Антиотладочные приёмы для 8-битных микроконтроллеров AVR

Николай Баландин (Москва)

Статья посвящена защите авторского кода в микроконтроллерах AVR от нежелательного исследования.

Введение

Статья посвящена актуальной теме – защите программ от взлома и нелегального исследования. С 1 января 2008 г. вступил в силу новый закон об авторском праве вместе с поправками в УК РФ (ужесточено наказание за совершение преступлений, предусмотренных частью третьей статьи 180 УК РФ «Нарушение авторских и смежных прав»), но ситуация в сфере защиты авторских прав на программное обеспечение была и остаётся непростой.

В наибольшей степени это касается программных продуктов, произведённых отечественными программистами, которые работают в небольших организациях или, что чаще всего, в одиночку. За их спиной не стоят мощные юридические департаменты фирм – гигантов программного обеспечения, которые могут вести многомесячные судебные тяжбы и выигрывать их. Российские программисты не имели и не имеют до настоящего времени практической возможности получить гарантию авторских прав на разработанное ими ПО.

Хотелось бы подчеркнуть, что речь идёт не только о программах с уникальными алгоритмами обработки данных в микроконтроллере, но даже о несложных «прошивках». Формально и самая сложная программа, и простейшая должны быть одинаково защищены как интеллектуальная собственность программиста. В отличие от труда писателя, где издание небольшого рассказа даёт некоторую гарантию авторских прав, в программировании всё наоборот. В связи с этим мы рассмотрим некоторые методы защиты интеллектуальной собственности программиста на примере модификации программ для микроконтроллеров AVR Atmega168. Все примеры были скомпилированы и дизассемблированы в программе AVR Studio, версия 4.15.

Противодействие автоматическим и интерактивным дизассемблерам

Автоматический дизассемблер анализирует исходный hex-файл с прошивкой микроконтроллера и формирует по нему ассемблерный листинг программы. Примером автоматического дизассемблера является программа *disavr* (www.atmel.ru). Интерактивный дизассемблер включает в себя мощный пользовательский интерфейс, который строит граф передачи управления, позволяет изменять имена меток, функций и переменных, добавлять комментарии к листингу программы, искать последовательности символов в тексте листинга или последовательности байт в памяти.

На сегодняшний день одним из лучших интерактивных дизассемблеров является *IDA PRO* (по умолчанию он не поддерживает микроконтроллеры AVR, для этого необходимо установить дополнительный модуль). Основными его преимуществами являются возможности построения графа передачи управления, повторного дизассемблирования участков кода в директивы *.DW* или *.DD* и наоборот (констант в памяти в инструкции), замены выбранных инструкций на новые (редактирование кода), нахождение перекрёстных ссылок. Наличие функции построения графа сильно упрощает понимание общей структуры построения алгоритма программы, однако, как будет показано ниже, можно эффективно бороться с механизмом автоматического построения графа.

«Запутывание» алгоритма программы

Этот метод является эффективным как для автоматических дизассемблеров, так и для интерактивных. Во-первых, можно вставлять команды условного ветвления, зная, что это условие будет ложным, т.е. управление не передаётся. Можно, например, обнулить один из регистров, затем, через несколько десятков строк кода, вставить инструкцию перехода по условию, что содержимое регистра не равно нулю. Мы точно знаем, что этот переход никогда не выполнится, а вот интерактивный дизассемблер добавит лишнюю ветвь в графе, что, скорее всего, сделает и пользователь, анализирующий программу вручную с автоматическим дизассемблером. При достаточно большом числе таких ветвей пользователю придётся пошагово выполнять программу в отладчике и проверять истинность всех этих условий, что приведёт к большим затратам времени.

Модификацией этого метода является выполнение инструкций *JMP* через *PUSH* и *RET*, *CALL* через *PUSH* и *JMP*, что позволяет «сбить с толку» интерактивный дизассемблер, который умеет отделять обычные метки в программе от точек входа в функцию (функции в графе оформляются в виде отдельных блоков кода). Если проводить такие замены *CALL* и *JMP*, то можно, например, сделать так, что в графе локальная метка внутри функции будет определена как отдельная функция в графе, при этом может быть потеряна точка возврата.

Вся эта путаница пусть и медленно, но всё же раскрывается при внимательном статическом анализе кода. Чтобы сделать невозможным статическое исследование кода, необходимо использовать переходы и вызовы функций по динамически изменяемым адресам, т.е. применить инструкции *ICALL* и *IJMP* (*indirect call to Z*, *indirect jump to Z*). Значение, помещаемое в пару регистров *Z* (адрес), следует задавать не константой, а вычислять при выполнении кода (при этом мы должны точно знать, что получится, иначе это приведёт к некорректной работе программы). Тогда ни автоматический, ни интерактивный дизассемблеры не смогут определить точку входа или возврата из процедуры, а пользователь сможет определить это лишь при пошаговой трассировке под отладчиком.

Выше мы рассмотрели «классические» методы защиты программы от исследования. Их недостаток состоит в

том, что дизассемблер будет выдавать правильный листинг программы. Однако существуют приёмы, позволяющие искажать выходной листинг кода, т.е. при дизассемблировании в листинге появятся фиктивные инструкции, которые мы не использовали, а наши инструкции «исчезнут». Остановимся на этом подробнее.

Перекрывающиеся инструкции

Напомним, что адресация флэш-памяти микроконтроллеров AVR разнесена по словам (2 байта), а не по байтам; минимальная длина инструкции составляет 2 байта; программный счётчик указывает на слова, а не байты. Целью намеренного искажения будут инструкции, состоящие из 2 слов (4 байтов). В таких инструкциях первые 2 байта являются кодом операции, а последние 2 байта – операндом (например, адресом безусловного перехода или адресом вызываемой функции).

Рассмотрим такую инструкцию на примере программы для микроконтроллера Atmega168; *JMP k* – безусловный переход на адрес *k* (*PC=k*, на флаги процессора не влияет, выполняется за три такта); *k* – адрес, по которому осуществляется переход. В шестнадцатеричном виде формат этой команды будет такой:

```
JMP             k
0x940C  0xXXXX
```

Так, например, инструкция *JMP Reset* (*Reset=0x00*, вектор сброса) будет иметь код *0x940C0000*. Или в таком фрагменте программы:

```
        JMP sub_1; переход на sub_1
...
.ORG 0x123
sub_1: ; точка входа расположена
по адресу 0x123
        NOP
;JMP sub_1 будет иметь код
0x940C0123. (0x0123 - адрес пере-
хода)
```

Эту особенность 4-байтовых инструкций можно использовать, чтобы «обмануть» дизассемблер и запутать того, кто попытается исследовать алгоритм работы программы.

Суть перекрывающихся инструкций состоит в том, что вместо адреса мы подставляем код операции любой другой инструкции, например, *LDI*, *ADD*, *SEI* или *CALL*. Далее, чтобы выполнился не переход *JMP*, а скрытая инструкция, надо сделать так, чтобы процессор начал считывать команду не с первого слова (по которому расположен код операции *JMP*), а со второго (которое представляет собой скрытую инструкцию). Заставить процессор сделать это довольно просто – необходимо лишь изменить PC (Program counter, счётчик команд) так, чтобы после завершения выполнения текущей инструкции он стал указывать на начало скрытой в *JMP* инструкции (использование инструкций вызова и ветвления).

Естественно, в нашей программе нам придётся задать операнд (код операции скрытой команды) не мнемоническим обозначением скрытой инструкции, а указать его числовой код. В простейшем случае можно получить скрытую инструкцию *NOP* следующим образом (JMP 0x00 имеет код 0x940C0000):

```
JMP 0x51; PC = 0x51
.ORG 0x50; JMP расположен по ад-
ресу 0x50, а переход идет на
0x51, т.е. в середину JMP
JMP 0x00; переход на третий байт
JMP 0x00, будет считан код опера-
ции 0x0000 - NOP
LDI R15, 1; это выполнится после
NOP, перехода на 0x0000 не будет.
```

При выполнении такого фрагмента кода произойдёт сначала переход, а затем *NOP*, скрытый в *JMP 0x00*, после этого в регистр R15 будет помещена единица, и не произойдет никакого перехода на вектор сброса (*JMP 0x00*).

Это – простейший случай; вместо *NOP* можно поставить что-нибудь более сложное, например, *ADD R15,R16*, тогда «переход» будет выглядеть как *JMP 0x0EF0*, и вместо перехода выполнится *R15=R15+R16*. Вообще говоря, передать управление на скрытую инструкцию можно не только через *JMP 0x51*, но и через *RJMP*, *CALL*, *ICALL*, *RCALL* и т.п.

Естественно, при обработке hex-файла с прошивкой контроллера дизассемблер «не поймёт», что выполняется не *JMP*, а другая инструкция, и выдаст *JMP*. Выяснить, что происходит на самом деле, можно только при помощи отладчика, пошагово выполняя программу, что, как правило, бывает просто физически невозможно в программах, содержащих более 1000 строк кода.

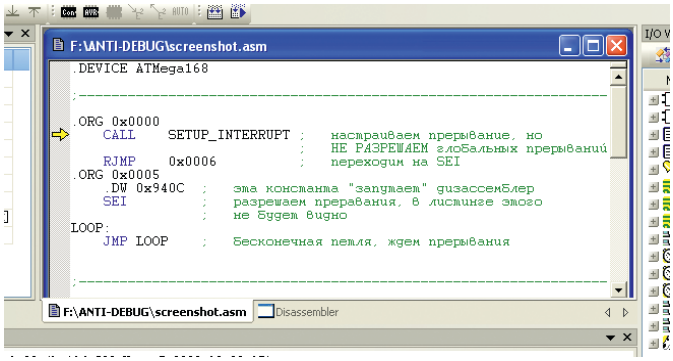

Рис. 1. Исходный текст программы со скрытой инструкцией SEI

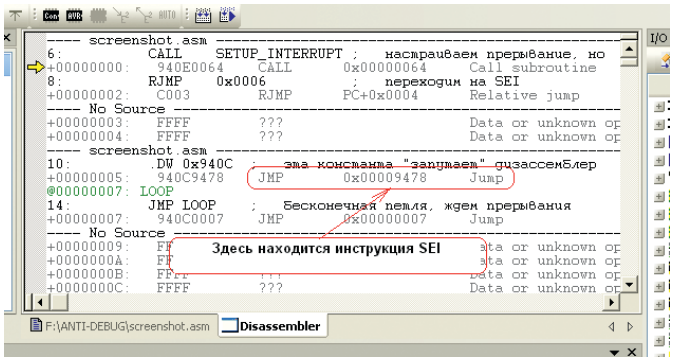

Рис. 2. Искажение исходного текста в дизассемблере

Однако существует более эффективный способ, который позволяет скрывать инструкции. Суть его состоит в том, что перед кодом, который необходимо «спрятать», ставится директива *.DW* (определить константу во флэш-памяти):

```
        JMP 0x51; переходим на JMP 0x20
.ORG 0x50
.DW 0x940C; код операции JMP
        JMP 0x20; сюда будет передано управление
```

Последняя инструкция будет скрыта при дизассемблировании программы, а получится:

```
JMP 0x940C; jump (по несуществующему адресу)
0020    ??? data or unknown opcode
(сообщение дизассемблера в AVR Studio)
```

Видно, что дизассемблер зашел в тупик, попытавшись расшифровать инструкцию с кодом *0x0020*.

Второй пример:

```
        JMP 0x51; переходим на SEI
.ORG 0x50
.DW 0x940C; код операции JMP
        SEI; разрешить глобальные прерывания
Листинг:
0x0050: JMP 0x9478 0x00009478; переход по несуществующему адресу.
```

Третий пример кода и выходной листинг дизассемблера приведены на рисунках 1 и 2.

Серийный номер микроконтроллера

Все вышеописанные приёмы могут помочь защитить вашу программу от исследования, но не от копирования прошивки микроконтроллера. Одним из методов такой защиты является привязка программы к серийному номеру микроконтроллера (который появился в новых микроконтроллерах, например AVR Xmega). Каждый выпускаемый МК получает на заводе уникальный серийный номер, который мы можем использовать для защиты программного кода прошивки микроконтроллера от незаконного копирования.

Известным методом защиты программы от копирования и исследования при наличии серийного номера МК является шифрование основного куска кода программы и добавление расшифровывающего модуля в начало программы. Однако этот приём, распространённый на машинах архитектуры x86, не подойдёт для микроконтроллеров по следующим причинам:

- при каждом запуске программа будет перезаписывать флэш-память, которая имеет ограниченное число циклов стирания-записи;
- перед выключением контроллера необходимо корректное завершение программы, т.е. модуль шифрования должен обратно зашифровать программу, вернув систему в исходное состояние. В противном случае станет доступным исходный код.

Защитить программу при помощи серийного номера микроконтроллера можно при помощи установки контрольных точек с инструкциями *ICALL* или *IJMP*. Рассмотрим это на следующем фрагменте программы:

```
.EQU SerialNumber = 0x55AA; 2 байта из серийного номера нашего контроллера
; имитируем наличие в ATMega168 серийного номера
.EQU RealJmpAddr = 0x100; настоящий адрес перехода где-то в программе
.EQU ModifiedAddr = SerialNumber^RealJmpAddr
; это - зашифрованный адрес перехода, привязанный к серийному номеру контроллера
...
CALL GetSerialNumber; эта подпрограмма читает 2 байта серийного номера из флэш-памяти и помещает их в регистры R30 и R31
LDI R16, LOW( ModifiedAddr )
LDI R17,HIGH( ModifiedAddr )
EOR R30, R16
EOR R31,R17; если программа выполняется в нашем контроллере, то в регистре Z (R30:R31) будет RealJmpAddr
IJMP; переходим по адресу в Z
```

Здесь предполагается, что при включении микроконтроллера программа сначала считывает байты серийного номера из флэш-памяти, вызывая функцию *GetSerialNumber*, которая сохраняет эти два байта в регистрах R30 и R31. Затем программа загружает в регистры R16 и R17 изменённый адрес перехода и выполняет между ними операцию *XOR*. Если программа будет загружена в МК с другим серийным номером, то ветвление произойдёт по неизвестному адресу и программа не будет корректно работать.

Заключение

Независимо от того, работаете ли вы над большим проектом или пишете небольшую демонстрационную программу, стоит подумать о мерах по защите вашего кода. Заранее неизвестно, останется ли эта программа на стадии доработки или найдёт широкое коммерческое применение. Конечно, универсальной защиты не существует. Применив описанные в статье методы защиты, вы не сможете быть на сто процентов уверенными в том, что никто не заимствует ваш код. Однако затраченное на это время будет сопоставимо со временем написания подобной программы.

Литература

1. www.atmel.com/avr.
2. www.atmel.ru.