

Auditable Credential-Exposure Checks on FreeBSD Without Sending Full Passwords

A practical, privacy-preserving range-lookup workflow with bounded disclosure, strict response handling, and useful audit evidence.

By Emre Çapan

Password-exposure checking is the process of checking whether a password someone intends to use has already appeared in a collection of exposed passwords. Services such as Have I Been Pwned's Pwned Passwords maintain searchable hashes of password values recovered from breaches and other compromise data. An application can consult that collection when a user

creates or changes a password and reject a candidate that is already known to attackers.

For example, a person may choose a password that is new to their account but has been used and exposed elsewhere. A match is a reason to choose a different password. It does not establish that this person's account was breached, identify a particular victim, or reveal where the password originated. A non-match means only that the value was not found in the checked collection; it is not a guarantee of strength or safety. This check supplements the surrounding authentication controls.

The lookup introduces a problem of its own: how can a client ask about a secret without giving that secret to the service answering the question? Sending a complete password over HTTPS still reveals it to the receiving service. Sending its complete unsalted hash also provides a verifier against which the service or an observer can test password guesses.

A range lookup limits that disclosure. The client calculates the password's SHA-1 hash locally and sends only its first five hexadecimal characters. The service returns the hash suffixes in that range, and the client performs the final comparison on the FreeBSD host. The complete password and complete hash are not sent to the range service. This is commonly called a k-anonymity lookup, but it is not complete anonymity: the service still sees the prefix, source IP address and request timing. SHA-1 serves only as the lookup format required by this particular corpus, not as a password-storage scheme.

This article builds an interactive FreeBSD checker around the public Pwned Passwords range interface. It follows the candidate from terminal entry through the HTTPS request, strict response validation and a minimal audit record. The implementation distinguishes a match, no match and an unavailable or invalid check, so a timeout never becomes a clean result. It also explains response padding, caching limits, testing, and when jails, Capsicum or an rc.d-managed worker are appropriate. The workflow is independent of any commercial product.

The real problem is the query, not the hash function

An exposure checker answers a narrow question: does a candidate password appear in a corpus of values previously observed in breaches or other public compromise data? It does not prove that an account using that password was breached. It does not identify where the password came from. It does not make a password safe when the answer is negative. The useful outcome is a decision signal: a positive match means the candidate should not be set or reused, while a negative match removes one known risk indicator but leaves ordinary password-quality and account-security controls in place.

The naive implementation sends the password to a remote service. HTTPS protects the connection in transit, but the service still receives the secret. A superficially improved implementation sends a complete SHA-1 digest. That avoids clear text on the wire, but the digest remains a stable verifier for guesses. Common passwords can be tested cheaply, and the service receives the exact lookup value.

A prefix-range protocol changes the disclosure. The client computes a 160-bit SHA-1 digest locally, sends the first five hexadecimal characters, and receives all known suffixes associated with that prefix. Five hexadecimal characters disclose 20 bits of the digest and select one of 1,048,576 possible ranges. The client compares the remaining 35 characters locally. The remote service learns the requested range, not the complete digest or the password. This pattern is commonly described as a k-anonymity range lookup. The label should not be mistaken for complete anonymity: the size of the returned group varies, and the service still observes the source and prefix.

SHA-1 is not suitable for storing passwords, signing data, or establishing collision resistance. FreeBSD's own `md5(1)` manual warns that SHA-1 is vulnerable to practical collision attacks [3]. In this workflow, SHA-1 is not a password-storage construction. It is a compatibility key for a public corpus that is already indexed by SHA-1. That distinction is essential. A local account database still requires a purpose-built, salted password hashing scheme, and

a new protocol should not choose SHA-1 merely because this range interface uses it.

Threat model and explicit limits

Before writing code, decide which observations and failures matter. The workflow in this article assumes the FreeBSD host is trusted at the moment of entry. If the host is compromised by a keylogger, a malicious kernel module, a hostile terminal, or a process with sufficient debugging privileges, a range lookup cannot protect the candidate. The workflow also assumes the configured HTTPS endpoint is the intended service and that the host's trust store and time are valid.

The design addresses the following risks:

1. **Disclosure through process arguments.**

The password is never accepted as a command-line argument. Arguments are visible through process inspection and are often copied into shell history, job records, or support transcripts.

2. **Disclosure through environment variables.**

The password is not supplied through the environment. Environment values can leak through diagnostics, process inspection under some privilege models, crash handling, and automation configuration.

3. **Disclosure to the range service.**

Only the five-character prefix leaves the host. The full digest and suffix remain local.

4. **Response-size observation.**

The request asks the service to pad its response. The Pwned Passwords documentation describes a target of 800 to 1,000 results and identifies synthetic entries with a count of zero [9]. Dense ranges can already contain more genuine records than that target, so a client must not impose a 1,000-line maximum. Padding reduces, but does not eliminate, information available from traffic size.

5. **Unsafe network behavior.**

The client allows HTTPS only, retains normal certificate and hostname verification, sets short connection and total timeouts, and treats network ambiguity as an unknown result rather than a clean password.

6. **Parser confusion.**

The response must match a narrow grammar before any line is trusted. Oversized, empty, malformed, or mixed-format bodies fail closed.

7. **Secret-bearing logs.**

Audit records omit the password, full digest, prefix, suffix, prevalence count, response body, and command output.

Several risks remain. The service sees the source IP address, request time, user agent, and 20-bit prefix. A network observer who can see only encrypted traffic still sees timing and approximate size. Repeated or incremental checks can create a sequence that narrows the secret. The HIBP documentation specifically warns against checking after each character is typed because an observer may combine those requests [9]. Wait until the candidate is complete and make one request.

A compromised service could return false data. TLS authenticates the endpoint named in the certificate, not the truth or completeness of its corpus. A negative result means only that the exact digest was not present in the response accepted at that moment. Availability failure, invalid TLS, a timeout, or a malformed body must produce an indeterminate result, never a negative one.

Finally, a shell process cannot promise deterministic erasure of a variable from all memory. Clearing and unsetting the variable shortens its useful lifetime but does not provide a formal zeroization guarantee. A higher-assurance client should use a small compiled program and clear sensitive buffers with a primitive such as FreeBSD's `explicit_bzero(3)`, which is designed not to be removed by compiler dead-store optimization [7].

Test target and dependencies

The original 31 August 2026 implementation was exercised on FreeBSD 15.1-RELEASE, with the base utility interfaces also checked against the 14.4 documentation. As of 10 September 2026, the FreeBSD security page lists 15.1-RELEASE, 15.0-RELEASE, 14.5-RELEASE and 14.4-RELEASE as supported [2]. FreeBSD 14.5 was released on 8 September; its inclusion in that list is not a claim that this example was tested on it. The release page distinguishes production and legacy releases from security support [1]. The examples use `base sh(1)`, `sha1(1)`, `mktemp(1)`, and `logger(1)`, with `curl` for one specific requirement: the range service's padding option is expressed as a custom HTTP request header. FreeBSD's `base fetch(1)` provides HTTPS retrieval and certificate controls, but its command-line interface does not provide a general custom-header option [4].

Install `curl` from the FreeBSD package repository:

```
# pkg install curl
```

The FreeBSD Handbook documents `pkg install curl` as the standard binary-package workflow [5]. The package brings its TLS dependencies through the package manager. Do not work around a trust-store problem with `curl's --insecure` option. Fix the trust store, system time, proxy interception, or endpoint configuration instead.

The script replaces the inherited search path with a short list of standard, administrator-controlled directories. It then resolves the package client and the base utilities that it retains in variables; other standard commands run through the same fixed path. This matters across the two target release families because the installed location of a base utility can change even when its command-line interface remains compatible. An administrator should verify all four resolutions with `command -v curl sha1 logger uuidgen` on the target release instead of silently substituting another program. Package paths belong under `/usr/local`; base-system utilities may be confirmed from the installed manual pages and filesystem. In particular, resolving `uuidgen` avoids depending on a release-specific absolute path.

The shell blocks in the next four steps are consecutive sections of one script. Assemble them in order. The first block contains the shebang and all setup; each later block continues in the same shell process.

Step 1: read the candidate without putting it in history

The entry wrapper accepts no positional arguments and requires an interactive terminal. It temporarily disables terminal echo, reads one completed line, restores the previous terminal state, and hashes the exact bytes before clearing the shell variable. It clears inherited variables used for secret material and disables shell tracing before reading the candidate. Do not run it through a debugger or tracing wrapper with real passwords.

```
#!/bin/sh
set +x
set -eu
umask 077

unset candidate digest prefix suffix count result http_code
result=

PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin
export PATH

CURL=$(command -v curl) || exit 69
SHA1=$(command -v sha1) || exit 69
LOGGER=$(command -v logger) || exit 69
UUIDGEN=$(command -v uuidgen) || exit 69

[ "$#" -eq 0 ] || {
    echo "usage: exposure-check" >&2
    exit 64
}
```

```

[ -t 0 ] || {
    echo "refusing non-interactive password input" >&2
    exit 64
}

old_tty=$(stty -g)
restore_tty() { stty "$old_tty" 2>/dev/null || true; }
trap restore_tty EXIT
trap 'restore_tty; exit 129' HUP
trap 'restore_tty; exit 130' INT
trap 'restore_tty; exit 143' TERM

printf "Candidate password: " >&2
stty -echo
IFS= read -r candidate || {
    printf "\ninput failed\n" >&2
    exit 65
}
restore_tty
trap - EXIT HUP INT TERM
printf "\n" >&2

digest=$(printf '%s' "$candidate" | "$SHA1" -q)
candidate=
unset candidate

```

Quoting is not cosmetic. `printf '%s' "$candidate"` preserves spaces and prevents wildcard expansion. Do not use `echo`, whose option and backslash handling varies with input and implementation. Do not use `sha1 -s "$candidate"`, because the password would then become an argument to another process.

This interface intentionally rejects a pipe. That choice makes the interactive tool hard to misuse from a CI system or cron job. An application integrating

the same engine should pass the candidate through an in-process API or a narrowly scoped file descriptor, not relax the wrapper to accept secrets through arguments.

Step 2: split the digest and make one bounded request

Normalize the digest to uppercase, confirm that it contains exactly 40 hexadecimal characters, then divide it into a five-character prefix and 35-character suffix. Only the prefix is interpolated into the URL.

```
digest=$(printf '%s'
"$digest" | tr '[:lower:]' '[:upper:]')

[ "${#digest}" -eq 40 ] || exit 70
case "$digest" in
    *[^0-9A-F]*) exit 70 ;;
esac

prefix=$(printf '%.5s' "$digest")
suffix=${digest#?????}
digest=
unset digest

response=$(mktemp -t exposure-check) || exit 70
trap 'rm -f "$response"' EXIT
trap 'exit 129' HUP
trap 'exit 130' INT
trap 'exit 143' TERM

sleep 1

http_code=
if ! http_code=("$CURL" -q \
```

```

--fail \
--silent \
--show-error \
--proto 'https' \
--tlsv1.2 \
--connect-timeout 5 \
--max-time 20 \
--max-filesize 1048576 \
--retry 2 \
--retry-delay 1 \
--retry-max-time 45 \
--user-agent 'freebsd-exposure-check/1.1' \
--header 'Add-Padding: true' \
--write-out '%{http_code}' \
--output "$response" \
"https://api.pwnedpasswords.com/range/$prefix")
then
    result=unknown
elif [ "$http_code" != 200 ]; then
    result=unknown
fi

prefix=
http_code=
unset prefix http_code

```

The `-q` option must be curl's first argument. It disables automatic loading of `curlrc` files, so a personal configuration cannot silently add tracing, redirect following or disabled certificate checks [10]. The `--proto` restriction permits HTTPS only. Redirect following is not enabled. The client captures the HTTP status separately and accepts exactly 200, so a 3xx response cannot be mistaken for range data. Curl verifies the peer certificate and hostname by default. `--tlsv1.2` establishes a minimum TLS version without disabling later versions supported by the client and server. The 20-second maximum applies to each transfer attempt. `--retry-max-time 45` prevents a new retry after the

retry window has elapsed, although an attempt begun before the limit can finish after it [10]. Two retries accommodate a transient failure without creating an unbounded loop.

The one-second delay is a local operational policy, not a claim that the example service requires it. Its documentation currently states that the Pwned Passwords endpoint has no rate limit [9]. A local ceiling is still useful because configuration mistakes, UI loops, and batch jobs can generate abusive traffic even when an upstream provider publishes no fixed quota.

Retry policy needs nuance. A timeout after a request was sent is not proof that the service did no work, but this operation is an idempotent GET, so a small retry count is acceptable. If the endpoint contract changes, review that assumption. Never retry indefinitely in an account-creation path, and never reinterpret exhausted retries as a negative match.

Step 3: validate before comparing

The documented SHA-1 range response consists of lines with a 35-character hexadecimal suffix, a colon, and a decimal prevalence count. Padding lines have the same shape and a count of zero [9]. Validate the complete response before searching it. Curl's `--max-filesize` option asks curl to fail when the response exceeds the ceiling. As its manual explains, curl may discover the size only during transfer when no usable size is declared, so this is a guard rather than a filesystem quota [10]. The post-download check rejects any empty or oversized body that remains. Together, these are application-level limits, not substitutes for filesystem quotas or broader resource controls.

```
if [ "${result:-}" !=
unknown ]; then
    bytes=$(wc -c < "$response" | tr -d ' ')
    [ "$bytes" -gt 0 ] && [ "$bytes" -le 1048576 ]
    || result=unknown
fi
```

```

if [ "${result:-}" != unknown ]; then
    if ! awk -F: '
        {
            sub(/\r$/, "")
            if (NF != 2 || length($1) != 35 ||
                $1 !~ /^[0-9A-F]+$ / || $2 !~ /^[0-9]+$ /) {
                exit 1
            }
        }
    END { if (NR == 0) exit 1 }
    ' "$response"
then
    result=unknown
fi
fi

```

```

if [ "${result:-}" != unknown ]; then
    if ! count=$(printf '%s\n' "$suffix" | awk -F: '
        FILENAME == "-" {
            if (FNR != 1 || length($0) != 35 ||
                $0 !~ /^[0-9A-F]+$ /) exit 2
            wanted=$0
            next
        }
        {
            sub(/\r$/, "")
            if ($1 == wanted && ($2 + 0) > 0) {
                print $2
                exit
            }
        }
    END { if (wanted == "") exit 2 }
    ' - "$response")
then

```

```
        result=unknown
    elif [ -n "$count" ]; then
        result=exposed
    else
        result=not_found
    fi
fi

suffix=
count=
unset suffix count
```

The shell's `printf` builtin passes the selected suffix to `awk` through a pipe. Do not use `awk -v wanted="$suffix"`: that would place a 140-bit password-derived value in the process argument list. A comparison-process failure is handled as `unknown`. Matching only a suffix with a positive count discards padded records. The exact prevalence count is deliberately not logged. A UI may choose to display it, but teams should first decide whether the number changes the remediation. Usually it does not. One observation is enough to reject or rotate a candidate, while a very large count can encourage false precision about corpus freshness and provenance.

This parser is strict about uppercase output because the example endpoint documents uppercase suffixes. A different provider may define another casing rule or hash family. Adapt the grammar to the provider's published contract rather than adding broad, forgiving conversions. A permissive parser can turn an HTML error page, proxy warning, or partially corrupted response into an incorrect security decision.

Step 4: give failures their own state

A security check needs three results, not two:

- `exposed`: the local suffix matched a validated positive-count record;

- not_found: the response was fetched, bounded, fully validated, and contained no positive match;
- unknown: the check could not establish either of the above.

Use separate exit codes so scripts cannot confuse operational failure with a safe result:

```
if !
event_id=$( "$UUIDGEN" -r ); then
    echo "Unable to create an audit event ID. Treat the result as
        unknown." >&2
    exit 20
fi

case "$result" in
    exposed)
        if ! "$LOGGER" -p auth.notice -t exposure-check \
            "event=$event_id outcome=exposed source=pwned-passwords-range
            client=freebsd-exposure-check/1.1"
        then
            echo "Audit logging failed. Treat the result as unknown." >&2
            exit 20
        fi
        echo "Match found. Do not set or reuse this password."
        exit 10
        ;;
    not_found)
        if ! "$LOGGER" -p auth.notice -t exposure-check \
            "event=$event_id outcome=not_found source=pwned-passwords-range
            client=freebsd-exposure-check/1.1"
        then
            echo "Audit logging failed. Treat the result as unknown." >&2
            exit 20
        fi
```

```

        echo "No match found in the checked corpus. This is not a safety
            guarantee."
    exit 0

    ;;

*)
    "$LOGGER" -p auth.warning -t exposure-check \
        "event=$event_id outcome=unknown source=pwned-passwords-range
        client=freebsd-exposure-check/1.1" || true
    echo "Check unavailable or invalid. Treat the result as unknown." >&2
    exit 20

    ;;

esac

```

FreeBSD's `logger(1)` is a shell interface to `syslog`, and its facility and priority can be routed through `syslog.conf(5)` [6]. The record above proves a more modest fact than many audit designs claim: a named client recorded an outcome at a time represented by the system log, with an opaque event identifier. It does not cryptographically prove which password was checked, that the service corpus was correct, or that the user acted on the result.

That limitation is beneficial. Logging the prefix would preserve 20 bits about the candidate. Logging the full digest would create an efficient offline guessing target. Logging a deterministic HMAC of the digest would enable correlation and would again become a verifier for anyone who later obtains the HMAC key. If a regulated workflow needs stronger evidence, bind the event to an internal account action by storing an unrelated application event ID and the policy decision, such as `password_change_rejected`, in the application audit system. Keep the secret-derived material out of the evidence record.

Log access and retention still matter. FreeBSD's logging framework can route local messages to files or remote collectors, and its BSM audit facility can capture fine-grained security events [11, 12]. More logs do not automatically produce more assurance. Decide who can read the event, how it is rotated, and how long the evidence remains useful. Avoid enabling high-volume auditing without a storage and review plan.

Caching without building a password oracle on disk

Range responses are shared by every password with the same prefix, so caching can reduce repeated network calls. It also changes the data model. A cache filename such as `21BD1.txt` records that someone on the host requested that prefix. The response contains hundreds of suffixes from a public corpus. Neither item is a clear-text password, but the request history may still be sensitive in a small environment or when combined with other observations.

For an interactive administrative tool, the simplest safe policy is no persistent cache. The temporary response is created under a restrictive `umask`, used once, and removed by a signal-aware trap. For a busy password-setting service, use an in-memory cache with a short time to live. If persistence is operationally necessary:

1. Store responses in a directory owned by the dedicated service account with mode `0700`;
2. Use prefix files only, never user IDs or account names in paths;
3. Apply a short, documented expiration period;
4. Validate cached content with the same grammar as network content;
5. Replace files atomically after a successful fetch;
6. Never store the selected suffix, full digest, candidate, or user-level match result;
7. Include cache age in monitoring, but not in the user-facing pass or fail decision unless policy defines a maximum age.

An offline copy of the entire public corpus eliminates query disclosure to the remote service but introduces a large dataset that must be downloaded,

updated, verified, stored, and searched. It is not automatically more private overall. The right choice depends on query volume, egress policy, storage controls, update discipline, and the sensitivity of local request metadata.

Jails, Capsicum, and rc.d: use the right boundary

FreeBSD jails virtualize filesystem, user, and networking access and can provide a useful layer around a long-running integration [13]. They do not make an unsafe client safe, and they are unnecessary for a single administrator typing a candidate into a short-lived local process. The highest-value controls remain simple: do not expose the secret through an argument, do not log secret-derived values, constrain outbound traffic, and fail closed.

A jail becomes reasonable when exposure checking is part of an organizational service. Run the queue consumer as a dedicated unprivileged account. Give the jail only the files it needs. Use a VNET jail or host firewall rules when independent egress control is required. FreeBSD includes PF, IPFW, and IPFILTER in the base system, and the Handbook describes using firewall rules to control both inbound and outbound traffic [14]. Limit the worker to DNS and HTTPS destinations required by policy, while planning for endpoint address changes and certificate validation.

Capsicum is another option for a purpose-built client. FreeBSD describes Capsicum as a capability and sandbox framework that restricts access to global namespaces after required descriptors have been opened [8]. A compiled checker could open its terminal, trust store, resolver channel, log socket, and output destination, then enter capability mode before parsing untrusted response data. That design needs careful decomposition because name resolution and new outbound connections normally involve global namespaces. It is a useful next step, not a shell-script toggle.

The same restraint applies to rc.d. FreeBSD's rc.d framework is designed to start and manage services through small `sh (1)` scripts using `/etc/rc.subr` [15]. The interactive checker is not a daemon and should not start at boot. If an application uses a persistent queue worker, install that worker under `/usr/local/libexec`, place a conventional control script under

`/usr/local/etc/rc.d`, disable it by default, and use `run_rc_command` through `rc.subr`. Keep password input inside the application boundary. The `rc.d` script should manage only the worker lifecycle, never contain credentials or candidate values.

A reproducible test plan

Test the pure parsing and decision logic separately from secret entry. Do not use a real account password in development. Use intentionally public fixtures or randomly generated synthetic values that are never assigned to an account.

1. Known positive

Use a publicly documented demonstration string, not a private credential. Confirm that the local SHA-1 is 40 hexadecimal characters, the request contains only five of them, the validated response contains the matching suffix with a positive count, the program exits 10, and the audit record contains no secret-derived field.

2. Expected non-match

Generate a long random synthetic value locally, check it once, and expect `not_found`. Because any value could theoretically appear in the corpus, the assertion should be worded as an expected fixture result and pinned to a captured, validated response for unit testing. The live integration test may change over time.

3. Padded zero-count entry

Feed the parser a valid 35-character suffix with count zero. Confirm that it is ignored. Then include the same suffix with a positive count in a separate fixture and confirm that it matches.

4. Malformed bodies

Exercise an empty file, an HTML page, a suffix of the wrong length, non-hexadecimal characters, a missing colon, a negative count, an extra field, and a body larger than the configured ceiling. Every case must produce `unknown` and exit 20.

5. Network and TLS failure

Test connection refusal, DNS failure, timeout, an untrusted certificate in a controlled test environment, and an HTTPS redirect to an HTTP URL. None may produce `not_found`. Verify that the error shown to the user does not contain the digest or response body.

6. Terminal interruption

Press Control-C while entering the candidate and while the request is in progress. Confirm that echo is restored and the temporary file is removed. Send HUP and TERM in a test shell and repeat the check.

7. Audit review

Inspect the configured log destination. The accepted schema should contain only an event ID, outcome, source identifier, system time supplied by the logging path, and perhaps the client version. Search the test log for the candidate, digest, prefix, suffix, prevalence count, and response lines. All searches must be empty. Also inspect child-process arguments and exported variables using a synthetic candidate: neither the candidate, complete digest nor selected suffix should appear. The five-character prefix in curl's request URL remains observable to local process inspection; that is part of the bounded disclosure, not a promise to hide the request from the host.

8. Concurrency and throttling

If integrating into a service, start concurrent checks and confirm that the shared limiter caps aggregate traffic, not merely traffic per process. Verify

that cache replacement is atomic and that a failed refresh leaves either a still-valid previous entry or an explicit unknown state.

The editor or reviewer should be able to reproduce parser tests without calling the live service. Keep small synthetic response fixtures in the source tree and make the network integration test opt-in. That split makes ordinary test runs fast, deterministic, and respectful of the public endpoint.

Operational policy matters more than the lookup

A well-built range check can still support a bad security policy. The control belongs at password creation and change, before the candidate is committed. When a match is found, reject the candidate and explain that it has appeared in a known corpus. Do not reveal breach records or imply that the user's specific account was compromised. Pair the control with rate limiting on the surrounding account workflow, multi-factor authentication, secure recovery, session review, and monitoring for account takeover signals.

For authorized organizational use, define scope explicitly. Check candidates supplied during your own authentication workflow or assets that the organization is legally authorized to manage. Do not collect passwords from unrelated users, test third-party accounts, or turn the range endpoint into an enumeration tool.

Review the endpoint contract periodically. Prefix length, padding behavior, response grammar, acceptable-use rules, and client requirements can change. Pinning assumptions in code without monitoring documentation creates a quiet failure mode. Record the client version in audit events, test after package upgrades, and treat an unexpected response as a reason to stop, not to relax validation.

Conclusion

Privacy-preserving credential-exposure checking is not achieved by hashing a password and calling an API. It comes from the complete data path: how the candidate enters the process, what leaves the host, how the network

request is constrained, how untrusted data is parsed, what is cached, which outcomes exist, and what the audit trail refuses to retain.

On FreeBSD, a small implementation can use familiar system components. `sha1(1)` supplies the corpus-compatible lookup digest, `mktemp(1)` creates a private response file, `logger(1)` records a minimal event, and the package system supplies `curl` for a padded HTTPS request. Jails, firewalls, Capsicum, and `rc.d` become relevant when the checker grows into a service, but they should reinforce a sound protocol rather than compensate for secret-bearing arguments or logs.

The most important result is `unknown`. A system that distinguishes uncertainty from absence can fail safely. A system that collapses timeouts and parser errors into “not found” eventually tells someone that an unchecked password is clean. Preserving that third state, while disclosing only a bounded prefix and retaining no reusable verifier, turns a clever range query into an auditable security control.

References

1. The FreeBSD Project, “Release Information.”

<https://www.freebsd.org/releases/>

2. The FreeBSD Project, “FreeBSD Security Information: Supported FreeBSD Releases.” <https://www.freebsd.org/security/>

3. The FreeBSD Project, `md5(1)`, including `sha1(1)`.

<https://man.freebsd.org/cgi/man.cgi?query=md5&sektion=1>

4. The FreeBSD Project, `fetch(1)`.

<https://man.freebsd.org/cgi/man.cgi?format=html&query=fetch&sektion=1>

5. The FreeBSD Documentation Project, “Installing Applications: Packages and Ports.” <https://docs.freebsd.org/en/books/handbook/ports/>

6. The FreeBSD Project, `logger(1)`.

<https://man.freebsd.org/cgi/man.cgi?format=html&query=logger&sektion=1>

7. The FreeBSD Project, `explicit_bzero(3)` .

https://man.freebsd.org/cgi/man.cgi?format=html&query=explicit_bzero§ion=3

8. The FreeBSD Documentation Project, "Security: Capsicum."

<https://docs.freebsd.org/en/books/handbook/security/#capsicum>

9. Have I Been Pwned, "API Documentation: Pwned Passwords."

<https://haveibeenpwned.com/API/v3>

10. The curl Project, "curl(1) Manual." <https://curl.se/docs/manpage.html>

11. The FreeBSD Documentation Project, "Configuration, Services, Logging and Power Management: Configuring System Logging."

<https://docs.freebsd.org/en/books/handbook/config/#configtuning-syslog>

12. The FreeBSD Documentation Project, "Security Event Auditing."

<https://docs.freebsd.org/en/books/handbook/audit/>

13. The FreeBSD Documentation Project, "Jails and Containers."

<https://docs.freebsd.org/en/books/handbook/jails/>

14. The FreeBSD Documentation Project, "Firewalls."

<https://docs.freebsd.org/en/books/handbook/firewalls/>

15. The FreeBSD Documentation Project, "Practical rc.d Scripting in BSD."

<https://docs.freebsd.org/en/articles/rc-scripting/>

Disclosure

The author is affiliated with CyberVisir Solutions Ltd., operator of LeakData.io. The workflow is product-independent; the company reference identifies the author's affiliation. Public documentation and synthetic test data were used;

no private breach records or live credentials are reproduced. AI tools assisted with research organization, language editing and code review. The original implementation was exercised on FreeBSD 15.1-RELEASE; the September revision adds the clarifications and implementation corrections described above.

Emre Çapan is Co-Founder and CTO of LeakData (<https://leakdata.io>) at CyberVisir Solutions Ltd. His work focuses on digital-risk monitoring, exposure triage, privacy-aware security workflows, and turning uncertain security signals into practical remediation steps.

Copyright © 2026 Held by Owner/Author. Publication Rights Licensed to The FreeBSD Journal. This work is licensed under a [Creative Commons Attribution International 4.0 License](#).

3980 Broadway Street
STE #103-107
Boulder CO 80304
1-720-207-5142
info@freebsd.foundation.org

