Cracking '96 - An Introduction, using Realmz

by The Morrin Hackers

edited and with annotations by Mr. Wood

for educational purposes only ;)

Part One:

Cracking Realmz, the Game registration only.

Introduction:

This is a quick tutorial to crack Realmz 3.0.1. It shows basic assembly and some simple cracking ideas in 1996. The Crack is by Morrin Hackers, as is this document. With all due respect to Fantasoft, the creators of Realmz, it is a great game; I cracked it and wrote this document for educational purposes, ie. for other people to learn some of the basics. Thank Fantasoft for creating such an easily hackable product that has given people hours of cracking enjoyment. :)

The Equipment: (Hardware and Software)

Basic Stuff:

- You need a Macintosh, either a 68xxx or PPC will do
- Realmz 3.0.1, available from shareware.com

Hacking/Cracking Stuff:

- ResEdit: (and the Code Viewer for it) The combination of these two is
sometimes known as SuperResEdit. The CodeViewer enables you to view the Code Resource of a file as assembly instructions. ResEdit is simply a resource editor and documentation can be found at Apple's Web site. (Both the documentation for ResEdit and ResEdit itself are free.)
- MacsBug: This is a very good debugging program which can also be found at Apple's web sites, as can documentation for it, although there is some built in. [the current version is 6.5.3]

The Brains: (Wetware)

For all cracking of this type you really need to know some assembly. Some documentation can, again, be found at Apple's Web sites, but the best introduction to it is, imho, "Assembly for cracking" by The Shepherd. It's not easy to get hold of, or wasn't when I got it, but you should be able to obtain a copy via a request in alt.hackintosh, however I have made quick explanations whenever I've dealt with it in this document, so you will be able to learn a bit from just reading this.

[ed note: Besides The Shepherd, mentioned above, there seem to be scant few tutorials on cracking the Mac, a truly sad reflexion on this fascinating subject. The Observer, for one, has written a series of excellent articles describing introducory crack methods. These are documented in "Basic Mac Cracking I - IV", as well as in a cameo bit in the MacHackFAQ v.2.0. However, as this person is very fond of Resorcerer, a $256 program, to do his work with, I feel that a how-to manual using freely available tools is in order. I'm sure owning Resorcerer is very nice and all, and someday, if I can justify buying it, I will, but for now I just can't afford it. Another excellent source is "How to Crack" (also known as "[k]") by The Vassal. This is rather old, as is The Shepherd's info, but even though the tools have changed the methods are still useful.]

Another note to make is: Realmz not PPC native, it is therefore not in PPC assembly, but 68k instead.

The Crack:

Firstly: The secret to all types of hacking is to understand a system, so as then to be able to manipulate it in a way that you want.

1)
So we need to understand the system.

[ed note: This reflects straight back to the comments on assembly above. To crack, one is taking the guts of a program and reworking them to suit your purposes. As such, one must have at least a certain understanding of what is going on in those same guts. Without some basic understanding of algorithms, programming

and assembly language, following the directions of someone else's crack (or worse, using Hack It or some similar patch program to do the work for you), is nothing more than an excercise in paint-by-numbers, or fill in the blanks. No guts, no glory.]

1.0.0) Open realmz. You will see that it brings up a dialog box with a register button at the bottom.
1.0.1) Click on the Register button. You'll see that nothing happens as we have not yet entered anything.
1.0.2) Type in a number and press the button again. So once you've done this we continue to analyze the system. It then brings up a dialog box asking for a name. This *usually* indicates that there is a relationship between the name and the number. (I must admit that in this case no such relationship exists)
1.0.4) You should type in your name and see what it does. It should then give you an error message saying that you've made a mistake.

1.1.0) We'll now try to figure out where in the code all this is happening by using MacsBug.
1.1.1) Press Command-reset (that's the Apple key and the funny triangle key). You should have jumped into MacsBug now, if you have you'll see a big white box and lots of numbers and lines.
1.1.2) Type 'help' then the <return> key to get used to the text interface and then play with it a bit.
1.1.3) Type 'atb GetDialogItemText' [no quotes] then the <return> key. This will set what is called a trap, you will be thrown into MacsBug whenever a GetDialogItemText function is called, note that whenever you want to get text from a dialog, the programmer uses this procedure. 'How do you
know that?' I hear you cry, I know because I program. :) But seriously: Apple have created loads of these and they often create the backbone for cracking with MacsBug. Type 'g' then the <return> key to return to normal. You'll probably notice that everything has slowed down a bit, it's alright, it's just MacsBug.
1.1.4) Why are we doing this? The answer is: we are trying to find out what is going on in the code when reamz gets your registration number. To do this we need to intercept the program when it finds out what we typed, this is exactly what MacsBug is going to do.
1.1.5) Type in anything you want into the registration number and press the register button.
1.1.6) You should have jumped into MacsBug again. This is because Realmz tried to use the GetDialogItemText function. Now's where knowing MacsBug comes in important. You need to know what is

going on. If your totally new to it you'd better skip to the back and read up on MacsBug, if you know a bit read on! Now we are in the program we need to know where, so check out the offset and code resource we're on and write it down an a piece of paper.
1.1.7) Type: 'atc' and then press the <return> key. This will clear all traps. a Trap is a fancy word for what we did earlier, but it's also a descriptive word, we traped the program when it tried to call the GetDialogItemText function. MacsBug calles that a trap, so what we do when we type: 'atc' is clear the trap so we dont accidentally jump back into MacsBug again. This will also speed up everything as MacsBug wil no longer have to check every instruction to see if it's a GetDialogItemText function.
1.1.8) Now we are going to use the trace and step commands. Type 't' to trace, this will make you execute the current instruction and then go to the next one without entering any procedures, there is another similar instruction: the 's' instruction which will take you into a procedure. Basically when you press 't' on a ToolBox Instruction(like the GetDialogItemText function we just cleared the trap for) instruction it will not enter the code(it will jump over the instruction) where the 's' - step command - will jump into the instruction if possible, so in the above example of a toolbox instruction you will enter the code for that toolbox instruction, in stead of jumping over it if you had pressed 's', but you should have pressed 't', so we have traced over it.
1.1.9) Type 't' until you get to a 'BNE' or 'BEQ' instruction. ('t' traces through the program)
1.1.10) You should have noticed that when a 'jsr' instruction was traced over MacsBug took a little longer than normal, why? This is because a JSR instruction is an instuction that actually executes a whole chunk of code. It will, just like a ToolBox function, be stepped over, and also like a toolBox Function you can step into it will the step command. The JSR command will execute a whole series of instuctions, usually this is used to do a specific task, like to draw a circle. One thing else to note is unlike a ToolBox function you can set traps for these because the programmer makes them.
1.1.11) Note down the offset of the BNE or BEQ instuction, and whether the program will branch or not. Thats what a branch statement is for, if a condition is true the program will branch off to another place, else it will continue. MacsBug tells you just above the command-line, whether the program is about to branch or continue, it is this which you will have to note down.
1.1.12) Here's an assembly note: 'BNE' stands for Branch if not equal to, this means that the program checks if the last instruction was equal to zero, if wasn't then it jumps, else it continues. This type of instruction is very commonly used to

check registration numbers and so on. Another very similar instruction is BEQ, this stands for Branch_if_EQual_to, this means that if the last instruction was equal to zero then it will branch, otherwise continue.
1.1.13) The fisrt BEQ or BNE instruction you find should look like this:

```
+024B0 00BA1810   BEQ        'CODE 0009 15AC'+023BA     ;
00BA171A   | 6700 FF08
```

and it should say that the program will not branch. Notice that the instruction above the BEQ is: 'tst.w' you can ignore the '.w' as it refers to what part of a series of bits is being acted upton. But the 'tst' part is interesting as it is a test instruction. In this case it is testing the length of the registration number you just typed in, you should be able to guess this from seeing the instruction above the 'tst' instruction, the toolBox function: StringWidth which gets the length of a string. So we know exactly what is going on.
1.1.14) You should continue to use the trace command until you get given the error message about the registration, then you can type in 'g' to continue. Remember to keep track of what is going on by using paper to note down offsets and the other stuff I told you too earlier.
1.1.15) I thought I'd mention this beacause of what I was saying about JSR instructions: You should have noticed that when you got to offset: '025F4' you find "JSR 'CODE 0008 15AC'+00654" and when you trace over it you find your self entering your name in to a dialog box in realmz. This procedure(CODE 0008 at 00654) is a procedure specially for you to type in your name) - now you can see more exactly how jsr instructions work.
1.1.16) This is just a note that you should find one BNE at 2642 (in code 9). If you want to see roughly what happened and what you should have check out the back: I've got it all written down.

2. We now understand the system, so it's t ime to get our hands dirty with some actual alterations! This is step two in hacking, once we understand a system we manipulate it.

2.0.1) Now we have enough data on where the program goes in these branch statements to start with ResEdit, so open Realmz(the game) in resEdit.
2.0.2) We now need to remember what happened when, and so what you've been writing down is essential. Open the Code set of resources, and then the code 9 resource. Once that is open scroll down to the offset we found the GetDialogItemText.(note that in

the ResEdit code Viewer it is written as: GetItText - This doesn't really matter) You may be wondering why I opened the CODE resource, the answer is: all 68k programs have there code in Code resources, and I'm afraid that I have found this out because I program, but it's more or less obvious if you've hacked much in the past. (not that you have :) Another interesting thing is that: PPC programs don't store there code in a resource at all, but instead they do it in teh dta fork of a file.
2.0.3) Start tracing through the code with your eyes, and check to see that what you wrote down was correct. If we think about it, one of the branch instructions before the error dialog was displayed must have been the check to see if you typed in the right code. Therefore we're going to do some fiddling. But before we do this there is a little problem with the format of ResEdit compared to that of MacsBug, I've documented it at the back because it's quite complicated so if you're unfamiliar with the difference check that out.
2.0.4) Go to the Second branch instruction that you found and change it to the opposite type of instruction(BNE -> BEQ and BEQ -> BNE). Remember the first one, we figured out, was just to check the length of your registration number. To do this you'll have to open the hex editor, after selecting the appropriate hex instruction on the far right of the Code Viewer window. The change is in fact very simple; the BNE instruction is 66xx and the BEQ instruction is 67xx. [Note that selecting an instruction is done in the code viewer, where the disassembled code is placed next to its hex and ascii equivalents. When opening the Hex Editor, this selection is automatically transferred, so changes can be made, and the changes are also automatically sent back]
2.0.5) Save and quit. then test the program. You don't get quite the desired result, but you do find out more about how the program works.
2.0.6) Go through all the branches, one at a time, changing the next and fixing the last one, then testing the program. This is slow but almost always bound to work.
2.0.7) You should find that when you get to BNE at 2642 (in code 9) and no other change is made that whatever happens the game is registered! Yes this is it, a game cracked, and all you had to do was change about a 67 and 66.

So how's that? Realmz cracked in 2 stages. ;-)

The purpose of this is not to crack Realmz as much as to learn how to crack, which I hope this has helped you to do. Thanks to: Mr. Wood for Editing and Thanks to Fantasoft for making the game.

MacsBug Sample Data

Here is my smaple data for the use of MascBug in part 1.1:

```
 All A-Traps actions cleared
Step (over)
  'CODE 0009 15AC'
     +02494 00BA17F4   _GetDialogItemText                    ;
40886948
     +02496 00BA17F6   MOVE.L     #$00E98E9E,-(A7)
     +0249C 00BA17FC   PEA        -$0008(A6)
     +024A0 00BA1800   JSR        'CODE 000F 15AC'+02854
     +024A4 00BA1804   CLR.W      -(A7)
     +024A6 00BA1806   MOVE.L     #$00E98E9E,-(A7)
     +024AC 00BA180C   _StringWidth                          ;
0018D1D0
     +024AE 00BA180E   TST.W      (A7)+
     +024B0 00BA1810   BEQ        'CODE 0009 15AC'+023BA     ;
00BA171A
above Will Not Branch
     +024B4 00BA1814   CMPI.L     #$0000038F,-$0008(A6)
     +024BC 00BA181C   BNE        'CODE 0009 15AC'+025F0     ;
00BA1950
above Will Branch
     +025F0 00BA1950   MOVE.W     #$0001,-(A7)
     +025F4 00BA1954   JSR        'CODE 0008 15AC'+00654
above Gets user Name
     +025F8 00BA1958   MOVE.L     $00E95114,$00E96C48
     +02602 00BA1962   MOVEQ      #$18,D0
     +02604 00BA1964   MOVE.L     $00E96C48,D1
     +0260A 00BA196A   DIVS.L     D0,D1
     +0260E 00BA196E   MOVE.L     D1,$00E96C48
     +02614 00BA1974   MOVEQ      #$18,D0
     +02616 00BA1976   ADD.L      D0,$00E96C48
     +0261C 00BA197C   MOVEQ      #$0F,D0
     +0261E 00BA197E   MULU.L     $00E96C48,D0
     +02626 00BA1986   MOVE.L     D0,$00E96C48
     +0262C 00BA198C   SUBI.L     #$00000100,$00E96C48
     +02636 00BA1996   MOVE.L     -$0008(A6),D0
     +0263A 00BA199A   CMP.L      $00E96C48,D0
     +02640 00BA19A0   ADDQ.L     #$2,A7
     +02642 00BA19A2   BNE        'CODE 0009 15AC'+026F8     ;
00BA1A58
above Will Branch
     +026F8 00BA1A58   MOVE.L     #$00E98E9E,-(A7)
```

```
     +026FE 00BA1A5E   MOVE.L     #$00010003,-(A7)
     +02704 00BA1A64   JSR        'CODE 000F 15AC'+02890
     +02708 00BA1A68   MOVE.L     #$00E98E9E,-(A7)
     +0270E 00BA1A6E   MOVE.L     #$00021772,-(A7)
     +02714 00BA1A74   JSR        'CODE 000F 15AC'+02890
     +02718 00BA1A78   MOVE.L     #$01901770,-(A7)
     +0271E 00BA1A7E   MOVE.L     #$00320096,-(A7)
     +02724 00BA1A84   MOVE.L     #$00E98E9E,-(A7)
     +0272A 00BA1A8A   JSR        'CODE 0005 15AC'+0235A
above Puts up failed Dialog.
```

MacsBug to ResEdit Formating

The Difference in format from MacsBug to ResEdit:

In MacsBug an instruction is shown:

```
OffSet           Instr      Where the branch is going
+024B0 00BA1810   BEQ        'CODE 0009 15AC'+023BA     ;
00BA171A
```

In ResEdit's CodeViewer:

```
         OffSet      Instr                  Where the branch is
going
+00130   000024B0    BEQ     Anon2+$0226   ;000023BA
```

Some Assembly:

```
  Hex Number  |  Instruction  |  What it does

--------------------------------------------------------------------
--------
  66xx xxxx

BNE

(Branch if Not Equal) Branches to the




number specified if the last result was not
```

equal to zero

  67xx xxxx

BEQ

(Branch if EQual) Branches to the instruction

if the result of the last instruction was

equal to zero.

  60xx xxxx

BRA

(BRAnch) Simply branches.

One thing that confuses many peoplel is that often when you have an instruction you get after it a fullstop then an 's' or a '.w' and so on. What this is doing is usually not very important to the cracker (imho) but to those no progarmmer minded: this is telling the computer to only use sertain bits when we jump. the '.s' indicates hwat is called the word length, but anless you want to get neck deep in assembly and programming you can ignore all these bits.

There are many other branch instructions, most of most of then beginning with '6', but you can find those by experimenting with resEdit. Have fun!

Some Help with MacsBug:

MacsBug is very complicated for those unused to it, but it is also one of the most powerful and simple to use programs for those who understand it, kind of like life...

Anyway: Here's the Basics,
1) at the very bottom of the screen is the command line, everything ou type is written here untill you press return at which point it is executed.

2) the box directly above the command line is the program state and contains the current possition in the programs code. You will see four line. The top one indicates the current possition of the program in it's code. If you're running a 68k program you will see something like: 'CODE 9 something' you can ignore the something, but the 'code 9' bit os important. the 'code' bit tells you that you are in a code resource, and the '9' bit tells you thge ID of the resource, in the above case it's CODE resource ID 9.

The three Bottom lines conatin the three next instructions to be executed, Where the top one with the '*' next to it will be executed next.

3) the Boxes on the left give you information about the current program's state and some further info on registers, but you can for now ignore all this.

4) the largest Box is the one above the Program State is the display area, anything that happens is told to you here, if you find yourself unexpectedly launched into macsBug It'll tell you why here. This is also wher you read the help from.

Using MacsBug:

The Basic commands you'll need to know are:

1) Trace: type 't' then the <return> key to trace through some code. This will execute the current instuction and go to the next one IN THE SAME PROCEDURE, so that you wont change your current procedure anless the current insturction will not return to the smae proceudure, in this case it simply take you to the next instruction.
2) Step: type 's' then the <return> key this is exactly the same as the above instruction exept that it will jump to the next instruction whether or not it leads to another proceudre which returns.

3) Go: type 'g' then the <return> to continue with whatever happend before you were thrown into MacsBug.

4) Escape to Finder: type 'es' then the <return> and you will be

put back to the finder just as if you force quit the application.

5) Cleat All Traps: type 'atc' then the <return> to clear al set traps. A trap is just a method to make youself jump into MacsBug whenever a program trys to do something like bring up a dialog box.

Part Two:

The Realmz 3.0.1 Registration Crack

Basic Data

Change Resource:

Type: CODE

Resource Number: 9

Line: 2642

From:

548F 6600 00B4 2F3C

To:

548F 6700 00B4 2F3C

How To:

1) Open Realmz in ResEdit.

2) Open the 'CODE' type resource ID 9.

3) change line 2642 to 548F 6700 00B4 2F3C.

What its doing:

As looked at with ResEdit + CodeViewer, you are changing a resource at

Type: CODE
Resource ID: 9

Module: Anon7
Offset: +3DE from Anon7 or 2642 from the start of the resource

The original line, before it was cracked, was:

| Anon7 Offset | Code Offset | Instruction | Data | Where Its going |
| +03DE | 2642 | BNE | Anon16+$494 | 26F8 |

The line as cracked, is

| Anon7 Offset | Code Offset | Instruction | Data | Where Its going |
| +03DE | 2642 | BEQ | Anon16+$494 | 26F8 |

So what's the difference???

The difference is BNE has been changed to BEQ.

BNE means jump if the last compared data was Not Equal to zero (btw - the last compared data is your password and the password you *should* have typed in). What's going on is that Realmz has just got your password and compared it to what it should be. Realmz then checks to see if they are equal, and jumps to the appropriate place.

So,if you typed in the right reg number, realmz continues to the bit of code where you register the game.

But, if you got it wrong, then it jumps, and puts up the error message etc.

What we did was simply to change the Branch_if_Not_Equal_to command to a Branch_if_EQual_to command, that way it doesn't matter what you typed, it will register the scenario anyway provided you typed in the wrong number.

Part Three:

The Realmz 3.0.1 Scenario Crack

Basic Data:

Change Resource:

Type: CODE

Resource Number: 7

Line: 2C10

From:

69FE 6726 3F3C 004E

To:

69FE 6026 3F3C 004E

How To:

1) Open realmz in ResEdit.

2) Open the 'CODE' type resource ID 7.

3) change line 2C10 to 69FE 6026 3F3C 004E.

What its doing:

As looked at with ResEdit + CodeViewer, you are changing a resource at

Type: CODE
Resource ID: 7
Module: Anon7
Offset: +D7A in Anon7, or 2C12 from the start of the resource

The original line, before it was cracked:

| Anon7 Offset | Code Offset | Instruction | Data | Where Its going |
| +D7A | 2C12 | BEQ.S | Anon7+DA2 | 2C3A |

The line as cracked:

| Anon7 Offset | Code Offset | Instruction | Data | Where Its going |
| +D7A | 2C12 | BRA.S | Anon7+DA2 | 2C3A |

So what's the difference???

The difference is BEQ.S has been changed to BRA.S

BEQ.S means jump if the last compared data was EQual (btw - the last compared data is your password and the password you *should* have typed in). What's going on is that Realmz has just got you password and compared it to what it should be. Realmz then checks to see if they are equal (uses a CMP.L command)

So, if you typed in the right password, they are equal, and Realmz jumps to the bit of code where you register the scenario.

But, if you got it wrong, then it doesn't jump, and instead continues, putting up the error message etc.

What we did was simply to change the Branch_if_EQual_to command to a simple BRAnch command, that way it doesn't matter what you typed, it will register the scenario anyway.