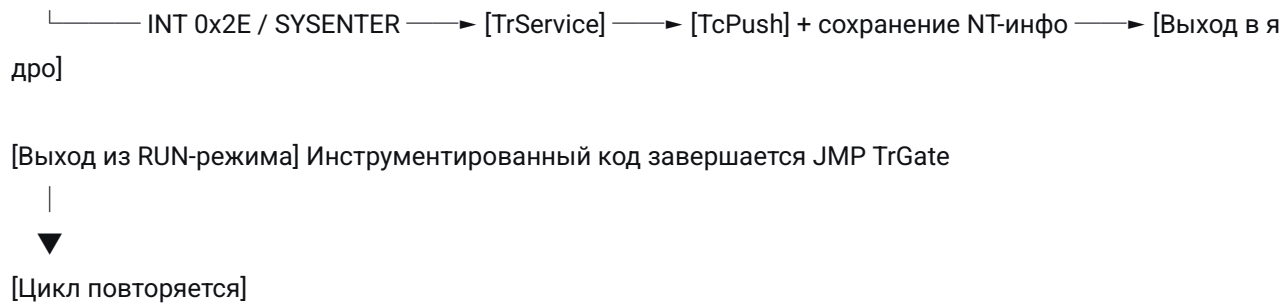
```
bool Filter(NT* NtEntry) {
    // Проверяем младший бит структуры NT (флаг, взведенный в TrService)
    if (NtEntry->Flags & 1) {
        // Это особый случай (например, KiCallbackReturn)
        // Делаем дополнительную очистку или корректировку стека
        // В коде DYE: "2st(kiret) will not be called in scylla"
    }

    // Возвращаем ZF: если 0 (ZF=1) — пропускаем дальнейшую обработку
    return (NtEntry->Sid == 0);
}
```

8. Полный цикл выполнения (связь всех алгоритмов)

text





Итог: Ключевые принципы TCM

Принцип	Реализация
Идентификация вызывающего кода	Индексный CALL: call [Map + I*8]
Хранение оригинальных RET-адресов	Стек CMS в TLS (массив структур CM)
Поддержка рекурсии	Инкремент/декремент Tcm.I (вершина стека)
Очистка "битых" возвратов (NtContinue)	Маркер NID_NP (IDLE) + пассивная сборка мусора при Push/Pop
Связка с системными вызовами	Двойной индекс: CMS.i ↔ NT.I
Скорость	O(1) операции (только сдвиги, сложения, чтение из TLS)
Потокобезопасность	Все данные в TLS (каждый поток имеет свой TCM)

Это не защита, а инфраструктура динамической бинарной инструментации (DBI), где TCM играет роль точного трекера потока управления, позволяя движку всегда знать: кто вызвал текущую функцию, откуда мы пришли и куда должны вернуться.

в более ранней версии была обработка:

```
; + EBX: POINTER TO ARG'S OF SERVICE
... ESI, NT.C
```

```
; + ESI: P TLS
; + EDI: PNTSTATE
```

```
assume esi:PTLS
```

```
NT_CONTINUE struct
Context      PCONTEXT ?
TestAlert    BOOLEAN ?
NT_CONTINUE ends
PNT_CONTINUE typedef ptr NT_CONTINUE
```

```
; +
; T-route via CM.
;
```

```
SyContinue proc C
    assume ebx:PNT_CONTINUE
```

```
    mov edi,[ebx].Context
    assume edi:PCONTEXT
    test [edi].ContextFlags,CONTEXT_CONTROL
    jz Skip
```

```
; Get current CME.
```

```
    mov eax,[esi].Tr.Service
    assume eax:PCME
```

```
; Mark CME(nt-gate) as IDLE.
```

```
    bts [eax].Flags,CME_IDLE
```

```
; Add new CME for T-route.
```

```
    inc [esi].Tcm.Index
    add eax,sizeof(CME)
```

```
; Calc CM-call pointer.
```

```
    mov ecx,[esi].Tcm.Index
    shl ecx,3 ; Index of CM.
    add ecx,[esi].Tcm.Map + Index
```



```

        add ecx,[esi].Tcm.Map ; ip

; Route via context.

        xchg [edi].rEip,ecx
        and [eax].Flags,0
        mov [eax].Ip,ecx
Skip:
        ret
SyContinue endp

NT_CALLBACK_RETURN struct
OutputBuffer      PVOID ?
OutputLength      ULONG ?
Status            NTSTATUS ?
NT_CALLBACK_RETURN ends
PNT_CALLBACK_RETURN typedef ptr NT_CALLBACK_RETURN

SyCallbackReturn proc C

; Mark CME(nt-gate) as IDLE.

        mov eax,[esi].Tr.Service ; * rename
        assume eax:PCME
        bts [eax].Flags,CME_IDLE

        xor eax,eax
        ret
SyCallbackReturn endp

NT_RAISE_EXCEPTION struct
ExceptionRecord   PEXCEPTION_RECORD ?
ContextRecord     PCONTEXT ?
FirstChance       BOOLEAN ?
NT_RAISE_EXCEPTION ends
PNT_RAISE_EXCEPTION typedef ptr NT_RAISE_EXCEPTION

SyRaiseException proc C
        assume ebx:PNT_RAISE_EXCEPTION
        lea ebx,[ebx].ContextRecord
        jmp SyContinue

```

```
    jmp SyContinue  
SyRaiseException endp
```

Отлично! Это **ранняя версия TCM-обработчиков для системных вызовов**, которая показывает **эволюцию архитектуры**. Здесь видно, как TCM развивался от простой маршрутизации через контекст до полноценной системы трассировки с IDLE-маркерами и очисткой мусора.

Давай разберем эту раннюю версию и сравним с финальной (из DYE.asm).

Ранняя версия: Структура и подход

1. Базовые структуры

asm

NT_CONTINUE struct

Context PCONTEXT ? ; Указатель на контекст
TestAlert BOOLEAN ? ; Флаг тестового оповещения

NT_CONTINUE ends

NT_CALLBACK_RETURN struct

OutputBuffer PVOID ? ; Буфер вывода
OutputLength ULONG ? ; Длина вывода
Status NTSTATUS ? ; Статус возврата

NT_CALLBACK_RETURN ends

NT_RAISE_EXCEPTION struct

ExceptionRecord PEXCEPTION_RECORD ?
ContextRecord PCONTEXT ?
FirstChance BOOLEAN ?

NT_RAISE_EXCEPTION ends

Ключевое отличие: В ранней версии структуры **явно описаны** для каждого системного вызова. В финальной версии (DYE.asm) используется **обобщенная** структура NT с указателем на аргументы.

2. Обработчик NtContinue (SyContinue)

Код:

asm

SyContinue proc C

mov edi, [ebx].Context ; Берем CONTEXT из аргументов

assume edi:PCONTEXT

test [edi].ContextFlags, CONTEXT_CONTROL

jz Skip ; Если нет CONTROL-флагов — пропускаем

; 1. Берем текущую запись в CMS (последнюю)

mov eax, [esi].Tr.Service ; Указатель на текущую CME

assume eax:PCME

; 2. Помечаем её как IDLE (паразитная)

bts [eax].Flags, CME_IDLE

; 3. Создаем НОВУЮ запись для T-route (переход на новый контекст)

inc [esi].Tcm.Index ; Расширяем стек

add eax, sizeof(CME) ; Переходим на новую запись

; 4. Вычисляем адрес стаба в таблице CM

mov ecx, [esi].Tcm.Index

shl ecx, 3 ; *8 (выравнивание)

add ecx, [esi].Tcm.Map ; Адрес стаба (Ip')

; 5. Подменяем Eip в контексте на адрес стаба

xchg [edi].rEip, ecx ; Теперь контекст укажет на стаб

; 6. Сохраняем оригинальный Eip в новой записи CMS

and [eax].Flags, 0 ; Очищаем флаги

mov [eax].Ip, ecx ; Сохраняем оригинальный Eip

Skip:

ret

SyContinue endp

Что делает:

1. Текущая запись CMS (вызов NtContinue) помечается как **IDLE**.
2. Создается **новая запись CMS** для адреса из CONTEXT.Eip .
3. Адрес в CONTEXT.Eip подменяется на **стаб из таблицы CM**.
4. При возврате из NtContinue управление пойдет на стаб → движок восстановит оригинальный Eip.

Связь с финальной версией:

- В DYE.asm этот код **интегрирован** в TrService (часть, обрабатывающая NtContinue).
- В ранней версии — отдельный обработчик.

3. Обработчик NtCallbackReturn (SyCallbackReturn)

Код:

asm

```
SyCallbackReturn proc C
; Помечаем текущую запись CMS как IDLE
mov eax, [esi].Tr.Service
assume eax:PCME
bts [eax].Flags, CME_IDLE

xor eax, eax
ret
SyCallbackReturn endp
```

Что делает:

- Просто помечает текущую запись CMS как **IDLE**.
- Не создает новую запись (в отличие от NtContinue).

Почему:

- NtCallbackReturn возвращает управление **в тень (shadow)** или в вызывающий код.
- Если возврат идет в тень, то IDLE-запись будет удалена при следующем системном вызове (через TcPush с Cleanup=true).

Связь с финальной версией:

- В DYE.asm это обрабатывается через **IDLE-очистку** в TcPush и TrTcmGate .
- Отдельный обработчик для NtCallbackReturn **отсутствует** — используется общая логика.

4. Обработчик NtRaiseException (SyRaiseException)

Код:

asm

```
SyRaiseException proc C
    assume ebx:PNT_RAISE_EXCEPTION
    lea ebx, [ebx].ContextRecord
    jmp SyContinue
SyRaiseException endp
```

Что делает:

- Просто перенаправляет на SyContinue .
- Берет ContextRecord из структуры исключения.

Логика:

- Исключение может изменить контекст выполнения (как NtContinue).
- Поэтому нужно создать новую запись CMS для нового контекста.

Связь с финальной версией:

- В DYE.asm исключения обрабатываются через KiUED (обработчик исключений).
- Там используется та же логика с IDLE-маркером.

Сравнение ранней и финальной версий

Аспект	Ранняя версия	Финальная версия (DYE.asm)
Структуры	Явные для каждого NT-вызова	Обобщенная структура NT
Обработчики	Отдельные процедуры (SyContinue , SyCallbackReturn)	Интегрированы в TrService + TcPush
IDLE-маркер	CME_IDLE (явный бит)	NID_NP (Not Present)
Создание новой записи	Ручное (inc Index + add eax)	Автоматическое через TcPush(FALSE)
Очистка IDLE	Ручная (в обработчиках)	Автоматическая (в TcPush и TrTcmGate)
Гибкость	Жесткая (привязана к конкретным NT-вызовам)	Гибкая (общая логика для всех)

Эволюция архитектуры TCM

Ранняя версия (SyContinue/SyCallbackReturn):

text

1. Вызов NtContinue
- ↓
2. SyContinue: помечаем текущую запись как IDLE
- ↓
3. SyContinue: создаем новую запись для CONTEXT.Eip
- ↓
4. Возврат → управление на стаб → TrTcmGate
- ↓
5. TrTcmGate: удаляем IDLE-записи

Плюсы:

- Простота (каждый обработчик делает ровно то, что нужно).
- Явный контроль над каждой ситуацией.

Минусы:

- Много дублирования.
- Нужно писать отдельный обработчик для каждого NT-вызова.
- Трудно масштабировать.

Финальная версия (DYE.asm):

text

1. Вызов NtContinue (или любого NT-вызова)
↓
2. TrService: сохраняем в CMS через TcPush(TRUE)
↓
3. TcPush(TRUE): автоматически удаляет IDLE-записи
↓
4. Если это NtContinue — создается ДВОЙНАЯ запись:
 - Первая: IDLE (исходный вызов)
 - Вторая: новая (CONTEXT.Eip)↓
5. Возврат → TrTcmGate: автоматически удаляет IDLE-записи

Плюсы:

- Один обработчик для всех NT-вызовов (TrService).
- Автоматическая очистка IDLE (в TcPush и TrTcmGate).
- Легко расширять (добавлять новые NT-вызовы без модификации).

Минусы:

- Сложнее понять (логика размазана по нескольким функциям).
- Требуется аккуратной работы с индексами.

Как это работает в связке с DYE

1. Вход в системный вызов (ранняя версия)

asm

```
; При INT 0x2E:  
; 1. Сохраняем контекст в CMS (через TcPush)  
; 2. Помечаем запись как NT  
; 3. Если это NtContinue → вызываем SyContinue  
; 4. SyContinue создает IDLE-запись
```

2. Вход в системный вызов (финальная версия)

asm

```
; При INT 0x2E:  
TrService:  
    invoke TcPush, TRUE    ; Сохраняем контекст, удаляем IDLE  
    mov [eax].Nid, NID_NT  ; Помечаем как NT  
  
; Если это NtContinue (определяется по Sid)  
.if ([Ebp].Rgp.rEax == NtContinue_Sid)  
    ; Создаем дополнительную запись для нового контекста  
    invoke TcPush, FALSE  
    mov [eax].Nid, NID_NP ; Помечаем старую как IDLE  
    ; ... подмена CONTEXT.Eip  
.endif
```

3. Возврат из системного вызова

asm

```
TrTcmGate:  
    ; 1. Вычисляем индекс  
    ; 2. В цикле удаляем IDLE-записи (NID_NP)  
    ; 3. Восстанавливаем оригинальный IP  
    ; 4. Сжимаем стек
```


Пример: Обработка NtContinue в финальной версии

Ранняя версия:

asm

SyContinue:

```
mov eax, [esi].Tr.Service
bts [eax].Flags, CME_IDLE ; IDLE
inc [esi].Tcm.Index ; Новая запись
add eax, sizeof(CME)
mov ecx, [esi].Tcm.Index
shl ecx, 3
add ecx, [esi].Tcm.Map
xchg [edi].rEip, ecx ; Подмена
mov [eax].Ip, ecx
ret
```

Финальная версия (из TrService):

asm

; 1. Первая запись (IDLE)

```
invoke TcPush, TRUE
mov [eax].Nid, NID_NT
```

; 2. Если это NtContinue

```
.if ([Ebp].Rgp.rEax == NtContinue_Sid)
; Проверяем контекст
mov edi, NT_CONTINUE.Context[ecx]
.if (валидный контекст)
; Помечаем первую запись как IDLE
bts [eax].Nid, NID_NP
```

; Создаем вторую запись (новая)

```
invoke TcPush, FALSE
add edx, 2
xchg CONTEXT.rEip[edi], edx
mov [eax].Ip, edx
```

```
.endif
.endif
```

Итог: Эволюция TCM

Этап	Характеристика	Ключевая идея
Ранний (SyContinue)	Явные обработчики для каждого NT-вызова	Ручное управление IDLE-записями
Промежуточный	Обобщение структур (NT_CONTINUE, NT_CALLBACK_RETURN)	Унификация аргументов
Финальный (DYE.asm)	Единый обработчик TrService + автоматическая очистка	IDLE через NID_NP + сборка мусора в TcPush / TrTcmGate

Ключевые улучшения:

- 1. **Масштабируемость** — не нужно писать обработчик для каждого NT-вызова.
- 2. **Автоматизация** — очистка IDLE встроена в базовые алгоритмы (TcPush , TrTcmGate).
- 3. **Надежность** — меньше шансов забыть обработать какой-то случай.

Это классическая эволюция от "прототипа" к "промышленному" решению.

Хочешь:

- 1. Разобрать, как обрабатываются **другие NT-вызовы** (NtClose, NtQueryInformationProcess)?
- 2. Посмотреть, как это **интегрируется** с генерацией кода (TrRun)?
- 3. Или как TCM работает в **многопоточном** окружении (TLS + синхронизация)?

Посмотрим управление стеком.

Отлично! Переходим к **управлению стеком** в DYE/TCM. Это одна из самых критичных частей движка, потому что DBI (Dynamic Binary Instrumentation) работает с **двумя стеками одновременно**:

1. **Стек потока** — где выполняется оригинальный и инструментированный код.
2. **Теневой стек (Shadow Stack)** — где работает сам движок (защищен от вмешательства).

Давай разберем все аспекты управления стеком из кода DYE.

1. Структуры для управления стеком

1.1 STACK structure (из H.inc)

cpp

STACK struct

Range RANGE <> ; Диапазон стека (Base/Limit)
Primary PVOID ? ; Первичная точка входа (при первом входе)
Current PVOID ? ; Текущий ESP
Bottom PVOID ? ; Нижняя граница (для проверки переполнения)
Working TST <> ; Рабочий фрейм (состояние задачи)

STACK ends

Назначение:

- Range — границы стека потока (из ТЕВ).
- Primary — сохраняется при первом входе в движок (для восстановления).
- Current — текущий ESP (обновляется при каждом входе).
- Bottom — нижняя граница стека (используется для детекции расширения стека).
- Working — фрейм для сохранения состояния задачи (регистры + IFR).

1.2 TIS с указателями на стеки

1.2 TLS с указателями на стек

c++

TLS struct

```

Flags    DWORD ?
Tcm      TCM <>
Stack    STACK <> ; Информация о стеке
State    PTST ? ; Указатель на Working (текущее состояние)
Ip       IP ? ; Текущий IP
Ip2      PVOID ? ; Предыдущий IP
Break    PVOID ?
Ia       IA <>
Work     PVOID ? ; Буфер для генерации кода
Service  TNT NT_LIM DUP(<>)
TLS ends

```

Ключевые поля:

- Stack — информация о стеке потока.
- State — указатель на Stack.Working (активный фрейм состояния).

1.3 Константы для теневого стека

asm

```

DTLS     equ PAGE_SIZE - 4 ; Указатель на TLS в стеке
DSTK     equ PAGE_SIZE - 2*4 ; Указатель на теньевой стек
DPIP     equ PAGE_SIZE - 3*4 ; Указатель на IP (для возврата)

```

Где это используется:

- В конце каждого стека потока (в TEB) резервируется место для теневого стека.
- fs:[DTLS] — указывает на структуру TLS.
- fs:[DSTK] — хранит ESP теневого стека.

2 Переключение на теньовой стек: TrGate

2. Переключаемся на теневой стек. TrGate

Код из DYE.asm:

asm

TrGate proc C

; 1. Переключаемся на теневой стек (сохраняем текущий ESP в TLS)

xchg D fs:[DSTK], esp

pushfd

pushad

; 2. Получаем TLS

%TLSR Ebx

; 3. Обновляем информацию о стеке

mov esi, fs:[DSTK] ; Текущий ESP (уже сохранен в TLS)

mov eax, fs:[TEB.Tib.StackBase] ; База стека (верх)

mov ecx, fs:[TEB.Tib.StackLimit] ; Лимит стека (низ)

and esi, NOT(PAGE_SIZE - 1) ; Выравниваем на границу страницы

mov edx, fs:[TEB.DeallocationStack] ; Стек для освобождения

sub esi, PAGE_SIZE ; Проверяем предыдущую страницу

sub ecx, PAGE_SIZE

; 4. Проверка: не переключился ли стек?

.if ([Ebx].Stack.Range.Base != Eax) && ([Ebx].Stack.Range.Limit != Edx)

; Стек переключился (например, при Fiber/SwitchToFiber)

mov [ebx].Stack.Range.Base, eax

mov [ebx].Stack.Range.Limit, edx

.endif

; 5. Проверка: расширился ли стек?

.if ([Ebx].Stack.Range.Base == Eax) && ([Ebx].Stack.Range.Limit == Edx)

.if ([Ebx].Stack.Bottom >= Esi) && (Ecx <= Esi)

; Стек расширился вниз

mov [ebx].Stack.Bottom, ecx

test D[ecx], ecx ; Touch-страница (физический commit)

.endif

.endif

; 6. Возвращаемся на стек потока

mov eax, [ebx].Stack.Bottom

ret 4

```
xchg D fs:[DS I K], eax
mov [ebx].Stack.Current, eax

; 7. Строим IFR-фрейм на стеке потока
popad
popfd
xchg D fs:[DSTK], esp

; IFR (IRET-фрейм)
pushfd
push cs
push eax    ; IP'
cld

; RGP (регистры)
pushad
mov ebp, esp
assume ebp:PTST

%TLSR Ebx
mov [ebx].State, esp

; ... продолжение (диспетчеризация)
TrGate endp
```

Что происходит по шагам:

Шаг 1: Переключение на теновой стек

```
asm

xchg D fs:[DSTK], esp
```

- Сохраняем **текущий ESP** (стек потока) в fs:[DSTK] .
- Загружаем в ESP **адрес тенового стека** (сохраненный ранее).
- **Важно:** xchg — атомарная операция, безопасная для многопоточности.

Шаг 2: Сохраняем контекст на теновом стеке

```
asm
```

```
pushfd  
pushad
```

- Сохраняем флаги и регистры на **теновом стеке** (не на стеке потока).
- Это защищает контекст движка от инструментированного кода.

Шаг 3: Получаем TLS

```
asm
```

```
%TLSR Ebx
```

- Загружаем в EBX указатель на структуру TLS (из `fs:[DTLS]`).

Шаг 4: Обновляем информацию о стеке потока

```
asm
```

```
mov esi, fs:[DSTK]  
mov eax, fs:[TEB.Tib.StackBase]  
mov ecx, fs:[TEB.Tib.StackLimit]  
and esi, NOT(PAGE_SIZE - 1)  
mov edx, fs:[TEB.DeallocationStack]
```

- Читаем актуальные границы стека из TEB (Thread Environment Block).
- Выравниваем ESP на границу страницы для проверки расширения.

Шаг 5: Проверка переключения стека (Fiber)

```
asm
```

```
.if ([Ebx].Stack.Range.Base != Eax) && ([Ebx].Stack.Range.Limit != Edx)  
    mov [ebx].Stack.Range.Base, eax  
    mov [ebx].Stack.Range.Limit, edx  
.endif
```

- Если стек изменился (например, при переключении Fiber) — обновляем информацию.

- **Важно:** При Fiber-переключении адрес стека меняется, а TLS остается тот же.

Шаг 6: Проверка расширения стека

asm

```
.if ([Ebx].Stack.Range.Base == Eax) && ([Ebx].Stack.Range.Limit == Edx)
    .if ([Ebx].Stack.Bottom >= Esi) && (Ecx <= Esi)
        mov [ebx].Stack.Bottom, ecx
        test D[ecx], ecx ; Touch
    .endif
.endif
```

- Если стек расширился (выделилась новая страница) — обновляем Bottom .
- test D[ecx], ecx — **touch-страница**: принудительно выделяем физическую память (чтобы избежать page fault позже).

Шаг 7: Возврат на стек потока

asm

```
mov eax, [ebx].Stack.Bottom
xchg D fs:[DSTK], eax
mov [ebx].Stack.Current, eax
```

- Загружаем в ESP **актуальный стек потока** (сохраненный в Bottom).
- Обновляем Current для последующих проверок.

Шаг 8: Строим IFR-фрейм на стеке потока

asm

```
pushfd
push cs
push eax ; IP'
cld
```

- Строим фрейм для IRET (возврат из движка).
- push cs — сохраняем текущий сегмент кода.

- `push eax` — сохраняем адрес возврата (IP').

Шаг 9: Сохраняем регистры (RGP)

```
asm
```

```
pushad
mov ebp, esp
mov [ebx].State, esp
```

- Сохраняем все регистры на стеке потока (в RGP).
- `ebp` указывает на начало фрейма TST .
- `State` хранит указатель на текущее состояние.

3. Управление стеком при генерации кода: TrRun

Код из DYE.asm:

```
asm
```

```
TrRun:
```

```
; 1. Сохраняем текущий IP
xchg [ebx].Ip, esi
mov ecx, [ebx].Ia.S1ze
mov [ebx].Ip2, esi
```

```
; 2. Копируем инструкцию в буфер Work
mov edi, [ebx].Work
mov esi, [ebp].Control.rEip
rep movsb
```

```
; 3. Добавляем JMP обратно в движок
mov al, 0E9H ; JMP near
stosb
lea eax, TrGate - 4
sub eax, edi
stosd
```

```
; 4. Переходим к следующему шагу
```

```
; 4. Переключаемся в кон-режим  
mov edx, [ebx].Work  
mov [ebp].Control.rEip, edx  
xor [ebx].Flags, (1 shl TF_RUN) or (1 shl TF_DYE)
```

TrRet:

```
; 5. Выход из движка (IRET)  
push [ebp].Control.rEFlags  
mov eax, [ebp].Rgp.rEsp  
popfd  
mov [ebp].Control.rEip[4], eax
```

TrTrap:

```
popad  
pop D fs:[DPIP]  
pop esp  
jmp D fs:[DPIP]
```

TrGate endp

Что происходит:

Шаг 1: Копируем инструкцию в буфер

asm

```
mov edi, [ebx].Work  
mov esi, [ebp].Control.rEip  
rep movsb
```

- Буфер `Work` (32 байта) выделен в секции `.code WORK`.
- Копируем оригинальную инструкцию (длина `la.Size`).

Шаг 2: Добавляем JMP TrGate

asm

```
mov al, 0E9H  
stosb  
lea eax, TrGate - 4  
sub eax, edi  
stosd
```

- Добавляем безусловный переход обратно в движок.
- Относительное смещение вычисляется как `TrGate - (edi + 4)`.

Шаг 3: Переключаемся в RUN-режим

asm

```
mov edx, [ebx].Work
mov [ebp].Control.rEip, edx
xor [ebx].Flags, (1 shl TF_RUN) or (1 shl TF_DYE)
```

- Подменяем `rEip` на адрес буфера (инструментированный код).
- Сбрасываем `TF_DYE` (выход из движка) и взводим `TF_RUN` (выполнение кода).

Шаг 4: Выход через IRET

asm

```
push [ebp].Control.rEFlags
mov eax, [ebp].Rgp.rEsp
popfd
mov [ebp].Control.rEip[4], eax
TrTrap:
popad
pop D fs:[DPIP]
pop esp
jmp D fs:[DPIP]
```

- Восстанавливаем флаги, ESP, регистры.
- Передаем управление на инструментированный код (через IRET).
- **Ключевой момент:** IRET загружает `CS:EIP` и `SS:ESP` из фрейма.

4. Управление стеком при системных вызовах

TcPush (сохранение стека в CMS)

asm

TcPush proc uses ebx Cleanup:BOOLEAN

%TLRSR Ebx

; 1. Берем текущий ESP из состояния

mov edx, [ebx].State

mov edx, TST.Rgp.rEsp[edx]

; 2. Очистка IDLE-записей (если включена)

.if Cleanup

mov eax, [ebx].Tcm.Cm

add eax, [ebx].Tcm.I * sizeof(CM)

@@:

.if [Ebx].Tcm.I

mov ecx, [eax].Nid[-sizeof(CM)]

bt ecx, NID_NP

.if Carry? || ([Eax].Stack[-sizeof(CM)] <= Edx)

; Удаляем запись

dec [ebx].Tcm.I

sub eax, sizeof(CM)

jmp @b

.endif

.endif

.endif

; 3. Сохраняем ESP в новой записи

mov [eax].Stack, edx

and [eax].Nid, 0

; 4. Вычисляем адрес стаба

mov edx, [ebx].Tcm.I

shl edx, 3

add edx, [ebx].Tcm.Map

; 5. Расширяем стек TCM

inc [ebx].Tcm.I

ret

TcPush endp

Что сохраняется в CMS:

cpp

```
struct CM {  
    PVOID Ip;    // Оригинальный RET-адрес  
    PVOID Stack; // ESP на момент вызова (из TST.Rgp.rEsp)  
    ULONG Nid;   // Флаги + ID сервиса  
};
```

Зачем сохранять ESP?

- При возврате через TrTcmGate проверяем: не "ушел" ли стек выше.
- Если CME.Stack <= Current_ESP — запись считается "мертвой" и удаляется.

5. Восстановление стека при возврате: TrTcmGate

asm

```
TrTcmGate proc C  
    ; 1. Получаем адрес возврата из стека  
    xchg D[esp], eax  
    pushfd  
    pushad  
    %TLSR Ebx  
  
    ; 2. Вычисляем индекс  
    sub eax, [ebx].Tcm.Map  
    and eax, NOT(7)  
    mov edi, [ebx].Tcm.Cm  
    lea edi, [edi][eax*2] ; +(I*8)*2  
    shr eax, 3 ; Ci  
  
    ; 3. Получаем текущий ESP  
    lea ebp, [esp][sizeof(RGP) + 8]  
  
    ; 4. Очищаем IDLE-записи  
@@:  
    mov ecx, [edi].Nid  
    bt ecx, NID_NT
```

```
.if Carry?
; Восстанавливаем ID сервиса
mov [ebx].Service.I, ecx
bts [ebx].Flags, TF_NR
.endif
.if Eax
sub edi, sizeof(CM)
bt [edi].Nid, NID_NP
.if Carry? || ([Edi].Stack <= Ebp)
dec eax
jmp @b
.endif
.endif

; 5. Сжимаем стек TCM
mov [ebx].Tcm.I, eax

; 6. Восстанавливаем оригинальный IP
mov edx, [edi].Ip
bts [ebx].Flags, TF_CM
mov [ebx].Ip, edx

popad
popfd
pop eax
; jmp TrGate
TrTcmGate endp
```

Проверка валидности стека:

```
asm

bt [edi].Nid, NID_NP ; Проверяем IDLE-маркер
.if Carry? || ([Edi].Stack <= Ebp)
dec eax
jmp @b
.endif
```

Условия удаления:

1. NID_NP (IDLE) — запись помечена как паразитная.
2. [Edi].Stack <= Ebp — стек записи находится **выше** текущего ESP (т.е. уже недействителен).

6. Полная картина управления стеком

Стеки в DYE:

text

Стек потока	
[Инструментированный код]	
[Оригинальные RET-адреса]	
[Локальные переменные]	
↑ ESP (изменяется при выполнении)	
[Теневой стек (fs:[DSTK])]	
[Сохраненные регистры (pushad/pushfd)]	
[Сохраненный контекст движка]	
↑ ESP (при входе в TrGate)	
[TLS (fs:[DTLS])]	
[TCM: стек вызовов CMS]	
[Информация о стеке потока]	
[Состояние задачи TST]	

Поток управления стеком:

text

- 1. [Стек потока] Выполнение кода
 - ↓ (перехват)
- 2. [Стек потока] CALL TrGate (или JMP TrGate)
 - ↓ (вход в движок)
- 3. [Теневой стек] Сохранение контекста (pushad/pushfd)
 - ↓ (xchg fs:[DSTK], esp)
- 4. [Теневой стек] Обновление информации о стеке
 - ↓ (возврат на стек потока)
- 5. [Стек потока] Построение IFR-фрейма
 - ↓ (сохранение регистров в TST)
- 6. [Стек потока] Декодирование инструкции
 - ↓ (генерация кода)
- 7. [Стек потока] IRET на инструментированный код
 - ↓ (выполнение)
- 8. [Стек потока] Выполнение JMP TrGate
 - ↓ (возврат в движок)
- 9. [Теневой стек] Восстановление контекста
 - ↓ (popad/popfd)
- 10. [Стек потока] Продолжение диспетчеризации

Ключевые моменты:

Момент	Что происходит	Почему это важно
Переключение на теневой стек	xchg fs:[DSTK], esp	Движок изолирован от инструментированного кода
Проверка расширения стека	cmp [ebx].Stack.Bottom, esi	Предотвращает page fault при росте стека
Touch-страница	test D[ecx], ecx	Принудительно выделяет физическую память
Сохранение ESP в CMS	mov [eax].Stack, edx	Позволяет удалять "мертвые" записи
Проверка целостности стека	[edi].Stack == Ebp	Автоматическая очистка

проверка валидности стека	[Ebx].Stack <= Ebp	IDLE
IRET-фрейм	pushfd + push cs + push eax	Переключение режимов (DYE ↔ RUN)

7. Особые случаи

7.1 Переключение Fiber (стек изменился)

```
asm

.if ([Ebx].Stack.Range.Base != Eax) && ([Ebx].Stack.Range.Limit != Edx)
    mov [ebx].Stack.Range.Base, eax
    mov [ebx].Stack.Range.Limit, edx
.endif
```

- При переключении Fiber **адрес стека меняется**, но TLS остается тот же.
- Движок автоматически определяет это и обновляет информацию.

7.2 Расширение стека (Guard Page)

```
asm

.if ([Ebx].Stack.Bottom >= Esi) && (Ecx <= Esi)
    mov [ebx].Stack.Bottom, ecx
    test D[ecx], ecx
.endif
```

- При приближении к Guard Page (страница-страж) стек может расшириться.
- Движок отслеживает это и обновляет Bottom .

7.3 Переполнение CMS (CM_LIM = 64)

```
asm

.if Tcm.Limit == Tcm.Index
```

```
.....  
    Fail ; Переполнение стека вызовов  
.endif
```

- Глубина стека вызовов ограничена 64 записями.
- При переполнении движок аварийно завершается (не может трассировать).

Итог: Управление стеком в DYE

Компонент	Назначение	Ключевая операция
Теневой стек	Изоляция движка от инструментированного кода	xchg fs:[DSTK], esp
TLS	Хранение состояния (TCM, стек, флаги)	%TLSR Ebx
CMS	Хранение истории вызовов (с ESP)	mov [eax].Stack, edx
TST (Task State)	Сохранение регистров + IFR-фрейм	mov [ebx].State, esp
IRET	Переключение между DYE и RUN-режимами	pushfd + push cs + push eax
IDLE-очистка	Удаление "мертвых" записей	[Edi].Stack <= Ebp
Touch-страница	Принудительное выделение памяти	test D[ecx], ecx

Это полноценная система управления стеком, которая:

1. Изолирует движок от инструментированного кода (теневой стек).
2. Отслеживает изменения стека (Fiber, расширение).
3. Сохраняет историю вызовов в CMS (с привязкой к ESP).
4. Автоматически очищает "мертвые" записи.

5. Поддерживает переключение между режимами (DYE ↔ RUN).

Хочешь:

1. Разобрать, как это работает при **глубокой рекурсии** (stack overflow)?
2. Посмотреть, как обрабатываются **исключения** (SEH) в контексте стека?
3. Или как движок работает с **Fiber** (переключение контекста)?

1

Отлично! Разберем, как DYE/TCM работает при **глубокой рекурсии** и как обрабатывает **stack overflow**.

1. Проблема глубокой рекурсии в DBI

1.1 Два уровня рекурсии

В DYE есть **два независимых стека вызовов**:

text

1. Стек потока (оригинальный код)	
- Рекурсивные вызовы функций	
- Ограничен размером стека (обычно 1 МБ)	
- Проверяется через Guard Page	
2. Стек TCM (CMS - Call Map Stack)	
- Хранит историю вызовов (оригинальные RET-адреса)	
- Ограничен константой CM_LIM = 64	
- Хранится в TLS (Thread Local Storage)	

Проблема: Эти два стека **не синхронизированы** автоматически. Глубокая рекурсия может переполнить CMS раньше, чем стек потока.

2. Ограничения TCM (CMS)

2.1 Константы из кода

asm

```
NT_LIM equ 64 ; Максимум системных вызовов
CM_LIM equ 64 ; Максимум записей в CMS (стек вызовов)
```

Почему 64?

- Достаточно для большинства приложений (редко встречается рекурсия глубже 64).
- Компактный размер: 64 записи × 16 байт = 1024 байта (помещается в L1-кэш).
- Быстрый доступ: индекс вычисляется за 2-3 инструкции.

2.2 Структура CMS

cpp

```
CM struct
    Ip    PVOID ? ; Оригинальный RET-адрес (4 байта)
    Stack PVOID ? ; ESP на момент вызова (4 байта)
    Nid   ULONG ? ; ID + флаги (4 байта)
    Rsvd  DWORD ? ; Резерв (4 байта)
CM ends ; Итого: 16 байт

; Массив: CM Cm[64] = 1024 байта
```

3. Обработка рекурсии в TCM

3.1 При вызове (TcPush)

asm

```
TcPush proc uses ebx Cleanup:BOOLEAN
```

%TLSR Ebx

; 1. Проверка: не переполнен ли CMS?

mov eax, [ebx].Tcm.I

.if [ebx].Tcm.Limit == Eax

; CMS переполнен!

%ERR R_NPR ; Not Present Return

.endif

; 2. Сохраняем запись

shl eax, 4 ; * sizeof(CM)

add eax, [ebx].Tcm.Cm

assume eax:PCM

; 3. Очистка IDLE (если включена)

.if Cleanup

mov edx, [ebx].State

mov edx, TST.Rgp.rEsp[edx]

@@:

.if [Ebx].Tcm.I

mov ecx, [eax].Nid[-sizeof(CM)]

bt ecx, NID_NP

.if Carry? || ([Eax].Stack[-sizeof(CM)] <= Edx)

; Удаляем IDLE-запись

dec [ebx].Tcm.I

sub eax, sizeof(CM)

jmp @b

.endif

.endif

.endif

; 4. Сохраняем оригинальный RET и ESP

mov ecx, [esp] ; Оригинальный RET-адрес

mov [eax].Ip, ecx

mov [eax].Stack, edx ; Текущий ESP

; 5. Расширяем стек

inc [ebx].Tcm.I

ret

TcPush endp

Что происходит при рекурсии:

text

Исходный код:

```
void recursive(int n) {
    if (n == 0) return;
    recursive(n-1); // ← ВЫЗОВ
}
```

Трассировка TCM:

Вызов #1 (n=5):

```
Tcm.l = 0
Cm[0].Ip = ret_addr_1
Cm[0].Stack = ESP_1
Tcm.l = 1
```

Вызов #2 (n=4):

```
Tcm.l = 1
Cm[1].Ip = ret_addr_2
Cm[1].Stack = ESP_2
Tcm.l = 2
```

...

Вызов #64 (n=...):

```
Tcm.l = 63
Cm[63].Ip = ret_addr_64
Cm[63].Stack = ESP_64
Tcm.l = 64 ← достигли лимита!
```

Вызов #65 (n=...):

```
Tcm.l == Tcm.Limit (64 == 64)
→ %ERR R_NPR (переполнение CMS!)
```

3.2 При возврате (TrTcmGate)

asm

TrTcmGate proc C

```
; 1. Вычисляем индекс
; 1. Вычисляем индекс
```

```
    xchg b[esp], eax
    pushfd
    pushad
    %TLSR Ebx

    sub eax, [ebx].Tcm.Map
    and eax, NOT(7)
    mov edi, [ebx].Tcm.Cm
    lea edi, [edi][eax*2] ; +(I*8)*2
    shr eax, 3 ; Ci

; 2. Проверка: индекс валидный?
    lea ecx, [eax+1]
    .if [Ebx].Tcm.I < Ecx
        %ERR R_NPR ; CMS под стеком (corruption)
    .endif

; 3. Очистка IDLE-записей
    lea ebp, [esp][sizeof(RGP) + 8]
@@:
    mov ecx, [edi].Nid
    bt ecx, NID_NT
    .if Carry?
        ; Восстанавливаем ID сервиса
        mov [ebx].Service.I, ecx
        bts [ebx].Flags, TF_NR
    .endif
    .if Eax
        sub edi, sizeof(CM)
        bt [edi].Nid, NID_NP
        .if Carry? || ([Edi].Stack <= Ebp)
            dec eax ; Удаляем IDLE-запись
            jmp @b
        .endif
    .endif

; 4. Сжимаем стек
    mov [ebx].Tcm.I, eax

; 5. Восстанавливаем оригинальный IP
    mov edx, [edi].Ip
    bts [ebx].Flags, TF_CM
    mov [ebx].Ip, edx
```

```

    popad
    popfd
    pop eax
    ; jmp TrGate
TrTcmGate endp

```

При возврате из рекурсии:

text

Возврат #64 (глубина 64):

```

    Tcm.l = 64
    l = 63 (последняя запись)
    Cm[63].lp = ret_addr_63
    Tcm.l = 63

```

Возврат #63:

```

    Tcm.l = 63
    l = 62
    Cm[62].lp = ret_addr_62
    Tcm.l = 62

```

...

Возврат #1:

```

    Tcm.l = 1
    l = 0
    Cm[0].lp = ret_addr_0
    Tcm.l = 0

```

4. Проблема: Переполнение CMS (Stack Overflow)

4.1 Как это выглядит

asm

; При вызове #65:

TcPush:

```

    mov eax, [ebx].Tcm.l ; eax = 64
    -
    -

```



```
.if [ebx].Tcm.Limit == Eax ; 64 == 64 → TRUE
    %ERR R_NPR      ; Остановка с кодом R_NPR (2)
.endif
```

Код ошибки: R_NPR = 2 (Not Present Return)

4.2 Что происходит при ошибке

asm

```
%ERR macro I
    push I
    int 3      ; Вызов отладчика (Breakpoint)
    lea esp,[esp][4]
endm
```

- Вызывается исключение INT 3 (#BP - Breakpoint).
- Если подключен отладчик — он получает управление.
- Если нет — программа падает (access violation).

5. Механизмы защиты от переполнения

5.1 IDLE-очистка (пассивная сборка мусора)

asm

```
.if Cleanup
    @@:
    .if [Ebx].Tcm.I
        mov ecx, [eax].Nid[-sizeof(CM)]
        bt ecx, NID_NP
        .if Carry? || ([Eax].Stack[-sizeof(CM)] <= Edx)
            dec [ebx].Tcm.I
            sub eax, sizeof(CM)
            jmp @b
        .endif
    .endif
.endif
```

```
.endif
```

Условия удаления:

1. NID_NP — запись помечена как IDLE (например, при NtContinue).
2. [Eax].Stack <= Edx — ESP записи **выше** текущего ESP (стек "ушел").

Пример очистки:

```
cpp

// Ситуация: вызов NtContinue сменил контекст
// В CMS осталась "мертвая" запись

// TcPush с Cleanup=TRUE:
while (Tcm.I > 0) {
    CM* prev = &Tcm.Cm[Tcm.I - 1];
    if (prev->Nid & NID_NP || prev->Stack <= Current_ESP) {
        Tcm.I--; // Удаляем!
    } else {
        break;
    }
}
```

5.2 Проверка валидности стека при возврате

```
asm

.if Eax
    sub edi, sizeof(CM)
    bt [edi].Nid, NID_NP
    .if Carry? || ([Edi].Stack <= Ebp)
        dec eax
        jmp @b
    .endif
.endif
```

- При возврате проверяем: не "ушел" ли стек у предыдущей записи.
- Если стек ушел выше — запись считается мертвой и удаляется.

5.3 Ограничение на глубину рекурсии (CM_LIM=64)

asm

```
.if Tcm.Limit == Tcm.Index
    Fail
.endif
```

- Жесткий лимит в 64 записи.
- Почему не больше? Потому что 64×16 байт = 1024 байта — идеально для L1-кэша.

6. Пример работы с глубокой рекурсией

6.1 Функция Фибоначчи (рекурсивная)

cpp

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

// Вызов: fib(40)

// Глубина рекурсии: ~40 (но может быть до 64)

6.2 Трассировка TCM

text

[Вход в fib(40)]

TcPush: Tcm.I = 0 → Cm[0] = { Ip: ret_main, Stack: ESP_0 } → Tcm.I = 1

[Вход в fib(39)]

TcPush: Tcm.I = 1 → Cm[1] = { Ip: ret_fib40, Stack: ESP_1 } → Tcm.I = 2

...

- - - - -

[Вход в fib(1)] (глубина 40)

TcPush: Tcm.I = 39 → Cm[39] = { Ip: ret_fib2, Stack: ESP_39 } → Tcm.I = 40

[Возврат из fib(1)]

TrTcmGate: I = 39 → восстанавливаем ret_fib2 → Tcm.I = 39

[Возврат из fib(2)]

TrTcmGate: I = 38 → восстанавливаем ret_fib3 → Tcm.I = 38

...

[Возврат в main]

TrTcmGate: I = 0 → восстанавливаем ret_main → Tcm.I = 0

6.3 Что если рекурсия глубже 64?

cpp

```
void deep_recursive(int n) {
    if (n == 0) return;
    deep_recursive(n-1); // Глубина 100+
}
```

// Вызов: deep_recursive(100)

Результат:

text

[Вход #64]

TcPush: Tcm.I = 63 → OK

[Вход #65]

TcPush: Tcm.I = 64 → Tcm.Limit == Tcm.I → %ERR R_NPR → INT 3 → CRASH!

7. Оптимизации для глубокой рекурсии

7.1 Увеличение CM_LIM

asm

; Вместо 64:

CM_LIM equ 128 ; $128 \times 16 = 2048$ байт

Плюсы: Глубокая рекурсия (до 128).

Минусы: Больше кэш-памяти, медленнее.

7.2 Динамическое расширение CMS

cpp

```
// Псевдокод
if (Tcm.I >= Tcm.Limit) {
    // Выделяем новый массив (realloc)
    PCM new_cm = HeapAlloc(..., Tcm.Limit * 2 * sizeof(CM));
    memcpy(new_cm, Tcm.Cm, Tcm.Limit * sizeof(CM));
    Tcm.Cm = new_cm;
    Tcm.Limit *= 2;
}
```

Плюсы: Безопасно для любой рекурсии.

Минусы: Медленнее (heap allocation), сложнее.

7.3 Tail Call Optimization (TCO)

cpp

```
// Вместо рекурсии:
void tail_recursive(int n) {
    if (n == 0) return;
    tail_recursive(n-1); // Хвостовая рекурсия
}
```

// Можно оптимизировать в цикл:

```
void tail_recursive(int n) {
    while (n > 0) {
        n--;
    }
}
```

```
}
```

В TCM: Если распознать хвостовую рекурсию, можно не пушить новую запись в CMS.

8. Полный алгоритм обработки рекурсии

text

```
| Вызов функции (CALL) |
| ↓ |
| [TcPush] |
|   ├── Проверка: Tcm.I < CM_LIM? |
|   |   ├── ДА → продолжаем |
|   |   └── НЕТ → %ERR R_NPR (переполнение CMS) |
|   ├── Очистка IDLE-записей (если Cleanup=TRUE) |
|   ├── Сохранение: Cm[Tcm.I] = { Ip: RET, Stack: ESP } |
|   └── Tcm.I++ |
| ↓ |
| Выполнение функции |
| ↓ |
| Возврат (RET) |
| ↓ |
| [TrTcmGate] |
|   ├── Вычисление индекса: I = (StubAddr - Map) / 8 |
|   ├── Проверка: Tcm.I > I? |
|   |   ├── ДА → продолжаем |
|   |   └── НЕТ → %ERR R_NPR (коррупция CMS) |
|   ├── Очистка IDLE-записей (при возврате) |
|   ├── Восстановление: RET = Cm[I].Ip |
|   ├── Tcm.I = I |
|   └── JMP RET |
```

9. Stack Overflow (переполнение стека потока)

9.1 Как это выглядит

- -

cpp

```

void recursive(int n) {
    char buffer[4096]; // Большой буфер на стеке
    if (n == 0) return;
    recursive(n-1);    // Глубокая рекурсия с большим фреймом
}

```

Проблема: Стек потока (1 МБ) переполняется быстрее, чем CMS (64 записи).

9.2 Обработка в DYE

asm

```

; В TrGate:
; Проверка расширения стека
.if ([Ebx].Stack.Bottom >= Esi) && (Ecx <= Esi)
    mov [ebx].Stack.Bottom, ecx
    test D[ecx], ecx    ; Touch-страница
.endif

```

Touch-страница: Принудительно выделяет физическую память для новой страницы стека.

9.3 Исключение Stack Overflow

Если стек переполняется, Windows генерирует исключение **STATUS_STACK_OVERFLOW** (0xC00000FD).

Обработка в DYE (KiUED):

asm

```

KiUED proc C
    mov esi, EXCEPTION_POINTERS.ExceptionRecord[esp][sizeof(RGP)][4]
    .if [Esi].ExceptionCode == STATUS_STACK_OVERFLOW
        ; Попытка восстановить стек
        %DBG "STACK OVERFLOW at 0x%X", [Esi].ExceptionAddress
        ; Можно попытаться увеличить стек (VirtualAlloc)
    .endif
KiUED endp

```

```
    ; Или аварийно завершить  
    .endif  
KiUED endp
```

10. Итог: Управление рекурсией в DYE

Аспект	Механизм	Ограничение
Глубина рекурсии	CMS (Call Map Stack) в TLS	64 записи (CM_LIM)
Проверка переполнения	cmp Tcm.I, Tcm.Limit	Аварийный останов (R_NPR)
Очистка мусора	IDLE-маркер + проверка ESP	Пассивная (при Push/Pop)
Стек потока	Touch-страница + проверка Bottom	1 МБ (по умолчанию)
Исключения	SEH (KiUED)	STATUS_STACK_OVERFLOW

Ключевая идея:

- CMS — это **теневого стек вызовов**, который живет в TLS.
- Он **не зависит** от стека потока (может быть глубже или мельче).
- При превышении лимита (64) — движок аварийно останавливается.
- IDLE-очистка помогает удалять "мертвые" записи (например, после NtContinue).

Хочешь:

1. Разобрать **обработку исключений SEH** в контексте TCM?
2. Посмотреть, как DYE работает с **Fiber** (легковесные потоки)?
3. Или как **IDLE-очистка** работает на практике (с примерами NtContinue/ NtCallbackReturn)?

TrRestack::


```
mov eax,[ebp].Rgp.rEsp
sub eax,esp
cmp eax,sizeof(TST) + 64
jnb TrCallback
; %DBG "RST %d", Eax
lea ecx,TrGate
bts [ebx].Flags,TF_RST
xchg [ebp].Control.rEip,ecx
mov [ebx].Ip,ecx
jmp TrRet
?
```

Отличный фрагмент! Это **один из самых важных механизмов защиты DYE — рестак (Restack)**. Давай разберем, что здесь происходит и зачем это нужно.

1. Что такое Restack в DYE?

Restack — это механизм **принудительного переключения стека** при нехватке места для работы движка.

Проблема: Движку DYE нужно место на стеке для:

- Сохранения контекста (TST — Task State, ~64 байт)
- Декодирования инструкций
- Генерации кода
- Обработки исключений

Если стек потока **почти заполнен** (мало места), движок не может безопасно работать → нужно переключиться на **запасной стек**.

2. Анализ кода TrRestack

2.1 Полный код

asm

TrRestack::

```
mov eax, [ebp].Rgp.rEsp ; 1. Текущий ESP потока
sub eax, esp            ; 2. Вычисляем свободное место
cmp eax, sizeof(TST) + 64 ; 3. Проверяем: достаточно ли места?
jnb TrCallback          ; 4. Если ДА → продолжаем работу

; 5. Если НЕТ (мало места) → рестак
lea ecx, TrGate          ; Адрес входа в движок
bts [ebx].Flags, TF_RST ; Вводим флаг рестака
xchg [ebp].Control.rEip, ecx; Подмена IP на TrGate
mov [ebx].Ip, ecx         ; Сохраняем IP в TLS
jmp TrRet                ; Выход из движка (IRET)
```

2.2 Пошаговый разбор

Шаг 1: Текущий ESP потока

asm

```
mov eax, [ebp].Rgp.rEsp
```

- `ebp` указывает на `TST` (Task State) на стеке.
- `[ebp].Rgp.rEsp` — это ESP **до входа в движок** (стек потока).
- Загружаем его в `EAX`.

Шаг 2: Вычисляем свободное место

asm

```
sub eax, esp
```

- `esp` — текущий указатель стека (уже внутри движка, ниже).
- `eax` теперь содержит **разницу** = сколько байт осталось до конца стека.

Шаг 3: Проверка

asm

```
cmp eax, sizeof(TST) + 64
jnb TrCallback
```

- `sizeof(TST)` — размер структуры состояния (~64 байта).
- `+64` — запас для безопасности.
- Если `eax >= 64 + 64 = 128` байт → **места достаточно** → переходим в `TrCallback`.
- Если `eax < 128` байт → **места мало** → выполняем рестак.

Шаг 4: Рестак (переключение стека)

asm

```
lea ecx, TrGate
bts [ebx].Flags, TF_RST
xchg [ebp].Control.rEip, ecx
mov [ebx].Ip, ecx
jmp TrRet
```

Что происходит:

1. `lea ecx, TrGate` — загружаем адрес входа в движок.
2. `bts [ebx].Flags, TF_RST` — взводим флаг `TF_RST` (Restack).
3. `xchg [ebp].Control.rEip, ecx` — подменяем IP в IFR-фрейме на `TrGate`.
4. `mov [ebx].Ip, ecx` — сохраняем IP в TLS.
5. `jmp TrRet` — выходим из движка через IRET.

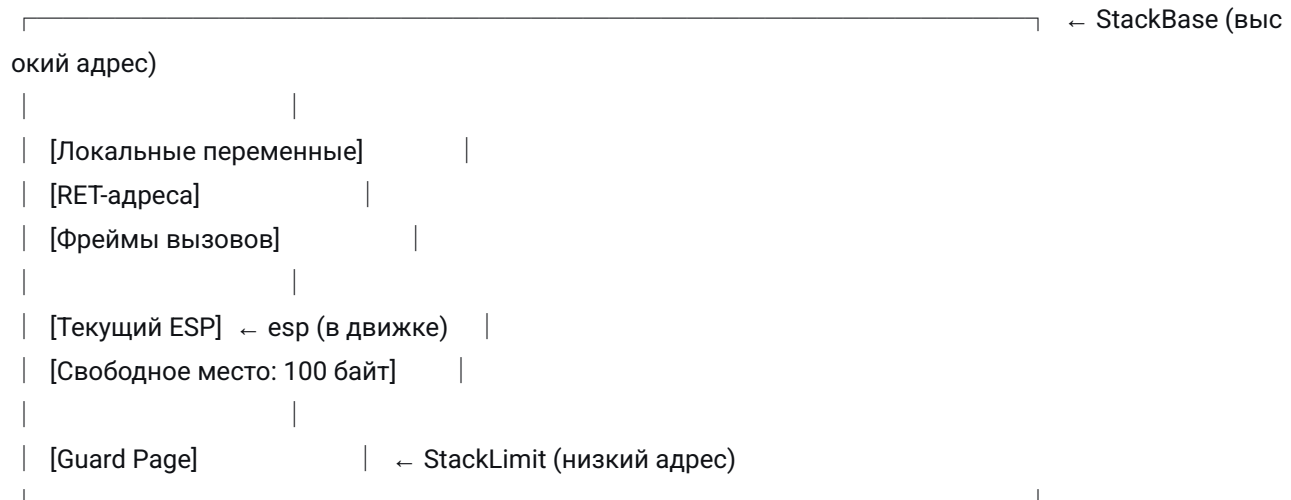
Результат: При выходе через IRET управление перейдет **обратно в TrGate** (вход в движок), но уже с **другим стеком**.

3. Зачем это нужно?

3.1 Стек потока почти заполнен

text

Стек потока (растет вниз):



Проблема: Если ESP приблизился к StackLimit (Guard Page), любая операция `push` или `sub esp` вызовет **Stack Overflow Exception**.

3.2 Что делает Restack?

text

До Restack:



Restack:

1. Сохраняем состояние в TLS
2. Выходим из движка через IRET
3. При входе в TrGate — переключаемся на новый стек

После Restack:



4. Как работает Restack на практике

4.1 Сценарий: Глубокая рекурсия с большим фреймом

cpp

```
void deep_function() {  
    char buffer[1024]; // Большой буфер на стеке  
    // ... делаем что-то  
    deep_function(); // Рекурсия  
}
```

Трассировка:

1. Вход в `deep_function()` (глубина 20):
 - Стек: ESP = 0x1000
 - Свободно: 50 КБ
2. Вход в движок (`TrGate`):
 - ESP = 0x0F00 (чуть ниже)
 - Свободно: 49.5 КБ → ОК
3. Генерация кода, выполнение...
4. Глубокая рекурсия (глубина 100):
 - Стек: ESP = 0x0200
 - Свободно: ~128 байт → **ТРИГГЕР RESTACK!**
5. `TrRestack` :
 - `sub eax, esp` → `eax = 128 - 64 = 64` (мало!)
 - Вводим `TF_RST`
 - Выходим через `IRET` на `TrGate`
6. Вход в `TrGate` (с новым стеком):
 - Стек переключен на запасной

- Свободно: много места

7. Продолжаем выполнение!

4.2 Флаг TF_RST

asm

```
bts [ebx].Flags, TF_RST
```

Где используется:

- При входе в TrGate проверяется TF_RST .
- Если флаг взведен — движок **не пытается** сохранять контекст заново, а **продолжает** с сохраненным состоянием.

В TrGate :

asm

```
TrGate proc C
; ...
bt [ebx].Flags, TF_RST
.if Carry?
; Это рестак — не сохраняем контекст повторно
; Просто используем сохраненный в TLS
btr [ebx].Flags, TF_RST ; Сбрасываем флаг
.else
; Обычный вход — сохраняем контекст
pushad
pushfd
; ...
.endif
; ...
TrGate endp
```

5. Связь с TCM и стеком CMS

5.1 Два стека

Стек	Назначение	Ограничение
Стек потока	Выполнение кода, локальные переменные	1 МБ (по умолчанию)
CMS (TCM)	История вызовов (оригинальные RET-адреса)	64 записи

Restack работает с **первым** — стеком потока.
CMS — отдельно, в TLS, не зависит от стека потока.

5.2 Почему CMS не помогает?

```
cpp

// Ситуация: стек потока почти заполнен
// CMS: 10 записей (много места)
// Стек: ESP почти у Guard Page

// Движок пытается выполнить инструкцию:
mov eax, [esp] ; ❌ Если ESP на Guard Page → Stack Overflow!

// Restack спасает: переключаемся на безопасный стек
```

6. Полный алгоритм Restack

```
text

1. [TrStep] Декодирование инструкции
  ↓
2. [TrCase] Обработка инструкции
  ↓
3. [TrRestack] Проверка свободного места на стеке:
  |— eax = TST.Rgp.rEsp - esp
```

```

└─ if (eax < sizeof(TST) + 64)
|   ↓
|   4. [Restack] Вводим TF_RST
|   ↓
|   5. [Restack] Подмена IP на TrGate
|   ↓
|   6. [TrRet] Выход через IRET
|   ↓
|   7. [TrGate] Вход с TF_RST → переключение стека
|   ↓
|   8. [TrStep] Продолжение выполнения
└─ else
    ↓
    9. [TrCallback] Продолжаем нормально

```

7. Опасности и ограничения

7.1 Бесконечный цикл Restack

Если стек **всегда** почти заполнен, движок будет постоянно рестакаться:

text

TrGate → TrStep → TrRestack → TrGate → TrStep → TrRestack → ...

Решение: Увеличить запасное место или выделить отдельный стек для движка.

7.2 Переключение стека и TCM

При Restack **CMS (TCM)** не меняется — она в TLS.

Но если переключиться на другой стек, нужно обновить TST.Rgp.rEsp :

asm

```

; В TrGate (при TF_RST):
.if [ebx].Flags & TF_RST
    mov eax, [ebx].Stack.Current ; Новый ESP
    mov [ebp].Rgp.rEsp, eax      ; Обновляем в TST
    ; ...

```



```
    btr [ebx].Flags, TF_RST    ; Сбрасываем флаг
#endif
```

7.3 Restack и системные вызовы

При системном вызове (INT 0x2E, SYSENTER) стек переключается на ядро. Restack **не должен** срабатывать в этом случае.

```
asm

; В TrService:
.if [ebx].Flags & TF_RST
    ; Пропускаем Restack — система сама переключит стек
    btr [ebx].Flags, TF_RST
#endif
```

8. Итог: Restack в DYE

Аспект	Описание
Назначение	Защита от Stack Overflow при нехватке места на стеке
Триггер	Свободное место < sizeof(TST) + 64 (~128 байт)
Механизм	Подмена IP на TrGate + взвод TF_RST
Результат	Переключение на безопасный стек (или продолжение)
Связь с TCM	CMS не меняется (хранится в TLS)
Флаг	TF_RST (Restack)

Ключевая идея: Restack — это **аварийный тормоз**, который не дает движку упасть из-за переполнения стека. Вместо краша он переключается на другой стек и продолжает работу.

Хочешь:

1. Разобрать, как работает **TrCallback** (продолжение после проверки)?
2. Посмотреть, как **теневого стек** используется в Restack?
3. Или как это интегрируется с **SEH** (исключениями Stack Overflow)?

Продолжить с DeepSeek