Project Title: Forensics Credential Harvester

Author: Rohan Das
Registration No: 23MCAN0218
Date: *31/10/2024*
Version: 1.0
Environment: Windows

TABLE OF CONTENTS

1. Introduction

This report outlines the implementation of a tool designed to retrieve saved passwords from Google Chrome and Mozilla Firefox browsers. The tool operates on a Windows system and provides a consolidated view of credentials stored within these browsers. The primary motivation for this tool is to simplify password recovery for legitimate purposes, such as data recovery or personal account management.

The tool consists of three primary Python scripts:

- `chrome.py`: Handles Chrome password extraction.
- `firefox.py`: Manages Firefox password retrieval.
- `main.py`: Integrates both extraction scripts, allowing the user to choose which browser's credentials to extract or exit.

---

2. Project Objective

The primary objective of this project is to enable password extraction for Chrome and Firefox by decrypting the saved credentials from each browser’s database files. Key functionalities include:

- **Chrome Password Extraction**: Decryption of saved Chrome passwords from the SQLite database.
- **Firefox Password Extraction**: Access and decryption of saved Firefox passwords.
- **User Interface**: A simple CLI interface allowing the user to select the browser for extraction or exit the program.

3. Tools and Libraries Used

The following libraries are utilized to enable the extraction and decryption functionalities:

- **Colorama**: Adds color coding for the CLI for better readability and user guidance.
- **SQLite3**: For accessing browser database files containing encrypted credentials.
- **Cryptodome**: Enables decryption of passwords via AES (for Chrome and Firefox).
- **os, sys, json, base64**: Standard Python libraries for file management, data encoding, and decoding.
- **shutil**: For handling temporary files created during database extraction.

Note: Each library serves a critical purpose, with encryption and decryption handled by `Cryptodome`, database access via `SQLite3`, and color-coded CLI outputs via `Colorama`.

---

4. Code Structure and Functionality

The code is structured across three main scripts, with a central script `main.py` integrating `chrome.py` and `firefox.py`. This modular design ensures ease of maintenance, updates, and reuse of each browser's password extraction logic.

4.1 `chrome.py`

- **Objective**: Extract and decrypt passwords saved in Google Chrome's SQLite database.
- **Methodology**:

- Accesses Chrome’s `Login Data` SQLite database file.
- Retrieves the decryption key from Chrome’s `Local State` file.
- Decrypts usernames and passwords and stores them in a structured format.

```
import os
import re
import sys
import json
import base64
import sqlite3
import win32crypt
from Crypto.Cipher import AES
import shutil
import csv

# Constants
CHROME_LOCAL_STATE_PATH = os.path.join(os.environ['USERPROFILE'],
r"AppData\Local\Google\Chrome\User Data\Local State")
CHROME_LOGIN_DATA_PATH = os.path.join(os.environ['USERPROFILE'],
r"AppData\Local\Google\Chrome\User Data")

def retrieve_secret_key():
    try:
        # Open Chrome's local state to retrieve the encrypted key
        with open(CHROME_LOCAL_STATE_PATH, "r", encoding='utf-8') as file:
            local_state_data = json.loads(file.read())
        encrypted_key =
base64.b64decode(local_state_data["os_crypt"]["encrypted_key"])[5:]
        decrypted_key = win32crypt.CryptUnprotectData(encrypted_key, None, None, None,
0)[1]
        return decrypted_key
    except Exception as error:
        print(f"[ERROR] Failed to retrieve secret key: {error}")
        return None
```

```python
def create_cipher_instance(aes_key, iv):
    return AES.new(aes_key, AES.MODE_GCM, iv)

def decrypt_chrome_password(ciphertext, secret_key):
    try:
        iv = ciphertext[3:15]  # Extract initialization vector
        encrypted_password = ciphertext[15:-16]  # Extract encrypted password
        cipher = create_cipher_instance(secret_key, iv)
        decrypted_password = cipher.decrypt(encrypted_password).decode()
        return decrypted_password
    except Exception as error:
        print(f"[ERROR] Decryption failed: {error}")
        return ""

def connect_to_chrome_db(login_data_path):
    try:
        shutil.copy2(login_data_path, "TemporaryLoginData.db")
        return sqlite3.connect("TemporaryLoginData.db")
    except Exception as error:
        print(f"[ERROR] Unable to access Chrome database: {error}")
        return None

def extract_and_save_passwords():
    try:
        # Setup CSV file to store extracted credentials
        with open('extracted_chrome_passwords.csv', mode='w', newline='',
encoding='utf-8') as password_file:
            csv_writer = csv.writer(password_file)
            csv_writer.writerow(["Index", "Website", "Username", "Password"])

            # Retrieve the secret key for decryption
            secret_key = retrieve_secret_key()
            if not secret_key:
                print("[ERROR] Secret key retrieval failed. Exiting.")
                return

            # Identify Chrome profiles
            profile_folders = [folder for folder in os.listdir(CHROME_LOGIN_DATA_PATH) if
re.match(r"^Profile|^Default$", folder)]
            for profile in profile_folders:
                login_data_path = os.path.join(CHROME_LOGIN_DATA_PATH, profile, "Login
Data")
```

```
                db_connection = connect_to_chrome_db(login_data_path)

                if db_connection:
                    cursor = db_connection.cursor()
                    cursor.execute("SELECT action_url, username_value, password_value
FROM logins")

                    for idx, entry in enumerate(cursor.fetchall()):
                        site_url, username, encrypted_pass = entry
                        if site_url and username and encrypted_pass:
                            decrypted_pass = decrypt_chrome_password(encrypted_pass,
secret_key)
                            print(f"[INFO] Record {idx} | Site: {site_url} | User:
{username} | Pass: {decrypted_pass}")
                            csv_writer.writerow([idx, site_url, username,
decrypted_pass])

                    # Close and remove the temporary database copy
                    cursor.close()
                    db_connection.close()
                    os.remove("TemporaryLoginData.db")
    except Exception as error:
        print(f"[ERROR] An unexpected error occurred: {error}")

if __name__ == '__main__':
    extract_and_save_passwords()
```

4.2 `firefox.py`

- **Objective**: Extract and decrypt passwords stored in Mozilla Firefox’s key database (`key4.db`).
- **Methodology**:
    - Connects to Firefox’s password database and retrieves encrypted usernames and passwords.

- Uses the Firefox decryption key to decrypt and display stored credentials in a user-friendly format.

```python
from __future__ import annotations

import argparse
import csv
import ctypes as ct
import json
import logging
import locale
import os
import platform
import sqlite3
import sys
import shutil
from base64 import b64decode
from getpass import getpass
from itertools import chain
from subprocess import run, PIPE, DEVNULL
from urllib.parse import urlparse
from configparser import ConfigParser
from typing import Optional, Iterator, Any

LOG: logging.Logger
VERBOSE = False
SYSTEM = platform.system()
SYS64 = sys.maxsize > 2**32
DEFAULT_ENCODING = "utf-8"

PWStore = list[dict[str, str]]



def get_version() -> str:
    """Obtain version information from git if available otherwise use
    the internal version number
    """

    def internal_version():
        return ".".join(map(str, __version_info__[:3])) +
"".join(__version_info__[3:])

    try:
        p = run(["git", "describe", "--tags"], stdout=PIPE, stderr=DEVNULL, text=True)
    except FileNotFoundError:
        return internal_version()

    if p.returncode:
        return internal_version()
    else:
        return p.stdout.strip()


__version_info__ = (1, 1, 1, "+git")
__version__: str = get_version()


class NotFoundError(Exception):
    """Exception to handle situations where a credentials file is not found"""

    pass


class Exit(Exception):
    """Exception to allow a clean exit from any point in execution"""

    CLEAN = 0
    ERROR = 1
    MISSING_PROFILEINI = 2
    MISSING_SECRETS = 3
    BAD_PROFILEINI = 4
    LOCATION_NO_DIRECTORY = 5
    BAD_SECRETS = 6
    BAD_LOCALE = 7

    FAIL_LOCATE_NSS = 10
    FAIL_LOAD_NSS = 11
    FAIL_INIT_NSS = 12
    FAIL_NSS_KEYSLOT = 13
    FAIL_SHUTDOWN_NSS = 14
    BAD_PRIMARY_PASSWORD = 15
    NEED_PRIMARY_PASSWORD = 16
    DECRYPTION_FAILED = 17

    PASSSTORE_NOT_INIT = 20
    PASSSTORE_MISSING = 21
    PASSSTORE_ERROR = 22

    READ_GOT_EOF = 30
    MISSING_CHOICE = 31
    NO_SUCH_PROFILE = 32

    UNKNOWN_ERROR = 100
    KEYBOARD_INTERRUPT = 102

    def __init__(self, exitcode):
        self.exitcode = exitcode

    def __unicode__(self):
        return f"Premature program exit with exit code {self.exitcode}"


class Credentials:
    """Base credentials backend manager"""

    def __init__(self, db):
        self.db = db

        LOG.debug("Database location: %s", self.db)
        if not os.path.isfile(db):
            raise NotFoundError(f"ERROR - {db} database not found\n")

        LOG.info("Using %s for credentials.", db)

    def __iter__(self) -> Iterator[tuple[str, str, str, int]]:
        pass

    def done(self):
        """Override this method if the credentials subclass needs to do any
        action after interaction
        """
        pass
```

```
class SqliteCredentials(Credentials):
    """SQLite credentials backend manager"""

    def __init__(self, profile):
        db = os.path.join(profile, "signons.sqlite")

        super(SqliteCredentials, self).__init__(db)

        self.conn = sqlite3.connect(db)
        self.c = self.conn.cursor()

    def __iter__(self) -> Iterator[tuple[str, str, str, int]]:
        LOG.debug("Reading password database in SQLite format")
        self.c.execute(
            "SELECT hostname, encryptedUsername, encryptedPassword, encType "
            "FROM moz_logins"
        )
        for i in self.c:
            # yields hostname, encryptedUsername, encryptedPassword, encType
            yield i

    def done(self):
        """Close the sqlite cursor and database connection"""
        super(SqliteCredentials, self).done()

        self.c.close()
        self.conn.close()


class JsonCredentials(Credentials):
    """JSON credentials backend manager"""

    def __init__(self, profile):
        db = os.path.join(profile, "logins.json")

        super(JsonCredentials, self).__init__(db)

    def __iter__(self) -> Iterator[tuple[str, str, str, int]]:
        with open(self.db) as fh:
            LOG.debug("Reading password database in JSON format")
            data = json.load(fh)
            try:
                logins = data["logins"]
            except Exception:
                LOG.error(f"Unrecognized format in {self.db}")
                raise Exit(Exit.BAD_SECRETS)

            for i in logins:
                try:
                    yield (
                        i["hostname"],
                        i["encryptedUsername"],
                        i["encryptedPassword"],
                        i["encType"],
                    )
                except KeyError:
                    # This should handle deleted passwords that still maintain
                    # a record in the JSON file - GitHub issue #99
                    LOG.info(f"Skipped record {i} due to missing fields")


def find_nss(locations: list[str], nssname: str) -> ct.CDLL:
    """Locate nss is one of the many possible locations"""
    fail_errors: list[tuple[str, str]] = []

    OS = ("Windows", "Darwin")

    for loc in locations:
        nsslib = os.path.join(loc, nssname)
        LOG.debug("Loading NSS library from %s", nsslib)

        if SYSTEM in OS:
            # On windows in order to find DLLs referenced by nss3.dll
            # we need to have those locations on PATH
            os.environ["PATH"] = ";".join([loc, os.environ["PATH"]])
            LOG.debug("PATH is now %s", os.environ["PATH"])
            # However this doesn't seem to work on all setups and needs to be
            # set before starting python so as a workaround we chdir to
            # Firefox's nss3.dll/libnss3.dylib location
            if loc:
                if not os.path.isdir(loc):
                    # No point in trying to load from paths that don't exist
                    continue

                workdir = os.getcwd()
                os.chdir(loc)

        try:
            nss: ct.CDLL = ct.CDLL(nsslib)
        except OSError as e:
            fail_errors.append((nsslib, str(e)))
        else:
            LOG.debug("Loaded NSS library from %s", nsslib)
            return nss
        finally:
            if SYSTEM in OS and loc:
                # Restore workdir changed above
                os.chdir(workdir)

    else:
        LOG.error(
            "Couldn't find or load '%s'. This library is essential "
            "to interact with your Mozilla profile.",
            nssname,
        )
        LOG.error(
            "If you are seeing this error please perform a system-wide "
            "search for '%s' and file a bug report indicating any "
            "location found. Thanks!",
            nssname,
        )
        LOG.error(
            "Alternatively you can try launching firefox_decrypt "
            "from the location where you found '%s'. "
            "That is 'cd' or 'chdir' to that location and run "
            "firefox_decrypt from there.",
            nssname,
        )

        LOG.error(
            "Please also include the following on any bug report. "
            "Errors seen while searching/loading NSS:"
        )
        for target, error in fail_errors:
            LOG.error("Error when loading %s was %s", target, error)

        raise Exit(Exit.FAIL_LOCATE_NSS)


def load_libnss():
    """Load libnss into python using the CDLL interface"""

    locations: list[str] = [
        os.environ.get("NSS_LIB_PATH", ""),
    ]

    if SYSTEM == "Windows":
        nssname = "nss3.dll"
        if not SYS64:
            locations += [
                "",  # Current directory or system lib finder
                "C:\\Program Files (x86)\\Mozilla Firefox",
                "C:\\Program Files (x86)\\Firefox Developer Edition",
                "C:\\Program Files (x86)\\Mozilla Thunderbird",
                "C:\\Program Files (x86)\\Nightly",
                "C:\\Program Files (x86)\\SeaMonkey",
                "C:\\Program Files (x86)\\Waterfox",
            ]

        locations += [
            "",  # Current directory or system lib finder
            os.path.expanduser("~\\AppData\\Local\\Mozilla Firefox"),
            os.path.expanduser("~\\AppData\\Local\\Firefox Developer Edition"),
            os.path.expanduser("~\\AppData\\Local\\Mozilla Thunderbird"),
            os.path.expanduser("~\\AppData\\Local\\Nightly"),
            os.path.expanduser("~\\AppData\\Local\\SeaMonkey"),
            os.path.expanduser("~\\AppData\\Local\\Waterfox"),
            "C:\\Program Files\\Mozilla Firefox",
            "C:\\Program Files\\Firefox Developer Edition",
            "C:\\Program Files\\Mozilla Thunderbird",
            "C:\\Program Files\\Nightly",
            "C:\\Program Files\\SeaMonkey",
            "C:\\Program Files\\Waterfox",
        ]
        # If either of the supported software is in PATH try to use it
        software = ["firefox", "thunderbird", "waterfox", "seamonkey"]
        for binary in software:
            location: Optional[str] = shutil.which(binary)
            if location is not None:
                nsslocation: str = os.path.join(os.path.dirname(location), nssname)
                locations.append(nsslocation)

    elif SYSTEM == "Darwin":
        nssname = "libnss3.dylib"
        locations += [
            "",  # Current directory or system lib finder
            "/usr/local/lib/nss",
            "/usr/local/lib",
            "/opt/local/lib/nss",
            "/sw/lib/firefox",
            "/sw/lib/mozilla",
            "/usr/local/opt/nss/lib",  # nss installed with Brew on Darwin
            "/opt/pkg/lib/nss",  # installed via pkgsrc
            "/Applications/Firefox.app/Contents/MacOS",  # default manual install
location
            "/Applications/Thunderbird.app/Contents/MacOS",
            "/Applications/SeaMonkey.app/Contents/MacOS",
            "/Applications/Waterfox.app/Contents/MacOS",
        ]

    else:
        nssname = "libnss3.so"
        if SYS64:
            locations += [
                "",  # Current directory or system lib finder
                "/usr/lib64",
                "/usr/lib64/nss",
                "/usr/lib",
                "/usr/lib/nss",
                "/usr/local/lib",
                "/usr/local/lib/nss",
                "/opt/local/lib",
                "/opt/local/lib/nss",
                os.path.expanduser("~/.nix-profile/lib"),
            ]
        else:
            locations += [
                "",  # Current directory or system lib finder
                "/usr/lib",
                "/usr/lib/nss",
                "/usr/lib32",
                "/usr/lib32/nss",
                "/usr/lib64",
                "/usr/lib64/nss",
                "/usr/local/lib",
                "/usr/local/lib/nss",
                "/opt/local/lib",
                "/opt/local/lib/nss",
                os.path.expanduser("~/.nix-profile/lib"),
            ]

    # If this succeeds libnss was loaded
    return find_nss(locations, nssname)


class c_char_p_fromstr(ct.c_char_p):
    """ctypes char_p override that handles encoding str to bytes"""

    def from_param(self):
        return self.encode(DEFAULT_ENCODING)


class NSSProxy:
    class SECItem(ct.Structure):
        """struct needed to interact with libnss"""

        _fields_ = [
            ("type", ct.c_uint),
            ("data", ct.c_char_p),  # actually: unsigned char *
            ("len", ct.c_uint),
        ]

        def decode_data(self):
            _bytes = ct.string_at(self.data, self.len)
            return _bytes.decode(DEFAULT_ENCODING)

    class PK11SlotInfo(ct.Structure):
        """Opaque structure representing a logical PKCS slot"""
    def __init__(self, non_fatal_decryption=False):
        # Locate libnss and try loading it
        self.libnss = load_libnss()
        self.non_fatal_decryption = non_fatal_decryption

        SlotInfoPtr = ct.POINTER(self.PK11SlotInfo)
        SECItemPtr = ct.POINTER(self.SECItem)

        self._set_ctypes(ct.c_int, "NSS_Init", c_char_p_fromstr)
        self._set_ctypes(ct.c_int, "NSS_Shutdown")
        self._set_ctypes(SlotInfoPtr, "PK11_GetInternalKeySlot")
        self._set_ctypes(None, "PK11_FreeSlot", SlotInfoPtr)
        self._set_ctypes(ct.c_int, "PK11_NeedLogin", SlotInfoPtr)
        self._set_ctypes(
            ct.c_int, "PK11_CheckUserPassword", SlotInfoPtr, c_char_p_fromstr
        )
        self._set_ctypes(
            ct.c_int, "PK11SDR_Decrypt", SECItemPtr, SECItemPtr, ct.c_void_p
        )
        self._set_ctypes(None, "SECITEM_ZfreeItem", SECItemPtr, ct.c_int)

        # for error handling
        self._set_ctypes(ct.c_int, "PORT_GetError")
        self._set_ctypes(ct.c_char_p, "PR_ErrorToName", ct.c_int)
        self._set_ctypes(ct.c_char_p, "PR_ErrorToString", ct.c_int, ct.c_uint32)

    def _set_ctypes(self, restype, name, *argtypes):
        """Set input/output types on libnss C functions for automatic type casting"""
        res = getattr(self.libnss, name)
        res.argtypes = argtypes
        res.restype = restype

        # Transparently handle decoding to string when returning a c_char_p
        if restype == ct.c_char_p:

            def _decode(result, func, *args):
                try:
                    return result.decode(DEFAULT_ENCODING)
                except AttributeError:
                    return result
```

```python
            res.errcheck = _decode

        setattr(self, "_" + name, res)

    def initialize(self, profile: str):
        # The sql: prefix ensures compatibility with both
        # Berkley DB (cert8) and Sqlite (cert9) dbs
        profile_path = "sql:" + profile
        LOG.debug("Initializing NSS with profile '%s'", profile_path)
        err_status: int = self._NSS_Init(profile_path)
        LOG.debug("Initializing NSS returned %s", err_status)

        if err_status:
            self.handle_error(
                Exit.FAIL_INIT_NSS,
                "Couldn't initialize NSS, maybe '%s' is not a valid profile?",
                profile,
            )

    def shutdown(self):
        err_status: int = self._NSS_Shutdown()

        if err_status:
            self.handle_error(
                Exit.FAIL_SHUTDOWN_NSS,
                "Couldn't shutdown current NSS profile",
            )

    def authenticate(self, profile, interactive):
        """Unlocks the profile if necessary, in which case a password
        will prompted to the user.
        """
        LOG.debug("Retrieving internal key slot")
        keyslot = self._PK11_GetInternalKeySlot()

        LOG.debug("Internal key slot %s", keyslot)
        if not keyslot:
            self.handle_error(
                Exit.FAIL_NSS_KEYSLOT,
                "Failed to retrieve internal KeySlot",
            )
        try:
            if self._PK11_NeedLogin(keyslot):
                password: str = ask_password(profile, interactive)

                LOG.debug("Authenticating with password '%s'", password)
                err_status: int = self._PK11_CheckUserPassword(keyslot, password)

                LOG.debug("Checking user password returned %s", err_status)

                if err_status:
                    self.handle_error(
                        Exit.BAD_PRIMARY_PASSWORD,
                        "Primary password is not correct",
                    )

            else:
                LOG.info("No Primary Password found - no authentication needed")
        finally:
            # Avoid leaking PK11KeySlot
            self._PK11_FreeSlot(keyslot)

    def handle_error(self, exitcode: int, *logerror: Any):
        """If an error happens in libnss, handle it and print some debug
information"""
        if logerror:
            LOG.error(*logerror)
        else:
            LOG.debug("Error during a call to NSS library, trying to obtain error
info")

        code = self._PORT_GetError()
        name = self._PR_ErrorToName(code)
        name = "NULL" if name is None else name
        # 0 is the default language (localization related)
        text = self._PR_ErrorToString(code, 0)

        LOG.debug("%s: %s", name, text)

        raise Exit(exitcode)

    def decrypt(self, data64):
        data = b64decode(data64)
        inp = self.SECItem(0, data, len(data))
        out = self.SECItem(0, None, 0)

        err_status: int = self._PK11SDR_Decrypt(inp, out, None)
        LOG.debug("Decryption of data returned %s", err_status)
        try:
            if err_status:  # -1 means password failed, other status are unknown
                error_msg = (
                    "Username/Password decryption failed. "
                    "Credentials damaged or cert/key file mismatch."
                )

                if self.non_fatal_decryption:
                    raise ValueError(error_msg)
                else:
                    self.handle_error(Exit.DECRYPTION_FAILED, error_msg)

            res = out.decode_data()
        finally:
            # Avoid leaking SECItem
            self._SECITEM_ZfreeItem(out, 0)

        return res


class MozillaInteraction:
    """
    Abstraction interface to Mozilla profile and lib NSS
    """

    def __init__(self, non_fatal_decryption=False):
        self.profile = None
        self.proxy = NSSProxy(non_fatal_decryption)

    def load_profile(self, profile):
        """Initialize the NSS library and profile"""
        self.profile = profile
        self.proxy.initialize(self.profile)

    def authenticate(self, interactive):
        """Authenticate the the current profile is protected by a primary password,
        prompt the user and unlock the profile.
        """
        self.proxy.authenticate(self.profile, interactive)

    def unload_profile(self):
        """Shutdown NSS and deactivate current profile"""
        self.proxy.shutdown()

    def decrypt_passwords(self) -> PWStore:
        """Decrypt requested profile using the provided password.
        Returns all passwords in a list of dicts
        """
        credentials: Credentials = self.obtain_credentials()

        LOG.info("Decrypting credentials")
        outputs: PWStore = []

        url: str
        user: str
        passw: str
        enctype: int
        for url, user, passw, enctype in credentials:
            # enctype informs if passwords need to be decrypted
            if enctype:
                try:
                    LOG.debug("Decrypting username data '%s'", user)
                    user = self.proxy.decrypt(user)
                    LOG.debug("Decrypting password data '%s'", passw)
                    passw = self.proxy.decrypt(passw)
                except (TypeError, ValueError) as e:
                    LOG.warning(
                        "Failed to decode username or password for entry from URL %s",
                        url,
                    )
                    LOG.debug(e, exc_info=True)
                    user = "*** decryption failed ***"
                    passw = "*** decryption failed ***"

            LOG.debug(
                "Decoded username '%s' and password '%s' for website '%s'",
                user,
                passw,
                url,
            )

            output = {"url": url, "user": user, "password": passw}
            outputs.append(output)

        if not outputs:
            LOG.warning("No passwords found in selected profile")

        # Close credential handles (SQL)
        credentials.done()

        return outputs

    def obtain_credentials(self) -> Credentials:
        """Figure out which of the 2 possible backend credential engines is
available"""
        credentials: Credentials
        try:
            credentials = JsonCredentials(self.profile)
        except NotFoundError:
            try:
                credentials = SqliteCredentials(self.profile)
            except NotFoundError:
                LOG.error(
                    "Couldn't find credentials file (logins.json or signons.sqlite)."
                )
                raise Exit(Exit.MISSING_SECRETS)

        return credentials


class OutputFormat:
    def __init__(self, pwstore: PWStore, cmdargs: argparse.Namespace):
        self.pwstore = pwstore
        self.cmdargs = cmdargs

    def output(self):
        pass


class HumanOutputFormat(OutputFormat):
    def output(self):
```

```python
        for output in self.pwstore:
            record: str = (
                f"\nWebsite:   {output['url']}\n"
                f"Username: '{output['user']}'\n"
                f"Password: '{output['password']}'\n"
            )
            sys.stdout.write(record)


class JSONOutputFormat(OutputFormat):
    def output(self):
        sys.stdout.write(json.dumps(self.pwstore, indent=2))
        # Json dumps doesn't add the final newline
        sys.stdout.write("\n")


class CSVOutputFormat(OutputFormat):
    def __init__(self, pwstore: PWStore, cmdargs: argparse.Namespace):
        super().__init__(pwstore, cmdargs)
        self.delimiter = cmdargs.csv_delimiter
        self.quotechar = cmdargs.csv_quotechar
        self.header = cmdargs.csv_header

    def output(self):
        csv_writer = csv.DictWriter(
            sys.stdout,
            fieldnames=["url", "user", "password"],
            lineterminator="\n",
            delimiter=self.delimiter,
            quotechar=self.quotechar,
            quoting=csv.QUOTE_ALL,
        )
        if self.header:
            csv_writer.writeheader()

        for output in self.pwstore:
            csv_writer.writerow(output)


class TabularOutputFormat(CSVOutputFormat):
    def __init__(self, pwstore: PWStore, cmdargs: argparse.Namespace):
        super().__init__(pwstore, cmdargs)
        self.delimiter = "\t"
        self.quotechar = "'"


class PassOutputFormat(OutputFormat):
    def __init__(self, pwstore: PWStore, cmdargs: argparse.Namespace):
        super().__init__(pwstore, cmdargs)
        self.prefix = cmdargs.pass_prefix
        self.cmd = cmdargs.pass_cmd
        self.username_prefix = cmdargs.pass_username_prefix
        self.always_with_login = cmdargs.pass_always_with_login

    def output(self):
        self.test_pass_cmd()
        self.preprocess_outputs()
        self.export()

    def test_pass_cmd(self) -> None:
        """Check if pass from passwordstore.org is installed
        If it is installed but not initialized, initialize it
        """
        LOG.debug("Testing if password store is installed and configured")

        try:
            p = run([self.cmd, "ls"], capture_output=True, text=True)
        except FileNotFoundError as e:
            if e.errno == 2:
                LOG.error("Password store is not installed and exporting was
requested")
                raise Exit(Exit.PASSSTORE_MISSING)
            else:
                LOG.error("Unknown error happened.")
                LOG.error("Error was '%s'", e)
                raise Exit(Exit.UNKNOWN_ERROR)

        LOG.debug("pass returned:\nStdout: %s\nStderr: %s", p.stdout, p.stderr)

        if p.returncode != 0:
            if 'Try "pass init"' in p.stderr:
                LOG.error("Password store was not initialized.")
                LOG.error("Initialize the password store manually by using 'pass
init'")
```

```
                raise Exit(Exit.PASSSTORE_NOT_INIT)
            else:
                LOG.error("Unknown error happened when running 'pass'.")
                LOG.error("Stdout: %s\nStderr: %s", p.stdout, p.stderr)
                raise Exit(Exit.UNKNOWN_ERROR)

    def preprocess_outputs(self):
        # Format of "self.to_export" should be:
        #     {"address": {"login": "password", ...}, ...}
        self.to_export: dict[str, dict[str, str]] = {}

        for record in self.pwstore:
            url = record["url"]
            user = record["user"]
            passw = record["password"]

            # Keep track of web-address, username and passwords
            # If more than one username exists for the same web-address
            # the username will be used as name of the file
            address = urlparse(url)

            if address.netloc not in self.to_export:
                self.to_export[address.netloc] = {user: passw}

            else:
                self.to_export[address.netloc][user] = passw

    def export(self):
        """Export given passwords to password store

        Format of "to_export" should be:
            {"address": {"login": "password", ...}, ...}
        """
        LOG.info("Exporting credentials to password store")
        if self.prefix:
            prefix = f"{self.prefix}/"
        else:
            prefix = self.prefix

        LOG.debug("Using pass prefix '%s'", prefix)

        for address in self.to_export:
```

```python
            for user, passw in self.to_export[address].items():
                # When more than one account exist for the same address, add
                # the login to the password identifier
                if self.always_with_login or len(self.to_export[address]) > 1:
                    passname = f"{prefix}{address}/{user}"
                else:
                    passname = f"{prefix}{address}"

                LOG.info("Exporting credentials for '%s'", passname)

                data = f"{passw}\n{self.username_prefix}{user}\n"

                LOG.debug("Inserting pass '%s' '%s'", passname, data)

                # NOTE --force is used. Existing passwords will be overwritten
                cmd: list[str] = [
                    self.cmd,
                    "insert",
                    "--force",
                    "--multiline",
                    passname,
                ]

                LOG.debug("Running command '%s' with stdin '%s'", cmd, data)

                p = run(cmd, input=data, capture_output=True, text=True)

                if p.returncode != 0:
                    LOG.error(
                        "ERROR: passwordstore exited with non-zero: %s", p.returncode
                    )
                    LOG.error("Stdout: %s\nStderr: %s", p.stdout, p.stderr)
                    raise Exit(Exit.PASSSTORE_ERROR)

                LOG.debug("Successfully exported '%s'", passname)


def get_sections(profiles):
    """
    Returns hash of profile numbers and profile names.
    """
    sections = {}
    i = 1
    for section in profiles.sections():
        if section.startswith("Profile"):
            sections[str(i)] = profiles.get(section, "Path")
            i += 1
        else:
            continue
    return sections


def print_sections(sections, textIOWrapper=sys.stderr):
    """
    Prints all available sections to an textIOWrapper (defaults to sys.stderr)
    """
    for i in sorted(sections):
        textIOWrapper.write(f"{i} -> {sections[i]}\n")
    textIOWrapper.flush()


def ask_section(sections: ConfigParser):
    """
    Prompt the user which profile should be used for decryption
    """
    # Do not ask for choice if user already gave one
    choice = "ASK"
    while choice not in sections:
        sys.stderr.write("\n\n[+] Select the Mozilla profile you wish to decrypt\n\n")
        print_sections(sections)
        try:
            choice = input("\n[*] Enter your option: ")
        except EOFError:
            LOG.error("Could not read Choice, got EOF")
            raise Exit(Exit.READ_GOT_EOF)

    try:
        final_choice = sections[choice]
    except KeyError:
        LOG.error("Profile No. %s does not exist!", choice)
        raise Exit(Exit.NO_SUCH_PROFILE)

    LOG.debug("Profile selection matched %s", final_choice)
    return final_choice


def ask_password(profile: str, interactive: bool) -> str:
    """
    Prompt for profile password
    """
    passwd: str
    passmsg = f"\nPrimary Password for profile {profile}: "

    if sys.stdin.isatty() and interactive:
        passwd = getpass(passmsg)
    else:
        sys.stderr.write("Reading Primary password from standard input:\n")
        sys.stderr.flush()
        # Ability to read the password from stdin (echo "pass" | ./firefox_...)
        passwd = sys.stdin.readline().rstrip("\n")

    return passwd


def read_profiles(basepath):
    """
    Parse Firefox profiles in provided location.
    If list_profiles is true, will exit after listing available profiles.
    """
    profileini = os.path.join(basepath, "profiles.ini")

    LOG.debug("Reading profiles from %s", profileini)

    if not os.path.isfile(profileini):
        LOG.warning("profile.ini not found in %s", basepath)
        raise Exit(Exit.MISSING_PROFILEINI)

    # Read profiles from Firefox profile folder
    profiles = ConfigParser()
    profiles.read(profileini, encoding=DEFAULT_ENCODING)

    LOG.debug("Read profiles %s", profiles.sections())

    return profiles
```

```
def get_profile(
    basepath: str, interactive: bool, choice: Optional[str], list_profiles: bool
):
    """
    Select profile to use by either reading profiles.ini or assuming given
    path is already a profile
    If interactive is false, will not try to ask which profile to decrypt.
    choice contains the choice the user gave us as an CLI arg.
    If list_profiles is true will exits after listing all available profiles.
    """
    try:
        profiles: ConfigParser = read_profiles(basepath)

    except Exit as e:
        if e.exitcode == Exit.MISSING_PROFILEINI:
            LOG.warning("Continuing and assuming '%s' is a profile location",
basepath)
            profile = basepath

            if list_profiles:
                LOG.error("Listing single profiles not permitted.")
                raise

            if not os.path.isdir(profile):
                LOG.error("Profile location '%s' is not a directory", profile)
                raise
        else:
            raise
    else:
        if list_profiles:
            LOG.debug("Listing available profiles...")
            print_sections(get_sections(profiles), sys.stdout)
            raise Exit(Exit.CLEAN)

        sections = get_sections(profiles)

        if len(sections) == 1:
            section = sections["1"]

        elif choice is not None:
            try:
                section = sections[choice]
            except KeyError:
                LOG.error("Profile No. %s does not exist!", choice)
                raise Exit(Exit.NO_SUCH_PROFILE)

        elif not interactive:
            LOG.error(
                "Don't know which profile to decrypt. "
                "We are in non-interactive mode and -c/--choice wasn't specified."
            )
            raise Exit(Exit.MISSING_CHOICE)

        else:
            # Ask user which profile to open
            section = ask_section(sections)

        section = section
        profile = os.path.join(basepath, section)

        if not os.path.isdir(profile):
            LOG.error(
                "Profile location '%s' is not a directory. Has profiles.ini been
tampered with?",
                profile,
            )
            raise Exit(Exit.BAD_PROFILEINI)

    return profile


# From https://bugs.python.org/msg323681
class ConvertChoices(argparse.Action):
    """Argparse action that interprets the `choices` argument as a dict
    mapping the user-specified choices values to the resulting option
    values.
    """

    def __init__(self, *args, choices, **kwargs):
        super().__init__(*args, choices=choices.keys(), **kwargs)
        self.mapping = choices

    def __call__(self, parser, namespace, value, option_string=None):
        setattr(namespace, self.dest, self.mapping[value])


def parse_sys_args() -> argparse.Namespace:
    """Parse command line arguments"""

    if SYSTEM == "Windows":
        profile_path = os.path.join(os.environ["APPDATA"], "Mozilla", "Firefox")
    elif os.uname()[0] == "Darwin":
        profile_path = "~/Library/Application Support/Firefox"
    else:
        profile_path = "~/.mozilla/firefox"

    parser = argparse.ArgumentParser(
        description="Access Firefox/Thunderbird profiles and decrypt existing
passwords"
    )
    parser.add_argument(
        "profile",
        nargs="?",
        default=profile_path,
        help=f"Path to profile folder (default: {profile_path})",
    )

    format_choices = {
        "human": HumanOutputFormat,
        "json": JSONOutputFormat,
        "csv": CSVOutputFormat,
        "tabular": TabularOutputFormat,
        "pass": PassOutputFormat,
    }

    parser.add_argument(
        "-f",
        "--format",
        action=ConvertChoices,
        choices=format_choices,
        default=HumanOutputFormat,
        help="Format for the output.",
    )
    parser.add_argument(
        "-d",
        "--csv-delimiter",
        action="store",
        default=";",
        help="The delimiter for csv output",
    )
    parser.add_argument(
        "-q",
        "--csv-quotechar",
        action="store",
        default='"',
        help="The quote char for csv output",
    )
    parser.add_argument(
        "--no-csv-header",
        action="store_false",
        dest="csv_header",
        default=True,
        help="Do not include a header in CSV output.",
    )
    parser.add_argument(
        "--pass-username-prefix",
        action="store",
        default="",
        help=(
            "Export username as is (default), or with the provided format prefix. "
            "For instance 'login: ' for browserpass."
        ),
    )
    parser.add_argument(
        "-p",
        "--pass-prefix",
        action="store",
        default="web",
        help="Folder prefix for export to pass from passwordstore.org (default:
%(default)s)",
    )
    parser.add_argument(
        "-m",
        "--pass-cmd",
        action="store",
        default="pass",
        help="Command/path to use when exporting to pass (default: %(default)s)",
    )
    parser.add_argument(
        "--pass-always-with-login",
        action="store_true",
        help="Always save as /<login> (default: only when multiple accounts per
domain)",
    )
    parser.add_argument(
        "-n",
        "--no-interactive",
        action="store_false",
        dest="interactive",
        default=True,
        help="Disable interactivity.",
    )
    parser.add_argument(
        "--non-fatal-decryption",
        action="store_true",
        default=False,
        help="If set, corrupted entries will be skipped instead of aborting the
process.",
    )
    parser.add_argument(
        "-c",
        "--choice",
        help="The profile to use (starts with 1). If only one profile, defaults to
that.",
    )
    parser.add_argument(
        "-l", "--list", action="store_true", help="List profiles and exit."
    )
    parser.add_argument(
        "-e",
        "--encoding",
        action="store",
        default=DEFAULT_ENCODING,
        help="Override default encoding (%(default)s).",
    )
    parser.add_argument(
        "-v",
        "--verbose",
        action="count",
        default=0,
        help="Verbosity level. Warning on -vv (highest level) user input will be 
printed on screen",
    )
    parser.add_argument(
        "--version",
        action="version",
        version=__version__,
        help="Display version of firefox_decrypt and exit",
    )

    args = parser.parse_args()

    # understand `\t` as tab character if specified as delimiter.
    if args.csv_delimiter == "\\t":
        args.csv_delimiter = "\t"

    return args


def setup_logging(args) -> None:
    """Setup the logging level and configure the basic logger"""
    if args.verbose == 1:
        level = logging.INFO
    elif args.verbose >= 2:
        level = logging.DEBUG
    else:
        level = logging.WARN

    logging.basicConfig(
        format="%(asctime)s - %(levelname)s - %(message)s",
        level=level,
    )

    global LOG
    LOG = logging.getLogger(__name__)


def identify_system_locale() -> str:
    encoding: Optional[str] = locale.getpreferredencoding()

    if encoding is None:
        LOG.error(
            "Could not determine which encoding/locale to use for NSS interaction. "
            "This configuration is unsupported.\n"
            "If you are in Linux or MacOS, please search online "
            "how to configure a UTF-8 compatible locale and try again."
        )
        raise Exit(Exit.BAD_LOCALE)

    return encoding


def main() -> None:
    """Main entry point"""
    args = parse_sys_args()

    setup_logging(args)

    global DEFAULT_ENCODING

    if args.encoding != DEFAULT_ENCODING:
        LOG.info(
            "Overriding default encoding from '%s' to '%s'",
            DEFAULT_ENCODING,
            args.encoding,
        )

        # Override default encoding if specified by user
        DEFAULT_ENCODING = args.encoding

    LOG.info("Running firefox_decrypt version: %s", __version__)
    LOG.debug("Parsed commandline arguments: %s", args)
    encodings = (
        ("stdin", sys.stdin.encoding),
        ("stdout", sys.stdout.encoding),
        ("stderr", sys.stderr.encoding),
        ("locale", identify_system_locale()),
    )

    LOG.debug(
        "Running with encodings: %s: %s, %s: %s, %s: %s, %s: %s", *chain(*encodings)
    )
    for stream, encoding in encodings:
        if encoding.lower() != DEFAULT_ENCODING:
            LOG.warning(
                "Running with unsupported encoding '%s': %s"
                " - Things are likely to fail from here onwards",
                stream,
                encoding,
            )

    # Load Mozilla profile and initialize NSS before asking the user for input
    moz = MozillaInteraction(args.non_fatal_decryption)

    basepath = os.path.expanduser(args.profile)

    # Read profiles from profiles.ini in profile folder
    profile = get_profile(basepath, args.interactive, args.choice, args.list)

    # Start NSS for selected profile
    moz.load_profile(profile)
    # Check if profile is password protected and prompt for a password
    moz.authenticate(args.interactive)
    # Decode all passwords
    outputs = moz.decrypt_passwords()

    # Export passwords into one of many formats
    formatter = args.format(outputs, args)
    formatter.output()

    # Finally shutdown NSS
    moz.unload_profile()


def firefox_decrypt():
    try:
        main()
    except KeyboardInterrupt:
        print("Quit.")
        sys.exit(Exit.KEYBOARD_INTERRUPT)
    except Exit as e:
        sys.exit(e.exitcode)
```

```
if __name__ == "__main__":
    firefox_decrypt()
```

4.3 `main.py`

- **Objective**: Provide a user interface for choosing browser password extraction.
- **Methodology**:
  - Displays a welcome ASCII banner and menu options.
  - Integrates `chrome.py` and `firefox.py`, allowing users to extract passwords from either or both browsers.
  - Executes the chosen extraction process, displays results, or exits the program.

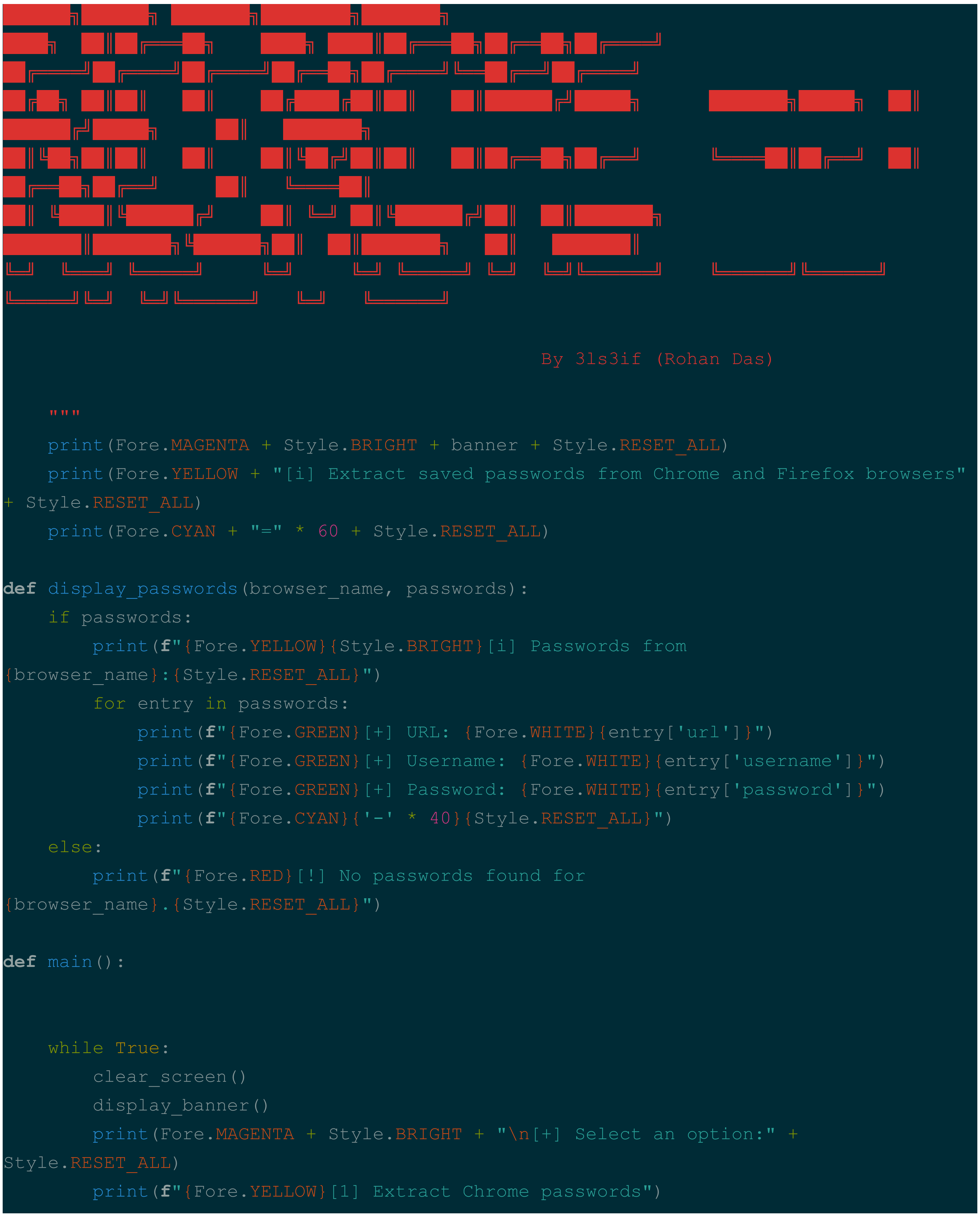
```
import chrome
import firefox
from colorama import Fore, Style, init
import os

# Initialize colorama for Windows support
init(autoreset=True)

def clear_screen():
    # Clear screen for Windows and Unix-like systems
    os.system("cls" if os.name == "nt" else "clear")

def display_banner():
    banner = r"""

███╗   ██╗ ██████╗     ███╗   ███╗ ██████╗ ██████╗ ███████╗╗     ███████╗███████╗╗
```



```
[illegible]

                                                By 3ls3if (Rohan Das)

    """
    print(Fore.MAGENTA + Style.BRIGHT + banner + Style.RESET_ALL)
    print(Fore.YELLOW + "[i] Extract saved passwords from Chrome and Firefox browsers"
+ Style.RESET_ALL)
    print(Fore.CYAN + "=" * 60 + Style.RESET_ALL)

def display_passwords(browser_name, passwords):
    if passwords:
        print(f"{Fore.YELLOW}{Style.BRIGHT}[i] Passwords from
{browser_name}:{Style.RESET_ALL}")
        for entry in passwords:
            print(f"{Fore.GREEN}[+] URL: {Fore.WHITE}{entry['url']}")
            print(f"{Fore.GREEN}[+] Username: {Fore.WHITE}{entry['username']}")
            print(f"{Fore.GREEN}[+] Password: {Fore.WHITE}{entry['password']}")
            print(f"{Fore.CYAN}{'-' * 40}{Style.RESET_ALL}")
    else:
        print(f"{Fore.RED}[!] No passwords found for
{browser_name}.{Style.RESET_ALL}")

def main():


    while True:
        clear_screen()
        display_banner()
        print(Fore.MAGENTA + Style.BRIGHT + "\n[+] Select an option:" +
Style.RESET_ALL)
        print(f"{Fore.YELLOW}[1] Extract Chrome passwords")
        print(f"{Fore.YELLOW}[2] Extract Firefox passwords")
        print(f"{Fore.YELLOW}[3] Exit{Style.RESET_ALL}")

        choice = input(Fore.CYAN + "\n[+] Enter your choice (1, 2, or 3): " +
Style.RESET_ALL)

        if choice == "1":
            try:
                print(f"\n{Fore.MAGENTA}{Style.BRIGHT}Extracting Chrome
passwords...{Style.RESET_ALL}\n")
                chrome_passwords = chrome.extract_and_save_passwords()
                display_passwords("Chrome", chrome_passwords)
                input("\n\n[+] Please press enter to continue...")
            except Exception as e:
                print(f"{Fore.RED}[ERROR] Chrome password extraction failed:
{e}{Style.RESET_ALL}")
                input("\n\n[+] Please press enter to continue...")

        elif choice == "2":
            try:
                print(f"\n{Fore.MAGENTA}{Style.BRIGHT}Extracting Firefox
passwords...{Style.RESET_ALL}\n")
                firefox_passwords = firefox.firefox_decrypt()
                display_passwords("Firefox", firefox_passwords)
                input("\n\n[+] Please press enter to continue...")
            except Exception as e:
                print(f"{Fore.RED}[ERROR] Firefox password extraction failed:
{e}{Style.RESET_ALL}")
                input("\n\n[+] Please press enter to continue...")

        elif choice == "3":
            print(Fore.GREEN + "\n[+] Exiting..." + Style.RESET_ALL)
            break

        else:
            print(Fore.RED + "[!] Invalid choice! Please enter 1, 2, or 3." +
Style.RESET_ALL)

if __name__ == "__main__":
    main()
```

5. Execution Workflow

The tool follows a straightforward execution flow:

1. **Initialization**: `main.py` displays a banner and CLI menu.
2. **User Selection**:
    - Option to extract Chrome passwords.
    - Option to extract Firefox passwords.
    - Option to exit.
3. **Password Extraction**:
    - Based on the user's choice, `main.py` invokes `chrome.py` or `firefox.py`.
    - Credentials are retrieved, decrypted, and displayed to the user.
4. **Output**:
    - Passwords are displayed in a color-coded format, enhancing readability.
    - The tool exits upon user selection of the "exit" option.

6. Conclusion

The Browser Password Extraction Tool is a practical utility for accessing saved credentials in Chrome and Firefox. Each component functions cohesively, ensuring reliable extraction and user-friendly display of results. The modular structure allows for potential expansion to additional browsers or improved decryption techniques in future versions.

*****