PC Cracking v.2
by: ProZaq

Table of Contents:
- Introduction
- Tools
- The PPC Processor in General
- Using MacsBug in a PPC Environment
- Cracking
- Hints for Using MacsBug in a PPC Environment
- Using a PPC Disassembler
- Using The Hex Editor
- Note About FAT Binary Applications
- End Notes

Introduction
~~~~~~~~~~ ~~

To the best of my knowledge, the only tutorial on cracking software written for the Power PC processor so far is by Dot Com. And in my opinion that file is a complete piece of shit! I'm sorry but someone who doesn't know what assembly or hexadecimals are should not be using a debugger, let alone crack software!

My aim in this tutorial is to introduce the general techniques for cracking PPC software for people who are familiar with the 68k processor structure. I will not go into detail about PPC assembly nor will I give any practical examples. You shouldn't need those! The purpose of this file is simply to explain the techniques used for cracking PPC software.

If you are new to cracking then I strongly recommend that you go through my other files on cracking (The Ultimate Mac Cracking Guide) because (and trust me on this one) PPC cracking does get ugly! You can find them in various issues of HackAddict or you can download them from my homepage: www.geocities.com/area51/rampart/4007/Welcome.html
In those tutorials I describe in great detail what cracking is all about. And the truth is that exactly the same principles apply to PPC cracking. The only problem is that there aren't too many tools available to help you on your quest to crack PPC programs. So you'll have to improvise quite a lot.

Trust me, unless you're a programmer or a practiced cracker you'll have no clue as to what I'm talking about most of the time in this file!

Tools
~~~~~

As always I've keep the price of the tools required down to a minimum ($0) and you shouldn't have any problems finding these programs:

1. MacsBug - The cracker's best friend! There's no need to go around finding "demos" of the other debuggers when g'old MacsBug works just fine!

2. ResEdit - All serious Mac user's best friend! Unfortunately ResEdit is not capable of disassembling the PPC assembly code, so bugger Super ResEdit, any old version will do just fine.

3. Any hex editor capable of reading both data and resource forks - I always use HexEditor by Jim Bumgardner (which is freeware), but I guess the data fork editor extension of ResEdit would do just as well...

4. A PowerPC Disassembler:

- PPCdissasembeler 2.0 by Alain Birtz - This is sorta like the Code editor part of Super ResEdit. The
good part about this particular program is that it displays the meaning of the PPC assembly mnemonics.

- Janus 0.1 by Peter Creath - This can sorta be used to find out information about the a-traps used by
the program. I don't use it much. It has a rather nasty interface, and don't give all that much useful information...

- PEF Viewer v1.0d8 - Apple's own PPC disassembler. It has a nice interface and is the most usable of
them all. Unfortunately not even this one can save any changes to the coding itself.

- MacNosy - Pay for it if you want, use it if you want. It's a lovely tool, but I must say that you can
crack just as well without it.

6. Wetware - You will definitely need your head! So make sure you're in immediate possession of it whilst cracking PPC software!

The PPC Processor in General
~~~~~~~~~~ ~~ ~~ ~~~~~~~~ ~~ ~~ ~~

They claim it's fast, and there's no doubt about that. They claim it's got a lotta registers, well 32 general purpose ones actually (plus all the FPU and 64 bit ones). And they claim it's better than the 68k processor, uhm... whatever. Sure it's got all those registers, but they are general purpose ones. And in practice you only use about 20 of them... Lemme warn you straight away: don't mess with the first three registers (r0, SP and TOC)! They hold information that are essential to the execution of the program such as return tables and lovely things like that.

I guess the biggest changed to get used to is that there are no address or data registers! Any of the registers can be used as either data or address registers. Meaning that it can hold a number or it can serve as a pointer to a location in the memory.
It's up to you to figure out the purpose of the register. Something I noticed rather early on is that I was forced to use the "dm" (display memory) command all the time, just to make sure that the damn register wasn't used as a pointer.

Another thing that shocked me was that all PPC commands are 4 bytes (8 digits) long. Oh well... this isn't the place to talk about the speed of executing 4 digit commands...

I will unfortunately not include a list of the assembly commands and what they mean in this tutorial. I guess it's not too important that you know exactly what each command means, but it might be useful to get a hold of a definition of all the different mnemonics, just in case... I can on the other hand recommend the reference manual to PowerFantasm or some files from Apple regarding programing in PPC assembly.

This on the other hand is important to know: every subroutine starts with the "mflr" command and ends with the "blr" command. Apparently a lot of times there are several "blr" commands within a subroutine (sort of a contradiction of terms huh?). This way the program doesn't use the BRA command to branch to the end of the subroutine but ends it straight away once it's done it's purpose.

I guess this is how they try to make up for the speed loss of the 4 byte long commands ;-). Another PPC assembly command that might be useful to recognize is the "bl" command. This is the common branch command that is know in 68k terms as Branch-To-Subroutine (BSR).

The BRA command has the hexadecimal value 4800xxxx and the NOP command has the value 60000000. So whenever you want to replace a conditional branch command with a BRA command, you change it's leftmost 4 digits to "4800" and keep the rest as it was. For example if you wanted a BEQ command represented by 41820034 to always branch you'd change it to 48000034 (remember not to change the rightmost 4 digits, as that will seriously fuck things up!). And you'd just replace the whole thing by 60000000 if you didn't want it to branch.

Another thing to notice when looking at PPC assembly code is that in simple arithmetic the registers are used in reverse order. For instance, here's an example of a simple addition in 68k:

add.w d1,d0

As you all know the values of d1 and d0 would be added up and stored in d0. In PPC the above example might look something like this:

add r3,r4,r5

However, the execution process is reversed. The above command adds r5 to r4 and stores their sum in r3.

Now then, A-Traps. Well they are known as system vectors/ symbols in PPC programing (but I'll just stick to the expression "a-trap" to be nostalgic). Apparently this is supposed to be a much faster alternative to patching the A-Traps, but for crackers they are a real pain in the ass!

Also if you take a look at the resource fork of a PPC file with ResEdit you will find that all the usual resources are there with the exception of the CODE resource. And if it is there then it's usually just some warning that the program won't run on 68k platforms. So where is the actual code? You guessed it; in the data fork. That's why you need a hex editor if you want to change it.

Using MacsBug in a PPC Environment
~~~~~~~~~~ ~~ ~~ ~~~~~~~~ ~~ ~~ ~~~~~~~~

The PPC part of MacsBug is structured the same way as the 68k part. To the left you have the stack, below it the current application name, and below that you have the registers. In the middle you have the assembly commands, and to the far right you have the command's hexadecimal value.

Unfortunately MacsBug is not developed to deal with PPC as well as it does with 68k assembly, but it is still very usable! The command used to break at a specific a-trap, is the "tvb" (TVector Break) command. Notice, however, that while in 68k it was enough to type "atb modal" to set a break for the ModalDialog a-trap, in PPC you have to type out the whole name of the a-trap. So if you want to be dropped into MacsBug at every ModalDialog a-trap, you have to issue the command "tvb ModalDialog".

The equivalent of "atc" is the "tvc" command.

Another thing that sux is that whenever MacsBug breaks for an a-trap it does not actually break before the trap but breaks at the first command inside the trap. So while in 68k you were dropped inside the program itself at the command branching off to the a-trap's subroutine, in PPC you're dropped into the actual a-trap, and have to get out of it first (remember that all subroutines end with the "blr" command).

Once you're out of the a-trap though, you can continue tracing/ stepping through the code as if you always did while cracking 68k programs.

Cracking
~~~~~~~~

You start off exactly like when you were cracking 68k programs. Issue the "tvb" command just like you would with the "atb" command.

Once you're dropped into MacsBug, remember that you are actually inside the a-trap subroutine so you need to trace through the code until you see the "blr" command. Everything that comes after that belongs to the program you're trying to crack. Remember that the PPC mnemonic for the "BSR" command is "bl". You can step into/ trace over any subroutine just like you could in 68k code using the "s" and the "t" commands.

Once you've found the conditionals you're looking for, the same principles apply in PPC code as they do for 68k code. Let's say that the below code won't branch but it would be "nice" if it would:

01A03244 beq $+0x0024 ; 0x01A03268 | 41820024

The "0x01A03268" part tells us (just like in 68k) that if the conditions would have been met it would have branched to the command that's in the memory at address "01a03268" ("0x" prefix is used to represent hexadecimal numbers). Now then, luckily the Program Counter (a special address register that keeps track of which command is to be executed next) works the same way in PPC assembly as in 68k. Thus, if you want the above command to branch, simply set the Program Counter to 01a03268. You would achieve this by typing "pc=01a03268".

On the other hand, you might be in a situation where you want the conditional not to branch. So simply jump over the command. How? Well since in PPC all commands are 8 digits long it makes things simpler than they are in 68k. If you ever have to jump over a command, type "pc=pc+4" (the current pc address plus 4 will give you the address of the next command in line).

Hints for Using MacsBug in a PPC Environment
~~~~~~~~~~ ~~ ~~ ~~~~~~~~ ~~ ~~ ~~~~~~~~~~ ~~ ~~~ ~~~

A very useful feature of MacsBug is the disassemble command, "il". Since there are no fabulous tools to use for disassembling PPC software, you might as well use the Debugger to do it. So how do you use the "il" command? If you want to disassemble from the current address just type "il". This can be very useful when you're trying to figure out what would happen if a conditional didn't branch.

If you want MacsBug to disassemble the assembly commands from a specific address you'll have to use the "il-c" command. Simply because if you don't MacsBug will disassemble the PPC commands in 68k mode and you'll get some very interesting results.

I also found it useful in some cases to figure out what happens in the subroutine before it hits a specific a-trap (especially when I was cracking software protected with hardware keys). This can very easily be achieved by typing "il-c pc-x" where "x" is any multiple of 4 (remember that every PPC command is 4 bytes, 8 digits, long). And if you recall your hexadecimal rules then hex 10 equals dec 16. So it's very convenient just to disassemble using "x" as multiples of hex 10 (meaning 4 assembly commands at a time).
To clear things up, if you want to disassemble from the previous assembly command you type "il-c pc-4". If you want to disassemble starting from the 4th last command
just use "il-c pc-10". You'll get the hang of it!

Another very useful command in PPC cracking is the break command, "br". If you care to read the help section of MacsBug you'll notice that the "br" command in a PPC environment causes an error. So use the "brp" command instead! Both "br" and "brp" can be cleared with the "brc" command.

You'll find break points to be a very useful alternative method to the "tvb" command. Remember how

when you're using the "tvb" command you are actually dropped inside the a-trap? Well once you find your way out of it, you can simply set a break point at the a-trap's subroutine and clear the TVector break. This way you'll still break every time for the a-trap, but before you actually enter the trap.

A very simple way of using the break command is with the help of the Program Counter. If you want to set a breakpoint at the previous assembly command type "brp pc-4"; if you want a break at the current command type "brp pc"; and if you want a break at the next command in line type "brp pc+4" (useful when you wanna get outa long loops). Rather simple, and saves you a lotta time!

Using a PPC Disassembler
~~~~~~~~~~ ~~ ~~ ~~~~~~~~ ~~

In theory, there's no "real" need for any PPC disassembler when you think about it. I mean, you can find out most of the things you want using MacsBug, and since you can't change the program with PPCdisassembler or PEF Viewer anyway (nor Nosey for that matter), there's not much point in using them. Well except maybe for one thing. And that is that PPCdisassembler tells you what the different assembly mnemonics mean (for example "addi" stands for "add immediate"). So if you're in doubt or despair, run around in circles, scream and yell! No, but whenever you can't figure out the meaning a command, you can always look up the command in PPCdisassembler and it'll explain it to you. Oh, yeah, don't forget to turn on the "add meaning" option in the Preferences. You might also want to set the origin to zero, to make the offsets of the commands realistic.

There is another use for the dissasemblers. And that is the following. In 68k, when you were looking for the resource ID's of dialogs or windows you used the CodeEditor part of Super ResEdit. You can use the dissasemblers to do this. Just search for the hex value of the dialog ID you're looking for and see if there's a branch command right after it somewhere. Since the first 2 bytes of the PPC assembly command are used for the mnemonic, don't just search for "80", search for "0080". This will greatly speed up your search. [If you're using PPCDissasembeler when searching for "0080" for example, make sure that you mask it with "FFFF" (open the "search" dialog and you'll understand what I'm talking about).]

If you're looking for the part of the code that passes a 4 letter resource to the memory, then you have to realize the "limits" of PPC. One of the limits is that you can only pass 2 byte long digits into a register with one call. So pushing the letters "PREF" into a register would require two commands. A typical example of how this is could be done:

lis R4,$5052 * load immediate shifted, 'PR'
li R4,$4546 * load immediate, 'EF'

Using The Hex Editor
~~~~~~~~~~ ~~ ~~ ~~~~~~

Right, so you finally found the conditional you need to change. What do you do? First of all, write down the hexadecimal value of the command you wish to change, along with the values of the next or previous four commands as well. Then, since all assembly information is stored in the program's data fork, open the program with your hex editor. Search for the hex values you wrote down, and replace the undesired one with 4800xxxx (BRA) or 60000000 (NOP) depending on what you need.
Quit the hex editor, launch the program you're trying to crack, and watch your computer freeze! Well that is if you fucked up somewhere on the way ;-). If you didn't you should have a fully functional cracked program!

A little hint here, in order to avoid unwanted freezes, when you're searching for the the hex values, make sure that there's only one set of those in the data fork! Since you can't really work with offsets as you could in the 68k environment, you are forced to go trial and error. I think it's obvious that if you write down many hex values of assembly commands, then the chance of them occurring in that specific order several times is reduced! Really, this is where the wetware part of the whole thing kicks in! Use your head!

Note About FAT Binary Applications
~~~~~~~~~~ ~~ ~~ ~~~~~~~~ ~~ ~~ ~~~~~~~~

FAT applications work on the principle, as you all know, that it'll run on both PPC and 68k computers. It achieves this by having the 68k code in the CODE resource and the PPC code in the data fork. There are two freewares out there called Strip 68k/ PPC by Phase Consulting. What they do is that they take away either the 68k or the PPC code from a FAT application. So if you feel more comfortable cracking 68k apps, then use the above program to remove the PPC coding from it!

NOTE: When I stripped the PPC code from a larger application I was faced with all sorts of error dialogs.

End Notes
~~~~~~~~~

And that's about all you need to know. If there's any need for it or if I'm extremely bored I might write a beginners guide to PPC cracking, but until then you'll just have to learn how to crack in 68k properly first! ;-)

1998.11.10 - updated 1999.05.16
ProZaq
prozaq@usa.net

“So the people turned to their God and asked:
- Oh Lord, will you take away pain and misery? And he replied:
- No, but I’ll give you ProZaq”