сишные трюки
(18h выпуск)

крис касперски ака мыщъх, a.k.a. souriz, a.k.a. nezumi, no-email

долгое время мы витали вокруг чистого ANSI C, без реверансов в сторону нестандартных расширений от различных производителей, которых развелось столько, что игнорировать их, все равно, что добывать огонь трением и писать гусиными перьями, поэтому наш сегодняшний выпуск повещен весьма щепетильной теме интимных взаимоотношений Си с платформой .NET и управляемым (managed) кодом, а любовный треугольник, как известно — самая нестойкая конструкция

трюк #1 – управляемый код на Си

Официально платформа .NET "крышует" C#, F#, Visual Basic и некоторые другие языки, в перечень которых Си, увы, не входит, однако, последние версии компилятора Microsoft Visual C++ поддерживают возможность трансляции программ в управляемый байт-код (по "научному" называемый MSCIL – Microsoft Common Intermediate Language – Общий Промежуточный Язык от Microsoft, но это слишком длинно и заумно, так что мы ограничимся термином "байт-код").

Если сделать небольшой пируэт хвостом, то можно писать Си программы на плюсах, транслируя их в байт-код. Конечно, "чистого" Си мы все равно не получим, однако, по крайней мере, обретем возможность вызывать функции стандартной библиотеки libc, "химичить" с указателями и т. д. Естественно, в силу строгой типизации языка Си++ придется ругаться матом (нецензурным кастингом), впрочем, об этом мы уже говорили в #09h выпуске "трюков".

Чтобы заставить приплюснутый компилятор генерировать байт-код, достаточно воткнуть в начало программы "using namespace System;" и добавить к командной строке ключ "/CLR", пример использования которого приведен ниже:

```
#include <stdio.h>
using namespace System;       // <- используем пространство имен System (из .NET)

void main()
{
        printf("hello, nezumi!\n");
}
```

Листинг 1 hello.cpp – программа на Си++, подготовленная к трансляции в управляемый код, и вызывающая функции стандартной библиотеки языка Си

Трансляция листинга 1 в исполняемый файл из командной строки осуществляется следующим образом:

```
$cl.exe /CLR hello.cpp
```

Листинг 2 трансляция Си++ программы в управляемый код

Если все сделано правильно, на диске образуется файл hello.exe, готовый к непосредственному исполнению и победоносно выводящий "hello, nezumi!" на экран.

трюк #2 –управляемый код и переполняющиеся буфера

Продвигая управляемый код на рынок, Microsoft неустанно перечисляла его преимущества: а) более высокую производительность на чисто вычислительных задачах; б) решение проблемы переполняющихся буферов; в) наличие автоматического сборщика мусора, предотвращающего утечки памяти.

Что касается производительности, то первые версии .NET'а _действительно_ обгоняли Си/Си++ программы в некоторых тестах за счет более компактной структуры байт-кода и динамической оптимизации при трансляции в память. Но уже начиная с .NET 2, производительность байт-кода заметно упала и положение спасает только то, что байт-код способен без перекомпиляции исполняться на процессорах разных типов (x86, x86-64, IA64), используя их преимущества, чего не может чистый машинный код.

А вот контроль за буферами и сборка мусора реально работают только в C# программах (да и то не без оговорок). "Управляемый" код, полученный путем трансляции Си++ программы, наследует все худшие черты языка Си, что мы сейчас и продемонстрируем на примере умышленного переполнения буфера:

```
#include <stdio.h>
#include <string.h>
using namespace System;

void main()
{
        char buf0[0x6]; char buf1[0x6]; char buf2[0x6];
        printf("enter str0  :");gets(buf0);
        printf("enter str1  :");gets(buf1);
        printf("enter str2  :")123;gets(buf2);
        printf("your str is :%s,%s,%s\n",buf0,buf1,buf2);
}
```

Листинг 3 программа, подготовленная к трансляции в управляемый код и допускающая переполнение буфера

Компилируем написанную программу в управляемый код с помощью ключа /CLR и смотрим: сможет ли она справится с ошибкой переполнения или нет. Мы имеем три массива по 06h байт каждый, куда вводим строки длинной в 09h байт (ес-но, эта величина выбрана произвольно).

Результат не заставляет себя ждать:

```
$hello-over.exe
enter str0  :111111111
enter str1  :222222222
enter str2  :333333333
your str is :111111112222222233333333,22222222333333333,333333333
```

Листинг 4 переполнение строковых буферов

Как видно, в buf0 "магическим" образом попали все три строки, в buf1 – вторая и третья строка, buf2 – выглядит неповрежденным, но затирает находящиеся за ним данные (которых в данном случае нет). В общем, все происходит как и следовало ожидать. Буфера последовательно размещаются в памяти и переполнение одного из них воздействует на последующие, хотя, в защиту управляемого кода следует отнести невозможность подмены адреса возврата из функции, а точнее нетривиальной этой операции, поскольку, архитектура виртуальной машины (с учетом компиляции части кода в память) чрезвычайно запутана и реализовать целенаправленную атаку с захватом управления намного сложнее, так что какой-то смысл в управляемой коде все-таки есть, однако, утечки памяти – это кошмар. Управляемый код, не обремененный искусственным интеллектом, не может отличить ситуацию "выделил память и забыл освободить" от "выделил и решил (пока) не использовать". Сборщик мусора реально "отлавливает" лишь небольшую часть ошибок, когда указатель на динамическую память присваивается локальной переменной функции и "погибает" вместе с ней при закрытии стекового фрейма, но стоит функции перед выходом передать этот указатель кому-то еще или сохранить его в глобальной переменной — как все! Сборщик мусора его не тронет.

трюк #3 – смесь управляемого и неуправляемого кодов

Приложения, критические к производительности, а так же программы, взаимодействующие с "внешним" миром (например, оборудованием) пишутся на смеси управляемого и неуправляемого кодов. К счастью, язык C# позволяет вызывать управляемые модули, написанные на Си++, из которых в свою очередь можно вызывать "нативные" (native) функции, компилируемые в машинный код.

Формально, виртуальная .NET-машина поддерживает механизм P/Invoke, предназначенный для прямых вызовов нативного кода, но в языках C#/Си++ он реализован не самым лучшим образом и для решения поставленной задачи, приходится совершать большое количество телодвижений. Но мы не боимся трудностей!

Начнем с того, что напишем Си++ программу, предназначенную для компиляции в машинный код. В ней нет ничего сложно за тем исключением, что все "экспортируемые" строки должны быть представлены в формате Unicode:

```
#include "string.h"
#include "nativecode.h"

void native_foo(wchar_t* c, int num)
{
	wchar_t* s = L"hello, this is native code!";
	wcsncpy_s(c, num, s, wcslen(s));
}
```

Листинг 5 nativecode.cpp – Си++ программа, предназначенная для компиляции в машинный код

Тут же создадим заголовочный файл с прототипом функции native_foo(), включаемый в остальные файлы проекта:

```
void native_foo(wchar_t* c, int num);
```

Листинг 6 nativecode.h — заголовочный файл

Теперь пишем Си++ программу, транслируемую в управляемый код и вызывающей нашу нативную функцию native_foo(), что достигается за счет использования конструкции "**ref class CPPClass**":

```
#include "nativecode.h"

using namespace System;

namespace souriz
{
	ref class CPPClass
	{
	public:
		static String^ foo_wrapper()
		{
			wchar_t c[0x69];
			native_foo(c, sizeof(c) / sizeof(c[0]));
			return gcnew String(c);
		}
	};
}
```

Листинг 7 clrcode.cpp — Си++ программа, подготовленная к трансляции в управляемый код и вызывающая нативную функцию native_foo()

Остается только заточить C# программу, вызывающую метод foo_wrapper() из Си++ программы, вызывающей в свою очередь нативную функцию native_foo(), что осуществляется посредством конструкции "**CPPClass.foo_wrapper()**":

```
using System;
using souriz;

namespace nezumi
{
	class Program
	{
		static void Main(string[] args)
		{
			String s = CPPClass.foo_wrapper();
			Console.WriteLine(s);
		}
	}
}
```

Листинг 8 program.cs – программа на C#, вызывающая метод foo_weapper() из управляемого Си++ кода, вызывающий в свою очередь нативную функцию native_foo()

А теперь собираем все это вместе с помощью следующего командного файла:

```
$cl.exe /c /MD nativecode.cpp
$cl.exe /clr /LN /MD clrcode.cpp nativecode.obj
$csc.exe /target:module /addmodule:clrcode.netmodule Program.cs
$link.exe /LTCG /CLRIMAGETYPE:IJW /ENTRY:nezumi.Program.Main  /SUBSYSTEM:CONSOLE
		/ASSEMBLYMODULE:clrcode.netmodule /OUT:mix.exe
		clrcode.obj nativecode.obj program.netmodule
```

Листинг 9 make.bat – командный файл, собирающий все файлы проекта воедино

Если сборка прошла успешно, на диске образуется mix.exe файл, заглянув в который дизассемблером, мы увидим смесь управляемого и неуправляемого кода. Проблема однако в том, что IDA Pro (самый популярный хакерский дизассемблер) не поддерживает смешенный режим и показывает либо машинный, либо управляемый код, в зависимости от настроек, выбранных еще на стадии загрузки исследуемого файла в базу, а потому написание "смешанных" программ — хороший защитный прием, существенно затрудняющий анализ (большинство начинающих хакеров вообще не увидят машинный код в .NET сборке и будут очень долго гадать как же все это работает). Отладка "смешенных" программ, не содержащих отладочной информации (по умолчанию она не генерируется), это вообще какой-то кошмар, серьезно напрягающий даже гуру.