Перевод статьи «**Detecting Hidden Processes by Hooking the SwapContext Function**»

Определение скрытых процессов методом перехвата функции SwapContext.

kimmo пишет:

Общедоступна утилита Klister, созданная Джоанной Рутковской, которая (утилита) может определять скрытые процессы, проверяя содержимое трех связных списков, управляемых ядром: KiWaitInListHead, KiWaitOutListHead и KiDispatcherReadyListHead. Тем не менее Klister может работать только в ОС Windows 2000 и перенесение (переписывание) кода в ОС Windows 2003/XP является не такой уж тривиальной задачей. Проблема в том, что код планировщика различен между этими версиями ОС и необходимые связные списки отсутствуют. К примеру в ОС Windows 2003/XP присутствуют только два списка KiWaitListHead и KiDispatcherReadyListHead, но они не содержат все потоки, существующие в системе. Я потратил немало времени безуспешно пытаясь выяснить в чем можно «подловить» систему. В конце концов я решил пойти в другом направлении. Jamie Butler упоминал в своем выступлении, посвященном прямому управлению объектами ядра (Direct Kernel Object Manipulation , DKOM) на конференции Black Hat в Европе в 2004 году, о том, что в теории один из способов обнаружить скрытый процесс – перехватить функцию SwapContext в модуле ntoskrn.exe. Эта функция занимается переключением контекстов между потоками. Указатели на структуры типа _KTHREAD двух потоков передаются посредством регистров ESI и EDI. Если мы перехватим эту функцию, то сможем получить каждый поток, выполняющийся в системе. Я решил проверить, на сколько осуществимо это решение. Я сделал прототип драйвера, который перехватывает функцию SwapContext, используя обходной метод и собирает идентификаторы процессов и потоков, а также имена их файлов. Исходный код содержится в файле swapcontext_hook.zip, который доступен в моем хранилище. Я создал только драйвер, который, используя функцию DbgPrint, выводит все собранные данные, т.е. устанавливать его в систему вам придется собственноручно (я для этого использовал INSTDRV.EXE доступного из хранилища Hoglund), а также подключить отладчик, или использовать утилиту DbgView от Sysinternals, для получения вывода. Я протестировал драйвер на ОС Windows XP SP1/SP1A и ОС Windows 2003, он работал стабильно и производительность была нормальной. Однако, учитывая то, что мы читаем данные напрямую из внутренних структур ядра, существует возможность что этот код вызовет «синий экран смерти» у вас на компьютере, так что будьте осторожны.
Код драйвера вроде бы хорошо документирован, поэтому если вас интересуют детали, то обращайтесь к коду. Основная идея функционирования драйвера следующая.

Сперва драйверу необходимо узнать адрес функции SwapContext. Это узнается путем сканирования известных расположений памяти, проверяя на наличие специальной сигнатуры, которая содержит первые 20 байт функции. Если сигнатура не была найдена, то будет просканирована все адресное пространство ntoskrnl.exe. За все эти действия отвечает функция FindSwapContextAddress.

Как только у нас будет адрес функции SwapContext, мы ее перехватим. Мы используем обходной метод перехвата, который детально описан Хантом и Брюбейкером (Hunt and Brubacher) в их статье "Обход: Бинарный перехват функций Win32"("Detours: Binary Interception of Win32 Functions"). За перехват в нашем случае отвечает функция InstallSwapContextHook. Основная идея метода состоит в том, чтобы как минимум первые

5 байтов перехватываемой функции заменить командой JMP rel32, которая перенаправит поток выполнения к нашей обходной функции. Количество заменяемых байт зависит от первых нескольких инструкций. Поскольку и машинные инструкции и их длины отличаются в ОС Windows XP и ОС Windows 2003, мне пришлось сделать замену динамической. К примеру, в ОС Windows XP нам необходимо заменить первые 7 байт, а в ОС Windows 2003, первые 6 байт. Чтобы определить длины инструкций, я использовал дизассемблерный движок XDE v1.01 , написанный Z0MBie. Первые 5 байт содержат инструкцию JMP rel32, остальные заполнены пустыми операциями (NOP). Функция InstallSwapContextHook также создает трамплин, который содержит инструкции, которые мы заменили и переход на оставшуюся часть исходной функции SwapContext. Моя обходная функция выглядит так:

```
void __declspec(naked) DetourFunction()
{
    __asm {
        // Сохранить параметры, которые будут изменены.
        pushad
        pushfd
        // Отменить прерывания. Перейти в однопроцессорный режим.
        cli
        // Регистр EDI содержит выполняющийся поток, который будет
переключен.
        push edi
        call ProcessData
        // Регистр ESI содержит ожидающий поток, которому будет передано
управление.
        push esi
        call ProcessData
        // Включить прерывания.
        sti
        // Восстановить сохраненное состояние.
        popfd
        popad

        // Перейти на «трамплин».
        jmp dword ptr pTrampoline
    }
}
```

ProcessData – это функция, которая достает необходимые данные из структур типа _KTHREAD, _ETHREAD и _EPROCESS и сохраняет эти данные в отдельной цепочной хеш-таблице. Я использую адрес виртуальной памяти потока как ключ в хеш-таблице (первоначально использовался идентификатор потока для этой цели, однако теоретически можно изменить идентификатор вредоносного потока на такой же, как у безопасного) и вставка потока происходит всего 1 раз за его жизненный цикл. Поскольку я использую адрес в памяти потока как ключ, я должен позаботится о том, чтобы запись о потоке удалялась из хеш-таблицы при завершении потока, т.к. новый поток может быть создан по тому же виртуальному адресу. Когда поток завершается, он сообщает об этом, устанавливая флаг Terminated в поле CrossThreadFlags, которое является частью структуры _ETHREAD. Функция ProcessData выглядит следующим образом:

```
void __stdcall ProcessData(DWORD *pEthread)
{
    // ЗАМЕЧАНИЕ: WinDbg использует смещения в байтах, мы используем двойные
слова
    DWORD *pEprocess = (DWORD *)*(pEthread + offsets.threadsProcess);
    DWORD *pCid = (DWORD *)(pEthread+offsets.CID);
    DWORD key;
    DATA data;
```

```
    // FIXME: Поток может быть скрыт, установив поле threadsProcess или CID
    // в значение NULL!
    if (pEprocess != NULL && pCid != NULL)
    {
        key = (DWORD)pEthread;
        data.processID = *pCid;
        data.threadID = *(pCid + 0x1);
        data.imageName = (BYTE *)(pEprocess+offsets.imageFilename);

        // Поток завершен, поэтому убираем его из хеш-таблицы.
        if (*(pEthread + offsets.crossThreadFlags) & 1)
        {
            Remove(key, pHashTable);
        }
        else
        {
            Insert(key, &data, pHashTable);
        }
    }
}
```

Хеш-таблица содержится в неперемещаемой области памяти, по той причине, что мы работаем на уровне IRQL = DISPATCH_LEVEL.

И, наконец, когда драйвер будет останавливаться, он уберет перехват, выведет содержимое хеш-таблицы посредством вызова DbgPrint, освободит все использовавшиеся ресурсы. Так что все довольно просто.

Я провел незначительное тестирование в результате которого этот метод оказался способным обнаружить все процессы скрытые любыми современными руткитами. Влияние на производительность незаметно при обычном использовании ПК. Поскольку я убираю потоки из хеш-таблицы по мере из завершения, этот драйвер отобразит только те процессы, которые будут иметь активные потоки. Однако, если мы не будем убирать их, и будем использовать уникальные ключи, то получим полный список всех процессов, которые запускались в системе.

Есть ли способ «обмануть» этот метод? Да, конечно, вы можете пропатчить ядро, убрать перехват и т.д.