hx

Hash Expression Language Specification

Waffle
September 2026

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

hx is a domain-specific language for describing, computing, and verifying cryptographic hash compositions. It provides a precise, human-readable notation for expressing the exact sequence of operations that produce a hash value from an input password, salt, and optional additional parameters.

The language was designed to satisfy three requirements that emerged from over a decade of work with mdxfind, hashpipe, and the broader password security research community:

1. **Reproducibility**— every hash result that mdxfind reports must be independently verifiable. If mdxfind matches a truncated SHA-512 digest against a 32-character hex string, the hx expression `cut(sha512(pass), 0, 32)` describes exactly what was computed. Given the same password, evaluating that expression must produce the identical 32-character string.
2. **Generality**— the notation must be capable of expressing every hash type that mdxfind supports (approximately 1000 types as of 2026), as well as compositions that have not yet been assigned type numbers. A user encountering a novel hash scheme should be able to describe it in hx without waiting for a new type to be added to mdxfind.
3. **Clarity**— the notation must be readable by humans who are not cryptographers. The expression `sha1(md5(pass) . salt)` should be immediately understandable as "compute the MD5 of the password, concatenate the salt, then compute the SHA-1 of the result."

hx is not a general-purpose programming language. It is deliberately constrained to operations relevant to hash computation: cryptographic hash functions, string concatenation, encoding transformations, iteration, and simple control flow. It is an interpreted language, and makes no attempt to compete with mdxfind or hashcat in raw throughput. Its value lies in correctness, expressiveness, and the ability to serve as a canonical reference notation for hash computations.

1.1 Scope and Boundaries

hx is designed to express *compositions of cryptographic hash primitives* — algorithms built by chaining, concatenating, iterating, and encoding the outputs of hash functions. This covers the vast majority of password hashing schemes encountered in practice: simple hashes, salted hashes, iterated hashes, compound multi-algorithm chains, and HMAC (Hash-based Message Authentication Code) constructions.

Key derivation functions (KDFs) such as bcrypt, scrypt, yescrypt, Argon2, and PBKDF2 lie outside the expressive scope of hx's composition model. These algorithms have internal state machines (Blowfish key schedule expansion, memory-hard sequential access patterns, configurable parallelism) that cannot be decomposed into a sequence of hash-function calls. They are registered in hx as *opaque built-in functions* — callable by name (**bcrypt()**, **scrypt()**, **pbkdf2_sha256()**) but not expressible as hx programs. The appendix identifies these types and notes their built-in status.

This is a deliberate design boundary, not a gap. A language that attempted to express bcrypt's internal Blowfish rounds or scrypt's memory-access pattern would require general-purpose byte manipulation, bitwise operations, and array indexing, defeating the simplicity and readability that are hx's primary design goals. The practical consequence is small: KDFs are a closed set of well-known algorithms, each with a standardized specification and mature implementations. Expressing their internals as hx programs would add complexity without adding insight.

hx operates on **raw bytes**. The **pass** variable contains exactly the byte sequence provided on the command line or read from standard input, with no character encoding conversion and no Unicode normalization applied. When a hash scheme requires a specific encoding (e.g., UTF-16LE for NTLM), the encoding must be expressed explicitly in the hx program using a transform function such as **utf16le()**. The language itself is encoding-agnostic: a password is a sequence of bytes, and a hash function receives a sequence of bytes.

1.2 Origin

The immediate impetus for hx was a problem encountered during analysis of large hash collections with mdxfind. When mdxfind searches a file of 32-character hex hashes, it tests every supported algorithm that produces a hex string of that length. A match against **SHA512** is reported even though SHA-512 normally produces 128 hex characters — mdxfind matched only the first 32 characters, a prefix collision.

The question that followed was deceptively simple: "how do I verify this match?" No existing tool could take the description "first 32 hex characters of the SHA-512 of this password" and compute the result. The user would need to write a small program, or chain together command-line tools with careful attention to encoding at each step.

This is the problem hx solves. The expression `cut(sha512(pass), 0, 32)` is a complete, unambiguous specification that can be evaluated by the hx interpreter to produce the exact output that mdxfind matched. From this starting point, the language grew to encompass salted hashes, iterated hashes, compound hashes with binary intermediates, and ultimately any composition of cryptographic primitives that the password hashing world has devised.

1.3 Relationship to Existing Tools

hx exists within the mdxfind tool ecosystem and is designed to complement, not replace, the existing tools:

- **mdxfind** performs high-speed hash searching across hundreds of algorithm types on CPU and GPU. The hx language describes exactly what mdxfind computes, in a form that can be independently verified. An hx expression can be mapped to an mdxfind type number (e.g., `sha1(md5(pass) . salt)` corresponds to **mdxfind −m e587**), connecting the readable notation to the optimized engine.
- **hashpipe** is a hash verification tool. When hx is integrated into hashpipe (via the **−X** flag), all of hashpipe's 500+ hash function implementations become available as hx primitives. This is the recommended way to use hx for any work beyond the basic hash types.
- **mdsplit** separates mdxfind output by type. The type names that mdsplit produces (e.g., SHA1MD5PASSSALT) correspond directly to hx expressions, providing a translation path from mdxfind's compressed type notation to the fully explicit hx form.

1.4 Modes of Use

hx operates in three modes of increasing complexity:

1. **Calculator mode.** A single expression on the command line, applied to one password or to a stream of passwords on standard input. This is the most common use: generating test vectors, verifying individual hashes, or exploring a hash scheme.

   ```
   hashpipe −X 'sha256(pass . salt)' −p password −s 12345
   ```

2. **Script mode.** An hx program is read from a file, allowing multi-statement computations with variables, loops, and conditionals. This mode is required for hash schemes with complex iteration patterns.

   ```
   hashpipe −F phpbb3.hx −p password −s saltsalt
   ```

3. **Batch mode.** In either calculator or script mode, passwords may be read from standard input (one per line) instead of being specified with **−p**. The hx program is compiled once and executed for each input password.

   ```
   cat wordlist.txt | hashpipe −X 'md5(pass . salt)' −s 12345
   ```

2. Language Specification

This section defines the hx language: its lexical structure, grammar, type system, evaluation rules, and built-in functions. The specification is intended to be precise enough that an independent implementation could produce identical output for any well-formed hx program.

2.1 The hx Algebra

Every hx expression is built from five kinds of element:

- **Variables** — the read-only inputs (**pass**, **salt**, **salt2**, **pepper**, **user**) and user-defined assignments.
- **Literal strings** — fixed byte sequences written in single or double quotes. Literals serve as HMAC keys, format prefixes (**"H9"**), fixed delimiters, and any other byte sequence the algorithm specifies in its definition.
- **Integer literals** — used as iteration counts, loop bounds, and positional arguments to functions such as **cut()**, **bcrypt()**, and the PBKDF2 family.
- **Function calls** — the computational primitives. A function takes one or more string (and occasionally integer) arguments and produces a single string result.
- **The concatenation operator .**, which joins two byte sequences end to end. Concatenation has the lowest expression precedence; function calls bind more tightly than concatenation.

Function composition is the central operation. The output of one function call may be supplied as the input to another: `sha1(md5(pass))` computes MD5 of the password and then SHA-1 of that result. Composition is unrestricted: any function output may feed any function input, and arbitrary nesting depth is permitted. The encoding contract (Section 2.5) determines whether the inner output is delivered to the outer call as canonical text (the default) or as raw bytes (when the inner call carries a **_bin** suffix).

Output-role suffixes (**_bin**, **_hex**, **_uc**, **_b64**, **_mcf**) are attached to a function name to select a non-default representation of its output. The five suffixes form a small algebra of their own: every function in the registry computes a canonical result; the suffix controls how that result is encoded on the way out. A suffix that does not apply to a given function family is a compile-time error.

The emit() function provides multi-value output. A single hx program may **emit()** zero or more intermediate values during evaluation; the program's final expression result is also emitted unless an earlier **emit()** call suppresses the automatic emission. This algebra is sufficient to describe hash schemes that produce several output forms from a single password — the **SHA1MD5HUM** family, the **PHPBB3** variants, the multi-form **SQL5** output, and any future scheme that follows the same pattern.

The remainder of this chapter formalises the syntax and semantics of these constructs.

2.2 Lexical Structure

An hx program is a sequence of Unicode characters encoded as UTF-8. The lexer partitions the input into tokens of the following classes:

2.2.1 Identifiers

An identifier begins with a letter (a–z, A–Z) or underscore, followed by zero or more letters, digits, or underscores:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

Identifiers serve as both variable names and function names. The same identifier may refer to a variable or a function depending on context: if followed by **(** or ^ it is a function call; otherwise it is a variable reference.

Hyphens are not permitted in identifiers. Hash type names that contain hyphens (such as GOST-CRYPTO or SHA3-256) use underscores in hx: `gost_crypto` and `sha3_256`. Hashpipe automatically registers both the original hyphenated name (for internal lookup) and the underscore variant (for use in hx expressions).

2.2.2 Reserved Words

The following identifiers are reserved and may not be used as variable names or function names:

Word	Meaning
pass	current password
salt	primary salt value
salt2	secondary salt value
pepper	pepper value
user	user or userid value
for	loop statement
to	loop bound keyword
if	conditional statement
else	conditional alternative

TABLE 1. Reserved Words

The variables **pass**, **salt**, **salt2**, **pepper**, and **user** are pre-populated from the command-line arguments (**−p**, **−s**, **−S**, **−P**, **−u**), and are read-only; they cannot appear on the left side of an assignment.

2.2.3 Integer Literals

An integer literal is a sequence of one or more digits, optionally preceded by a minus sign:

```
42
1000
-3
```

Integer values are used for iteration counts in the ^ operator, loop bounds in **for** statements, and numeric arguments to functions such as **cut()**.

2.2.4 String Literals

A string literal is enclosed in double quotes or single quotes. Standard backslash escapes are recognized within double-quoted strings:

```
"hello world"
"prefix\tsuffix"
```

String literals produce byte sequences. They are most commonly used for hash format prefixes and fixed delimiters in output construction.

2.2.5 Operators and Delimiters

The following single and multi-character tokens are recognized:

Token	Meaning
.	concatenation
^	iteration
=	assignment
,	argument separator
( )	grouping and function call
{ }	block delimiters
==	equality comparison
!=	inequality comparison
< > <= >=	relational comparison
;	statement separator

TABLE 2. Operators and Delimiters

2.2.6 Statement Separators

Statements are separated by newlines or semicolons. Both are equivalent and interchangeable. Multiple consecutive separators (blank lines) are permitted and ignored.

Inside braces ({ ... }), newlines are treated as whitespace, not as statement separators. Within a block, statements must be separated by semicolons:

```
for i = 1 to 10 {
    h = md5(h); k = sha1(k)
}
```

This rule has one practical consequence: the **else** keyword of an **if** statement must appear on the same line as the closing brace of the then-block:

```
if h == target { x = 1 } else { x = 0 }
```

2.2.7 Comments

A comment begins with **#** and extends to the end of the line. Comments are ignored by the lexer:

```
# This is a comment
h = md5(pass)   # inline comment
```

2.3 Grammar

The following grammar defines the syntax of hx programs in extended BNF (Backus–Naur Form) notation. Terminal symbols are shown in **bold**; non-terminals in *italic*. Optional elements are enclosed in brackets; repetition is indicated by braces.

program	*sep_opt stmt_list sep_opt*
stmt_list	*stmt* { *seps stmt* }
stmt	*assignment* | *for_stmt* | *if_stmt* | *expr*
assignment	**IDENT =** *expr*
for_stmt	**for IDENT =** *expr* **to** *expr block*
if_stmt	**if** *condition block* [**else** *block*]
condition	*expr relop expr*
relop	**==** | **!=** | **<** | **>** | **<=** | **>=**
block	**{** *sep_opt stmt_list sep_opt* **}**
expr	*primary* | *expr* **.** *primary*
primary	**IDENT** | **reserved_var** | **NUMBER**
	| **STRING** | *funcall* | **(** *expr* **)**
funcall	**IDENT (** *arglist* **)**
	| **IDENT ^ NUMBER (** *arglist* **)**
arglist	*expr* { **,** *expr* }
seps	**SEP** { **SEP** }
sep_opt	[*seps*]

Figure 1. hx Grammar in Extended BNF

The concatenation operator **.** is left-associative and has the lowest precedence among expression operators. Function calls bind more tightly than concatenation.

The grammar is implemented using GNU Bison (LALR(1) parser generator) and Flex (lexical analyzer generator). Three shift/reduce conflicts arise from the interaction of optional leading separators with statement lists; all three are resolved correctly by Bison's default shift preference.

2.4 Type System

hx has two value types: **strings** (byte sequences) and **integers**. There is no Boolean type; conditions are evaluated by comparison operators that consume two values and branch accordingly.

All hash function inputs and outputs are strings. A string is an arbitrary sequence of bytes with an associated length; it is not required to be valid UTF-8 or null-terminated. Binary data (raw hash digests) and text data (hex-encoded hashes, passwords, salts) are both represented as strings. The language makes no distinction between them at the type level; the distinction is maintained by convention and by the **_bin** suffix on function names.

Integers appear only in three contexts: iteration counts in the ^ operator, loop bounds and loop variables in **for** statements, and numeric arguments to functions such as **cut()**. Integer values are 64-bit signed quantities.

2.5 The Encoding Contract

The most important design decision in hx is the default encoding of hash function outputs. This choice permeates the entire language and determines whether compound hash expressions produce correct results.

The rule is simple: every hash function, HMAC, KDF, or crypt built-in returns its *canonical traditional form* by default. For fixed-length digests — **md5**, **sha1**, **sha256**, **sha512**, **md4**, and the rest of the plain hash and HMAC families — the canonical form is **lowercase hexadecimal**. For variable-length key derivation functions — **pbkdf2_***, **scrypt**, **argon2id**, **argon2i**, **argon2d** — the canonical form is **hex of the derived bytes** (length = 2 × dklen). For structured-output crypt built-ins — **bcrypt**, **yescrypt**, the **md5crypt** family, **phpass** — the canonical form is the full **MCF (Modular Crypt Format) or PHC (Password Hashing Competition) string** as it would appear in a shadow file (**$2b$12$...**, **$y$j9T$...**, **1salt$hash**, **$H$9...**, etc.).

This rule was chosen because it matches the convention used by the password hashing community. When a security researcher writes `md5(md5($pass))` they mean: compute the MD5 of the password, encode the 16-byte result as 32 lowercase hex characters, then compute the MD5 of those 32 characters. The outer MD5 receives a 32-byte ASCII input, not a 16-byte binary input. This is what `md5(md5(pass))` computes in hx. By the same logic, a security researcher writing `bcrypt(pass, salt, 12)` means the full 60-character **$2b$12$...** string — not the 31-character hash portion, and not the raw 23-byte blowfish output.[1]

2.5.1 Output-role suffixes

To select a non-default representation of a function's output, attach one of five *output-role suffixes* to the function name. The suffix is stripped at compile time and sets a flag on the **CALL** instruction; the base function always computes the same underlying result, and the suffix controls only how that result is returned.

Suffix	Role	Meaning
(none)	canonical	Traditional form for this family.
_bin	binary	Raw key material; digest bytes or derived bytes, before any encoding.
_hex	hex	Lowercase hex of `_bin`.
_uc	uppercase hex	Uppercase hex of `_bin`. Distinct from **upper()** under iteration: the role is what feeds the next round, so `md5_uc^N` chains uppercase while `upper(md5^N)` uppercases only the final result. The two agree at N=1 and diverge from N=2 onward.
_b64	base64	Native base64 of the hash portion: bcrypt-base64 for **bcrypt**, crypt-base64 for the **md5crypt** family, standard base64 elsewhere.
_mcf	MCF/PHC	Full crypt string. Defined only for families that have a standard MCF form; compile-time error otherwise.

TABLE 3. Output-role suffixes

For fixed-length digests, the default is identical to **_hex**; the invariant `md5(md5(pass)) == md5(md5_hex(pass))` holds, and both are distinct from `md5(md5_bin(pass))` . For crypt built-ins the default is identical to **_mcf**; `bcrypt(pass, salt, 12) == bcrypt_mcf(pass, salt, 12)` .

A worked example covers all five variants:

Expression	Returns
`bcrypt(pass, salt, 12)`	60-char $2b$12$<salt22><hash31>
`bcrypt_mcf(pass, salt, 12)`	same as above (alias)
`bcrypt_b64(pass, salt, 12)`	31-char bcrypt-base64 hash portion
`bcrypt_bin(pass, salt, 12)`	23 raw bytes of blowfish output
`bcrypt_hex(pass, salt, 12)`	46-char hex of the 23 raw bytes

TABLE 4. bcrypt output-role variants

Mixed chains work naturally. `sha1(bcrypt_bin(pass, salt, 12))` feeds 23 raw bytes into SHA-1 and returns a 40-character hex digest. `sha1(bcrypt(pass, salt, 12))` feeds the full 60-character **$2b$12$...** string into SHA-1 — a different hash, and almost certainly not what was intended.

Encoding can also be made explicit using the **hex()** function. The expression `hex(md5_bin(pass))` is equivalent to `md5(pass)` ; it takes the raw binary MD5 output and encodes it as hex. This explicit form is useful when constructing output formats or when clarity is more important than brevity.

---

1. The internal blowfish state is 24 bytes wide, but the canonical OpenBSD implementation encodes only 23 of those bytes into the **$2b$** hash portion to remain bug-compatible with the original implementation. The externally accessible raw form is therefore 23 bytes, not 24.

2.5.2 Scope of the output-role suffixes

The five suffixes are defined for *hash functions*, *HMAC functions*, *key derivation functions*, and *crypt built-ins* — any function whose output has a well-defined canonical form that a caller might want to override. Not every suffix is valid on every family. In particular, **_mcf** is valid only on families that produce an MCF/PHC string (**bcrypt**, **yescrypt**, the **md5crypt** family, **phpass**), and **_b64** is valid only where a family has a well-defined native-base64 hash portion. Applying an unsupported suffix is a compile-time error, the same rule that rejects **upper_bin** or **base64_bin**.

Applying any output-role suffix to a string transform function (**upper**, **lower**, **rev**, **cut**) or a standalone encoding function (**base64**, **phpass_encode**) is a compile-time error. These functions do not produce a canonical hash output and have no encoding step to redirect.

The **hex()** function is a special case. `hex_bin(x)` is defined as the identity function: it returns its input unchanged. This is consistent with the semantics of **_bin** as "skip the encoding step" — **hex()** *is* the encoding step, so skipping it returns the original input. This identity is occasionally useful in generic expressions but should not be relied upon for clarity.

2.5.3 Default output form by family

The following table summarizes the canonical return form for each built-in family. All entries in this table are implemented as hx built-ins.

Family	Default return	Example
Fixed-length digest (md5, sha1, sha256, sha512, md4, ...)	lowercase hex of digest	`sha1(pass)` = 40 hex chars
HMAC (hmac_sha1, hmac_sha256, hmac_sha512, ...)	lowercase hex	`hmac_sha256(k,m)` = 64 hex
PBKDF2 (pbkdf2_sha1, sha256, sha512, md5)	hex of derived bytes	`pbkdf2_sha256(p,s,1000,32)` = 64 hex
scrypt	hex of derived bytes	`scrypt(p,s,N,r,p,dk)` = 2×dk hex
argon2 (argon2i, argon2d, argon2id)	hex of derived bytes	`argon2id(p,s,t,m,p,dk)` = 2×dk hex
bcrypt	$2b$...$ MCF (60 chars)	`bcrypt(p,s,12)` = $2b$12$<22><31>
yescrypt	y...$ MCF string	`yescrypt(p,s,...)` = yj9T$salt$hash
md5crypt	1...$ MCF	`md5crypt(p,s)` = 1salt$hash
apr1	$apr1$...$ MCF	`apr1(p,s)`
sha256crypt	5...$ MCF	`sha256crypt(p,s,r)`
sha512crypt	6...$ MCF	`sha512crypt(p,s,r)`
sm3crypt	$sm3$...$ MCF	`sm3crypt(p,s,r)`
gost12_512crypt	$gost12512$...$ MCF	`gost12_512crypt(p,s,r)`
descrypt	13-char salt+hash	`descrypt(p,s)` = sshash
phpass	H...$ MCF (34 chars)	`phpass(p,s,N)` = H9<salt><hash>
sha1crypt	$sha1$...$ MCF	`sha1crypt(p,s,r)`
bsdicrypt	_<rounds><salt> <hash>	`bsdicrypt(p,s)`
gost_yescrypt	gy...$ MCF	`gost_yescrypt(p,s)`

Family	Default return	Example
sevenzip	$7z$ structure (10 or 12 fields)	`sevenzip(p,s)`
cmiyc	$cmiyc$...$	`cmiyc(p,s)`

TABLE 5. Default output form of each built-in family

2.6 Evaluation Model

An hx program is processed in three phases: parsing, compilation, and execution.

2.6.1 Parsing

The input text is tokenized by the Flex-generated lexer and parsed by the Bison-generated parser into an abstract syntax tree (AST). Each node in the AST represents a language construct: literal value, variable reference, function call, concatenation, assignment, loop, conditional, or block.

The parser handles the brace-depth-aware newline rule: the lexer tracks the nesting depth of **{** ... **}** blocks and emits newlines as separator tokens only at depth zero.

2.6.2 Compilation

A single recursive walk of the AST produces a flat array of bytecode instructions. During this walk, the compiler:

- Resolves variable names to integer slot indices. The four built-in variables occupy fixed slots: **pass** = 0, **salt** = 1, **salt2** = 2, **pepper** = 3. User-defined variables are assigned sequential slots starting at 4.
- Resolves function names to entries in the function registry. If the name ends in **_bin**, the suffix is stripped and the *binary* flag is set on the **CALL** instruction. If the base name is not found in the registry, compilation fails with an error message identifying the unknown function and the source line.
- Collects string constants into a deduplication table.
- Emits jump instructions for **for** loops and **if** conditionals, with forward references patched after the target address is known.
- Tracks stack depth to determine the maximum stack size needed at runtime.

The compiler produces a *program* structure containing the instruction array, the string constant table, the variable name table (for diagnostics), and the computed maximum stack depth.

2.6.3 Execution

The bytecode is executed by a stack-based virtual machine. The VM maintains:

- A *stack* of values (strings and integers), used for expression evaluation.
- A *variable*array indexed by slot number, pre-populated with the built-in variables before each run.
- A *bump arena* allocator for all per-password string data. The arena is reset (rewound to offset zero) before each password, ensuring that no memory is leaked across runs and that allocation is a single pointer increment.

The VM executes instructions in a loop with a **switch** dispatch on the opcode. The instruction set is:

Opcode	Operand	Effect
PUSH_VAR	slot	Push variable onto stack
PUSH_STR	index	Push string constant
PUSH_INT	value	Push integer
STORE	slot	Pop stack top into variable
CALL	entry, nargs, bin	Pop args, call function, push result
CONCAT	—	Pop two values, push concatenation
HALT	—	Stop; stack top is result
JUMP	address	Unconditional jump
INC	slot	Increment integer variable
JUMP_LE	address	Pop two integers, jump if a ≤ b
JUMP_EQ	address	Pop two values, jump if equal
JUMP_NE	address	Pop two values, jump if not equal
DUP	—	Duplicate stack top
POP	—	Discard stack top

TABLE 6. Bytecode Instruction Set

The **CALL** instruction carries a direct pointer to the function entry in the registry, resolved at compile time. No name lookup occurs during execution. The *binary* flag is also encoded in the instruction, so the **_bin** decision is made once at compile time, not per-password.

The result of a program is the value on top of the stack when the **HALT** instruction is reached. In calculator mode, this value is written to standard output, followed by a newline. If the result is a binary string (from a **_bin** function), the raw bytes are written directly; the user is responsible for applying **hex()** or another encoding function if text output is desired.

2.7 Diagnostics and Error Handling

hx defines the following error conditions. A conforming implementation must report these errors and must not silently produce incorrect output.

2.7.1 Compile-Time Errors

The following conditions are detected during compilation and prevent execution:

- **Unknown function**— the identifier in a function call does not match any entry in the function registry (after stripping **_bin**, if present). The error message must identify the function name and the source line.
- **Invalid _bin application**— the **_bin** suffix is applied to a function that is not a hash function or HMAC function (see Section 2.4.1).
- **Syntax error**— the input does not conform to the grammar defined in Section 2.2. The error message should identify the source line and the unexpected token.

2.7.2 Runtime Behavior

The following conditions may arise during execution. hx defines their behavior to ensure reproducibility:

- **Empty strings**— an empty password, salt, or intermediate value is a valid zero-length byte sequence. Hash functions accept zero-length input and produce their standard digest (e.g., `md5("")` produces `d41d8cd98f00b204e9800998ecf8427e`).
- **cut() bounds clamping**— if the *start* argument exceeds the string length, the result is an empty string. If *start* + *length* exceeds the string length, the result is truncated to the available bytes. Negative *start* values that resolve to a position before the beginning of the string are clamped to zero. These are not errors.
- **for loop with start > end**— if the start value exceeds the end value, the loop body is not executed. This is not an error.
- **Integer overflow**— iteration counts and loop bounds are 64-bit signed integers. Values exceeding $2^{63} - 1$ produce undefined results. In practice, iteration counts above 2^{32} are computationally infeasible and are likely user errors.
- **Stack overflow**— the maximum stack depth is computed at compile time. Programs that exceed implementation limits (deeply nested function calls) may be rejected at compile time or may produce undefined behavior at runtime.
- **Binary data in text transforms**— applying **upper()** or **lower()** to binary data (e.g., the output of a **_bin** function) operates on individual bytes using the C locale. Bytes that are not ASCII letters are passed through unchanged. This is defined behavior, not an error.

2.8 Program Structure

An hx program is a sequence of statements. The last expression evaluated becomes the program's result.

A single-expression program:

```
sha256(pass . salt)
```

A multi-statement program with the result on the last line:

```
h = md5_bin(salt . pass)
for i = 1 to 2047 {
    h = md5_bin(h . pass)
}
hex(h)
```

The final `hex(h)` is an expression statement; its value is left on the stack and becomes the program output.

3. Functions

Functions are the computational primitives of hx. Every function takes one or more string arguments and produces a string result. Functions are registered in a flat table and looked up by name at compile time.

3.1 Hash Functions

When hx is used via **hashpipe −X** or **hashpipe −F**, every hash type registered in hashpipe's type table is available as an hx function. The function name is the mdxfind type name, lowercased, with hyphens replaced by underscores.

The following table lists the hash functions corresponding to mdxfind types e1 through e31 (the primitive hash algorithms), plus selected higher-numbered types. All functions accept the **_bin** suffix for raw binary output.

Function	Digest bytes	Hex chars	mdxfind type
md5	16	32	e1 MD5
md4	16	32	e3 MD4
md2	16	32	e4 MD2
wrl	64	128	e5 WRL (Whirlpool)
hav128	16	32	e6 HAV128
sha0	20	40	e7 SHA0
sha1	20	40	e8 SHA1
sha224	28	56	e9 SHA224
sha256	32	64	e10 SHA256
sha384	48	96	e11 SHA384
sha512	64	128	e12 SHA512
gost	32	64	e13 GOST
gost_crypto	32	64	e14 GOST-CRYPTO
hav256	32	64	e15 HAV256
rmd128	16	32	e16 RMD128
rmd160	20	40	e17 RMD160
tiger	24	48	e18 TIGER
tth	24	48	e19 TTH
ed2k	16	32	e20 ED2K
aich	20	40	e21 AICH
has160	20	40	e22 HAS160
edon256	32	64	e23 EDON256
edon512	64	128	e24 EDON512
sne128	16	32	e25 SNE128
sne256	32	64	e26 SNE256
md6	16	32	e27 MD6
md6128	16	32	e28 MD6128
md6256	32	64	e29 MD6256
md6512	64	128	e30 MD6512

TABLE 7. Primitive Hash Functions (e1–e30)

Types beyond e31 include additional Haval variants (hav128_4, hav128_5, hav160_3, etc.), the full BLAKE, BMW, CubeHash, ECHO, Fugue, Groestl, Hamsi, JH, Keccak, SHA-3, Luffa, Shabal, SHAvite, SIMD, and Skein families, as well as HMAC variants and keyed constructions. All are available by their lowercased, underscore-normalized names.

Uppercase hash types (such as MD5UC, e2) are not separate functions in hx. The **upper()** transform function is used instead: `upper(md5(pass))` is the hx equivalent of mdxfind type e2 (MD5UC). This is more explicit: the uppercase encoding is a formatting operation, not a property of the hash function.

3.2 HMAC Functions

HMAC functions take two arguments: the key and the message.

```
hmac_sha1(pass, salt)      # HMAC-SHA1, key=pass, msg=salt
hmac_sha256(pass, salt)    # HMAC-SHA256
```

The convention follows the standard HMAC definition: the first argument is the key, the second is the message.

3.3 String Transform Functions

Transform functions operate on strings without performing cryptographic hashing.

Function	Arguments	Description
hex(x)	1	Encode bytes as lowercase hex
fromhex(x)	1	Decode hex string to bytes
upper(x)	1	Convert to uppercase
lower(x)	1	Convert to lowercase
rev(x)	1	Reverse byte sequence
rotate(x, N)	2	Rotate string by N positions
cap(x)	1	Capitalize first lowercase letter
cap(x, N)	2	Capitalize letter at position N
rot13(x)	1	ROT13 alphabetic rotation
pad(x, N)	2	Null-pad to N bytes
xor(a, b)	2	Byte-wise exclusive OR
and(a, b)	2	Byte-wise AND
or(a, b)	2	Byte-wise OR
wperm(x, w0, w1, ...)	2+	Permute 4-byte words
bswap32(x)	1	Reverse bytes within each 4-byte group
length(x)	1	Byte length (returns integer)

TABLE 8. String Transform Functions

The **hex()** function is particularly important for explicit encoding control. The expression `hex(md5_bin(pass))` is equivalent to `md5(pass)`; both produce the same 32-character hex string. The explicit form is preferred when the intent is to emphasize that a binary intermediate is being encoded for output.

The **rotate()** function rotates a string by N character positions. Positive N rotates right (last N characters move to the front); negative N rotates left (first |N| characters move to the end):

```
rotate("abcdef", 2)     # "efabcd"
rotate("abcdef", -2)    # "cdefab"
rotate(md5(pass), 5)    # MD5 hex rotated right by 5
```

This function is essential for expressing rotated hash matches, where the hex representation of a hash has been cyclically shifted before storage. mdxfind detects these as `rotNN_TYPExMM` in its output. The hx equivalent is `rotate(type(pass), N)`.

The **xor()** function performs byte-wise exclusive OR on two byte sequences. If the arguments are of unequal length, the shorter argument is cycled (repeated) to match the length of the longer. This is primarily useful for expressing the internal structure of HMAC (ipad/opad XOR) and for obfuscation schemes that XOR data with a key:

```
xor(sha1_bin(pass), salt)
```

The **and()** and **or()** functions perform byte-wise AND and OR respectively, with the same cycling semantics as **xor()**. These are useful for masking digest bytes:

```
hex(and(sha1_bin(pass), "\x00\x00\x00\x00\xff..."))
```

The **cap()** function capitalizes a letter within a string. With one argument, it finds and capitalizes the first lowercase ASCII letter. With two arguments, it capitalizes the letter at the specified position:

```
cap(md5(pass))          # first lowercase letter
cap(md5(pass), 0)       # character at position 0
```

The **rot13()** function applies ROT13 alphabetic rotation to each ASCII letter, leaving non-letters unchanged.

The **pad()** function null-pads a value to a fixed length. If the input is shorter than the specified length, zero bytes are appended; if longer, it is truncated. This is required for schemes that use fixed-width fields:

```
md5(pad(pass, 100))     # RADMIN2
```

The **bswap32()** function reverses the byte order within each 4-byte group of its argument, performing a 32-bit byte-swap on every complete group. If the input length is not a multiple of 4, the trailing 1–3 bytes are copied unchanged:

- `bswap32(fromhex("aabbccdd"))` → 0xddccbbaa (4 bytes: one full group reversed)
- `bswap32(fromhex("aabbccddee"))` → 0xddccbbaa 0xee (5 bytes: one group reversed, one byte unchanged)
- `bswap32(fromhex("aabb"))` → 0xaabb (2 bytes: no full group, copied as-is)
- `bswap32("")` → "" (empty)

The function is intended for converting between big-endian and little-endian representations of 32-bit word arrays, as commonly encountered in hash digest output. Applied to text, the result is meaningless.

The **wperm()** function permutes 4-byte words within a byte sequence. The first argument is the data; subsequent arguments are zero-based word indices specifying the new order:

```
hex(wperm(md5_bin(pass), 3, 2, 0, 1))  # words d,c,a,b
```

The **length()** function returns the byte length of its argument as an integer. The result has both an integer value (for use in loop bounds and comparisons) and a string representation (for concatenation):

```
for i = 0 to sub(length(pass), 1) {
    h = h . md5(cut(pass, i, 1))
}
```

3.4 Extraction Functions

Extraction functions select portions of a string.

3.4.1 cut()

The **cut()** function extracts a substring, with semantics similar to the C library function **substr()**. It accepts one, two, or three arguments:

Form	Meaning
cut(x)	identity (returns x unchanged)
cut(x, start)	from start to end of string
cut(x, start, length)	length characters from start

TABLE 9. cut() Function Forms

The *start* position is zero-based. A negative *start* counts from the end of the string:

```
cut("abcdef", 2, 3)     # "cde"
cut("abcdef", -3)        # "def"
cut("abcdef", -3, 2)     # "de"
```

3.4.2 trunc()

trunc() is a convenience alias for `cut(x, 0, n)`:

```
trunc(sha512(pass), 32)  # first 32 hex characters
```

3.5 Encoding Functions

3.5.1 base64()

Standard Base64 encoding as defined in RFC 4648, with **=** padding characters. Primarily used with **_bin** outputs to produce base64-encoded hash digests:

```
base64(md5_bin(pass))
```

3.5.2 phpass_encode()

The itoa64 encoding used by phpass, PHPBB3, and WordPress. This encoding uses the 64-character alphabet `./0-9A-Za-z` with little-endian 6-bit packing. A 16-byte MD5 digest produces 22 encoded characters.

```
phpass_encode(md5_bin(pass))
```

This function is essential for producing the correct output format for PHPBB3 and WordPress hashes, where the final hash value is not hex-encoded but uses this custom alphabet.

3.5.3 sha512crypt_encode()

The custom base64 encoding used by SHA-512 crypt (**6**). Uses the same alphabet as phpass (**./0-9A-Za-z**) but with a different byte permutation and big-endian bit packing specific to the crypt(3) specification.

3.5.4 bcrypt_encode()

The custom base64 encoding used by bcrypt, with alphabet `./ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789`.

3.5.5 frombase64()

Decodes a standard base64 string (RFC 4648) to raw bytes. The inverse of **base64()**. **=**-padding is handled correctly. This is required for hash types where the salt is stored in base64 encoding:

```
sha1(utf16le(pass) . frombase64(salt))
```

3.5.6 fromhex()

Decodes a hexadecimal string to raw bytes. The inverse of **hex()**. Each pair of hex characters produces one byte.

3.5.7 progress_encode()

The custom base64-style encoding used by Progress OpenEdge passwords. Operates on a 16-byte MD5 digest and produces a 22-character output drawn from the alphabet `./0-9A-Za-z`. The byte-permutation and bit-packing differ from both **phpass_encode()** and **sha512crypt_encode()**; the function exists to faithfully reproduce the on-disk Progress OpenEdge format.

3.5.8 juniper_encode()

The Type-9 / "9" password encoding used by Juniper Junos OS. Operates on a 16-byte MD5 digest and produces a 13-character ASCII output using a fixed permuted alphabet defined by Juniper. The result is the **9** secret stored in **configuration** files.

3.5.9 cisco_pix_encode()

The 16-character Base64 encoding used by Cisco PIX and ASA enable/local password hashes. Operates on a 16-byte MD5 digest and produces a 16-character output using the alphabet `./0-9A-Za-z` with PIX/ASA-specific bit packing. The PIX password algorithm itself is *md5*(**pad(pass . salt, 16)**) followed by this encoding; **cisco_pix_encode()** covers only the encoding step.

3.5.10 lotus64_encode()

The Lotus / Domino base64-style encoding used in the Domino-6 password format. Operates on a 10-byte buffer (8 hash bytes plus 2 length/header bytes) and produces a 14-character output, which is the body of an **(Gxxxxxxxxxxxxx)** Domino-6 hash.

3.5.11 besder_encode()

The 8-character lossy fold used by Besder camera authentication (mdxfind type e963). Accepts a 16-byte input (typically an MD5 digest); for each of the four 4-byte groups, two characters are emitted from the alphabet `0-9A-Za-z` using the rule ((b**0+b**1) **&**0xff × **62**). The expression `besder_encode(md5_bin(pass))` reproduces the BESDER-AUTH (e960) hash format.

3.5.12 dahua_encode()

The Dahua-camera variant of the same fold used by **besder_encode()**, without the **&**0xff byte truncation ((b**0+b**1) × **62**). This corresponds to mdxfind type e959 (DAHUA-AUTH).

3.6 Cipher Functions

Cipher functions wrap the OpenSSL block-cipher primitives used by hash schemes that include an explicit encrypt/decrypt step. All produce raw ciphertext or plaintext bytes; the default rendering is lowercase hex, but the **_bin**, **_hex**, and **_b64** suffixes apply per the encoding contract.

Function	Arguments	Description
des(key, pt)	2	Single-DES ECB encrypt of one 8-byte block
des3(key, pt)	2	Triple-DES (EDE) ECB encrypt of one 8-byte block
aes_ecb_encrypt(key, pt)	2	AES-ECB encrypt; key 16/24/32 bytes
aes_cbc_encrypt(key, iv, pt)	3	AES-CBC encrypt, no padding
aes_cbc_decrypt(key, iv, ct)	3	AES-CBC decrypt, no padding strip
aes_unwrap(kek, wrapped)	2	RFC 3394 AES Key Unwrap

TABLE 10. Cipher Functions

des() and **des3()** operate on a single 8-byte block. The key is zero-padded (or truncated) to 8 or 24 bytes; the plaintext is padded to 8 bytes if shorter. These cover legacy schemes such as Oracle 7 (**oracle7()**, which calls **des()**), AS/400 password hashes (**as400_des()**), and Domino-5 challenge mixing.

aes_ecb_encrypt() encrypts a positive multiple of 16 bytes under AES-128, AES-192, or AES-256, selected by the key length. This is the building block for the AES-NOKDF family (e963/e964/e965), where the password is zero-padded to the key size and the salt is the single-block plaintext:

```
aes_ecb_encrypt(pad(pass, 32), fromhex(salt))
```

aes_cbc_encrypt() and **aes_cbc_decrypt()** operate on positive multiples of 16 bytes with a 16-byte IV. Neither function performs PKCS#7 padding or strip-on-output: the caller is responsible for preparing or verifying the padding bytes explicitly. This deliberate choice exposes the raw block behavior so that schemes which use a non-standard padding marker (e.g. Veeam VBK's 12-byte **0x0c** trailer) can verify the marker themselves with a **cut()** plus equality test:

```
dk = pbkdf2_sha1_bin(utf16le(pass), salt, iter, 48)
pt = aes_cbc_decrypt_bin(cut(dk,0,32), cut(dk,32,16), ct)
if cut(pt,4,12) == fromhex("0c0c0c0c0c0c0c0c0c0c0c0c") { emit("ok") }
```

aes_unwrap() implements the RFC 3394 AES Key Wrap reverse operation. The key-encryption key (*kek*) must be 16 or 32 bytes (AES-128 or AES-256), and the wrapped input must be *8* + *8*n bytes for some *n >= 1*. On a successful unwrap (the leading IV matches **0xa6a6a6a6a6a6a6a6**), the function returns the *8*n-*byte* plaintext; on IV mismatch, it returns the empty string. The empty/non-empty distinction is the standard way to verify Apple FileVault/APFS passwords:

```
length(aes_unwrap(pbkdf2_sha256_bin(pass, salt, iter, 32), wkey)) > 0
```

3.7 Proprietary Algorithm Primitives

A small number of hash schemes use proprietary, hand-built password mixers that are not derived from standard hash or KDF building blocks. For each such scheme the algorithm has been ported verbatim from **mdxfind.c** and exposed as an opaque hx primitive.

Function	Arguments	Description
domino5(pass)	1	Lotus Domino 5 challenge mixer
domino6(pass)	1	Lotus Domino 6 password hash (calls lotus64_encode)
oracle7(user, pass)	2	Oracle 7 / 8 / 9 / 10 user-and-password DES chain
sap_bcode(user, pass)	2	SAP CODVN B (BCODE) password digest
as400_des(user, pass)	2	IBM AS/400 OS/400 DES-based password hash
axcrypt(pass)	1	AxCrypt key-file mixing function
racf_encrypt(user, pass)	2	IBM RACF DES-based password hash
racf_kdfaes(pass, user, hdr, salt)	4	IBM RACF KDFAES (HMAC-SHA256 memory mixer)

TABLE 11. Proprietary Algorithm Primitives

Each of these primitives is documented in a trailing footnote of **hx**(8) (Appendix A, *Notes [1]–[32]*), which gives the C reference implementation, the exact buffer layout, and any required input preprocessing.

The default rendering for the digest-producing primitives (**oracle7()**, **sap_bcode()**, **as400_des()**, **axcrypt()**, **racf_encrypt()**, **racf_kdfaes()**, **domino5()**) is lowercase hex; the **_bin**, **_hex**, **_b64** suffixes apply. The encoder-style primitives (**domino6()**, **lotus64_encode()**, **juniper_encode()**, **cisco_pix_encode()**, **progress_encode()**, **besder_encode()**, **dahua_encode()**) return ASCII text and accept no role suffix.

3.8 Character Encoding Functions

Character encoding functions convert the byte representation of a string.

Function	Arguments	Description
utf16le(x)	1	Convert UTF-8 to UTF-16 little-endian
utf16be(x)	1	Convert UTF-8 to UTF-16 big-endian
utf7(x)	1	Convert to UTF-7 encoding
zext16(x)	1	Zero-extend each byte to 16 bits little-endian
from_cp1252(x)	1	Convert UTF-8 to Windows-1252 (Latin-1 superset)
from_cp1251(x)	1	Convert UTF-8 to Windows-1251 (Cyrillic)
ebcdic(x)	1	Convert ASCII to IBM EBCDIC (CP037)

TABLE 12. Character Encoding Functions

utf16le() is essential for NTLM and related Microsoft hash types, which hash the UTF-16LE encoding of the password rather than its UTF-8 or ASCII representation:

```
md4(utf16le(pass))      # NTLM
```

The conversion handles full Unicode: multi-byte UTF-8 sequences are properly decoded and re-encoded as UTF-16, including surrogate pairs for characters above U+FFFF.

zext16() performs the same byte-level expansion as **utf16le()** but without UTF-8 decoding: every input byte is paired with a trailing 0x00, regardless of whether the input is valid UTF-8. This is used by hash schemes that name themselves after UTF-16LE but in practice operate on the raw byte stream.

from_cp1252() and **from_cp1251()** convert a UTF-8 string to the corresponding single-byte Windows code page. Characters that are not representable in the target code page are passed through as **'?'**. These conversions are required by hashes which were historically computed over the legacy single-byte encoding rather than UTF-8.

ebcdic() converts an ASCII string to IBM EBCDIC code page 037 (the standard EBCDIC mapping used in z/OS RACF and related mainframe contexts). The companion EBCDIC variant CP1047 is used internally by **racf_kdfaes()** and is not exposed as a separate primitive.

3.9 Arithmetic Functions

Arithmetic functions operate on integer values and return integer results (with a string representation for use in string context).

Function	Arguments	Description
add(a, b)	2	Integer addition
sub(a, b)	2	Integer subtraction
mul(a, b)	2	Integer multiplication
mod(a, b)	2	Integer modulo (b = 0 returns 0)
length(x)	1	Byte length of x

TABLE 13. Arithmetic Functions

These functions enable computed loop bounds and conditional iteration patterns:

```
for i = 0 to sub(length(pass), 1) {
    h = h . md5(cut(pass, i, 1))
}
```

3.10 Output Function

3.10.1 emit()

The **emit()** function writes its argument to standard output, followed by a newline, and returns the value unchanged (passthrough). It is used for hash types that produce multiple output values from a single password:

```
h = md5(pass)
emit(md5(pass . h))
emit(md5(pass . ":" . h))
```

When a program contains any **emit()** calls, the automatic final-value output is suppressed. Without **emit()**, the last expression value is output as before. This rule is unambiguous: the presence of **emit()** in the compiled program determines the output mode.

If **emit()** receives binary data (any byte outside the printable ASCII range 0x20–0x7e), it automatically hex-encodes the output. This prevents raw binary from corrupting the output stream when **_bin** results are emitted.

3.11 Key Derivation Functions

KDF functions are opaque built-ins that call native implementations. They accept integer parameters for cost/iteration factors:

Function	Signature
bcrypt	bcrypt(pass, salt, cost)
scrypt	scrypt(pass, salt, N, r, p, dklen)
argon2id	argon2id(pass, salt, time, mem_kb, parallel, dklen)
argon2i	argon2i(pass, salt, time, mem_kb, parallel, dklen)
argon2d	argon2d(pass, salt, time, mem_kb, parallel, dklen)
yescrypt	yescrypt(pass, setting)
pbkdf2_sha1	pbkdf2_sha1(pass, salt, iterations, dklen)
pbkdf2_sha256	pbkdf2_sha256(pass, salt, iterations, dklen)
pbkdf2_sha512	pbkdf2_sha512(pass, salt, iterations, dklen)
pbkdf2_md5	pbkdf2_md5(pass, salt, iterations, dklen)

TABLE 14. Key Derivation Functions

PBKDF2 functions are available in both standalone and hashpipe builds (via OpenSSL). The remaining KDFs require hashpipe integration.

bcrypt() returns the full crypt string `($2b$CC$salt+hash)` by default, or the hash portion only with **_bin**. The cost parameter ranges from 4 to 31 (default 12).

scrypt() parameters: N = CPU/memory cost (power of 2, default 16384), r = block size (default 8), p = parallelism (default 1), dklen = output length (default 32).

argon2id() parameters: time = iterations (default 3), mem_kb = memory in KB (default 65536), parallel = lanes (default 4), dklen = output length (default 32).

4. Iteration and Control Flow

Many password hashing schemes apply a hash function repeatedly to increase computational cost, or branch on a comparison of a derived value against a known target. hx provides three mechanisms for expressing iteration and conditional execution: the ^ operator, the **for** loop, and the **if** statement.

4.1 The ^ Operator

The caret operator provides concise notation for simple re-hashing, where the output of one round becomes the sole input to the next:

```
md5^1000(pass)
```

This expression applies **md5()** to the password, then applies **md5()** to the result, 999 more times, for a total of 1000 applications. The encoding rule applies at each step: each round receives the hex output of the previous round.

With the **_bin** suffix, the raw bytes are chained instead:

```
md5_bin^1000(pass)
```

The ^ operator is syntactic sugar. The expression `md5^3(pass)` is exactly equivalent to:

```
md5(md5(md5(pass)))
```

The ^ operator is limited to the case where the function is applied to its own output with no additional inputs mixed in per round. For schemes that mix salt, password, or other data at each iteration, the **for** loop is required.

4.2 The for Loop

The **for** statement provides explicit iteration with a named loop variable, a start value, and an end value (inclusive):

```
for var = start to end {
    body
}
```

The loop variable is an integer that is incremented by one after each iteration of the body. The body is executed while the variable is less than or equal to the end value.

The **for** loop is required for hash schemes where additional data is mixed into each iteration. The canonical example is PHPBB3 (phpass), which concatenates the previous hash result with the original password at each round:

```
h = md5_bin(salt . pass)
for i = 1 to 2047 {
    h = md5_bin(h . pass)
}
hex(h)
```

This cannot be expressed with the ^ operator because the loop body is not simply "apply the function to its own output" — it concatenates **pass** at each step.

The initial assignment `h = md5_bin(salt . pass)` is the first of 2048 total MD5 operations. The loop runs 2047 more iterations, each taking the binary output of the previous round, concatenating the password, and computing a new MD5. The final `hex(h)` converts the last binary digest to hex for output.

4.3 The if Statement

The **if** statement executes a block conditionally based on the result of a relational expression.

```
if expr relop expr {
    then-body
} else {
    else-body
}
```

The **else** clause is optional. The condition is a single relational test, not a chain of Boolean expressions; there are no **and**, **or**, or **not** operators on Booleans because hx has no Boolean type. The supported relops are **==**, **!=**, **<**, **>**, **<=**, **>=**.

Comparison semantics are strict-typed: integer—versus—integer compares signed magnitudes, string—versus—string compares with **memcmp**, and a mixed integer/string comparison raises a runtime error rather than coercing. A comparison such as `length(pass) > 8` works because both sides are integers; a comparison such as `cut(d, 4, 12) == fromhex("0c0c0c0c0c...")` works because both sides are strings.

The intended use of **if** is to drive **emit()** from a positive verification, so that the program prints something on a match and nothing otherwise:

```
dk = pbkdf2_sha1_bin(utf16le(pass), fromhex(salt), iter, 48)
pt = aes_cbc_decrypt_bin(cut(dk,0,32), cut(dk,32,16), fromhex(ct))
if cut(pt,4,12) == fromhex("0c0c0c0c0c0c0c0c0c0c0c0c") { emit("ok") }
```

This pattern fits hash schemes whose verification step is a structural property of a decrypted value (Veeam VBK, Apple FileVault/APFS, certain WPA flavors) rather than a hash equality. The grammar requires the closing brace and the **else** keyword to appear on the same line; see **Statement Separators** in the Lexical Structure chapter.

5. Compound Hash Expressions

The power of hx lies in its ability to express arbitrary compositions of hash functions. A compound hash applies multiple functions in sequence, with the output of one becoming the input to the next, optionally concatenated with salt or other data.

5.1 Simple Composition

Functions nest naturally:

```
sha1(md5(pass))
```

This computes MD5(password) as a 32-character hex string, then computes SHA-1 of those 32 characters. The result is a 40-character hex string. It corresponds to mdxfind type **SHA1MD5PASS**.

5.2 Composition with Salt

Salt is introduced via concatenation:

```
sha1(md5(pass) . salt)
```

This computes MD5(password) as hex, concatenates the salt, then computes SHA-1 of the combined string. It corresponds to mdxfind type **SHA1MD5SALT** (e587).

The placement of salt relative to other components is significant. The following are all different computations:

```
md5(pass . salt)        # password then salt
md5(salt . pass)        # salt then password
md5(salt . pass . salt)  # salt, password, salt
```

These correspond to mdxfind types **MD5PASSSALT**, **MD5SALTPASS**, and **MD5SALTPASSSALT** respectively.

5.3 Binary Intermediates

When two hash functions are chained with a binary intermediate (the raw digest bytes rather than their hex encoding), the inner function uses the **_bin** suffix:

```
sha1(md5_bin(pass))      # feed 16 raw bytes to SHA-1
sha1(md5(pass))          # feed 32 hex chars to SHA-1
```

The first expression corresponds to mdxfind type **SHA1MD5RAW** (e641); the second to **SHA1MD5PASS** (e161). These produce different results and must not be confused.

Binary intermediates are common in real-world hash schemes. MySQL's native password format uses:

```
sha1(sha1_bin(pass))
```

Many LDAP SSHA schemes concatenate the raw hash with the salt before base64-encoding the result. The **_bin** suffix makes these compositions explicit and unambiguous.

5.4 Constructing Output Formats

Some hash schemes have distinctive output formats that include type identifiers, salt values, and the encoded hash in a specific arrangement. hx expressions can construct these formats using string literal concatenation.

PHPBB3 output format:

```
"$H$9" . salt . phpass_encode(h)
```

A hypothetical format with a prefix and base64:

```
"{SSHA}" . base64(sha1_bin(pass . salt) . salt)
```

These expressions produce the complete output string, including all formatting, ready for comparison against a hash file.

6. Hashpipe Integration

hx is fully integrated into hashpipe. It has access to the full set of hash functions that hashpipe supports.

6.1 Function Registration

At startup, when the **−X** or **−F** flag is present, hashpipe initializes its type registry (the same initialization used for normal hash verification) and then walks the registry to register each type as an hx function.

For each hashpipe type that has a *compute* function and a positive hash length, a bridge entry is created in the hx function registry. The bridge entry stores a pointer to the hashpipe compute function and the digest size in bytes. When the hx VM executes a **CALL** instruction for a bridged function, it invokes the generic bridge wrapper, which:

1. Extracts the input data from the hx value on the stack.
2. Calls the hashpipe compute function with the standard signature `(pass, passlen, salt, saltlen, dest)`.
3. Hex-encodes the digest (or returns raw binary if the **_bin** flag is set).
4. Stores the result in the arena and pushes it onto the hx stack.

This bridge mechanism means that adding a new hash function to hashpipe automatically makes it available in hx expressions, with no changes to the hx source code.

6.2 Command-Line Interface

The following flags control hx mode in hashpipe:

Flag	Meaning
−X *expr*	Evaluate hx expression
−F *file*	Run hx script file
−p *pass*	Single password (otherwise stdin)
−s *salt*	Salt value (shared with statfile flag)
−S *salt2*	Second salt value
−P *pepper*	Pepper value
−u *user*	User/userid value

TABLE 15. hashpipe hx Mode Flags

The **−X** and **−F** flags are mutually exclusive. When either is present, hashpipe enters hx mode and does not perform its normal hash verification pipeline.

The **−s** flag serves double duty: in normal hashpipe mode it specifies the statistics file; in hx mode it provides the salt value. This dual use is intentional to minimize the number of new flags required.

6.3 User-Defined Hash Types

An hx expression may be registered as a user-defined hash type for use in mdxfind and hashpipe, with no C code and no recompile. The expression is placed in a **userdef.txt** stanza file and selected in mdxfind with **−m u<id>**, producing matches labelled **USER_<name>xNN**. hashpipe loads the same file and identifies **USER_*** matches in pipe mode — a candidate line such as

```
echo <digest>:<password> | hashpipe
```

reports the matching user type alongside any matching built-in type. The full stanza format, the **−m u<id>** selector, the load-time advisories, and the v1 CPU and unsalted scope are documented in the mdxfind manual under "User-Defined Hash Types".

7. Worked Examples

The following examples demonstrate hx expressions for common hash schemes, with explanations of each step. All examples can be executed with **hashpipe −X** and produce verifiably correct output.

7.1 MD5 (e1)

```
$ hashpipe -X 'md5(pass)' -p password
5f4dcc3b5aa765d61d8327deb882cf99
```

The simplest case. **md5()** computes the MD5 digest of the password and returns it as 32 lowercase hex characters.

7.2 SHA-1 of MD5 with Salt (e587)

```
$ hashpipe -X 'sha1(md5(pass) . salt)' -p password -s test
d9365e268de8b33a989c390ebaf3f12246b281da
```

Evaluation proceeds inside-out: **md5(pass)** produces the 32-character hex string `5f4dcc3b5aa765d61d8327deb882cf99`, which is concatenated with **salt** ("test") to form a 36-character string, which is then hashed with SHA-1.

7.3 Triple MD5 (MD5x03)

```
$ hashpipe -X 'md5^3(pass)' -p password
5a22e6c339c96c9c0513a46e44c39683
```

Equivalent to `md5(md5(md5(pass)))`. Each round hashes the 32-character hex output of the previous round.

7.4 Truncated SHA-512 (Partial Match)

```
$ hashpipe -X 'cut(sha512(pass), 0, 32)' -p password
b109f3bbbc244eb82441917ed06d618b
```

The full SHA-512 hex output is 128 characters. **cut()** extracts the first 32, producing the same string that mdxfind would match against a 32-character hex hash file when testing SHA-512 candidates.

7.5 PHPBB3 / phpass

```
$ hashpipe -X 'h = md5_bin(salt . pass); \
    for i = 1 to 2047 { h = md5_bin(h . pass) }; \
    "$H$9" . salt . phpass_encode(h)' \
    -p password -s saltsalt
$H$9saltsaltWxax0v4yfV8oT17RChGlx1
```

This is the complete PHPBB3 computation in a single expression. The **H9** prefix indicates the hash variant and iteration count ($2^{SU}11 = 2048$ iterations). The salt is concatenated literally. The final hash value is encoded with **phpass_encode()** using the itoa64 alphabet, not hex.

Key observations:

- **md5_bin()** is used throughout because phpass chains raw binary digests, not hex strings.
- The initial hash includes the salt: `md5_bin(salt . pass)`.
- Each subsequent round concatenates the original password to the previous binary result: `md5_bin(h . pass)`. This is the pattern that requires a **for** loop rather than the ^ operator.
- The final `phpass_encode(h)` produces 22 characters of itoa64-encoded output from the 16-byte MD5 digest.

7.6 MySQL Native Password

```
$ hashpipe -X 'upper(sha1(sha1_bin(pass)))' -p password
```

MySQL's native password format is SHA-1 of the raw SHA-1 digest, with uppercase hex output. The **_bin** suffix on the inner SHA-1 is critical: the outer SHA-1 receives 20 raw bytes, not the 40-character hex string.

7.7 HMAC-SHA1

```
$ hashpipe -X 'hmac_sha1(pass, salt)' -p key -s message
2088df74d5f2146b48146caf4965377e9d0be3a4
```

The first argument is the HMAC key, the second is the message. The result is the 40-character hex encoding of the 20-byte HMAC-SHA1 output.

7.8 Base64-Encoded MD5

```
$ hashpipe -X 'base64(md5_bin(pass))' -p password
X03MO1qnZdYdgyfeuILPmQ==
```

This produces the same output as `echo -n password | openssl md5 -binary | openssl base64`. The **_bin** suffix is essential: base64 encodes the raw 16-byte digest, not the 32-character hex string.

7.9 Compound with String Literal

```
$ hashpipe -X 'md5("prefix" . pass . "$")' -p password
```

String literals can appear anywhere in a concatenation. This computes MD5 of the string "prefixpassword$".

7.10 Last N Characters of Password

```
$ hashpipe -X 'md5(cut(pass, -6))' -p helloworld
```

cut() with a negative offset extracts from the end. This computes `md5("oworld")`.

8. Debugging

hx provides two diagnostic flags that expose the internal representation of a compiled program.

8.1 AST Dump (−d)

The **−d** flag (standalone hx only) prints the abstract syntax tree to standard error:

```
$ hx -d 'sha1(md5(pass) . salt)'
BLOCK (1 stmts)
  CALL sha1 (nargs=1)
    CONCAT
      CALL md5 (nargs=1)
        VAR pass
      VAR salt
```

The tree shows the nesting of function calls and the concatenation structure, matching the evaluation order.

8.2 Bytecode Dump (−b)

The **−b** flag prints the compiled bytecode:

```
$ hx -b 'sha1(md5(pass) . salt)'
--- hx bytecode: 6 instructions, 4 vars, 0 strings ---
  var[0] = pass
  var[1] = salt
  var[2] = salt2
  var[3] = pepper
    0  PUSH_VAR   0 (pass)
    1  CALL       md5 nargs=1
    2  PUSH_VAR   1 (salt)
    3  CONCAT
    4  CALL       sha1 nargs=1
    5  HALT
```

The bytecode shows the flat instruction sequence that the VM executes. Arguments are pushed left to right, function calls consume their arguments and push the result. The variable slot assignments, string constant table, and instruction addresses are all visible.

9. The Algorithm Catalog

Every algorithm that mdxfind recognises is described by exactly one entry in the hx algorithm catalog. The catalog is presented as Appendix A (**hx**(8)) and is published alongside this specification as *https://www.mdxfind.com/hx.pdf* . Each entry consists of an **eN** type number, a short symbolic name, and the *hx*expression that defines the computation.

The hx expression *is* the definition of the algorithm. The symbolic name is a mnemonic; the type number is a stable identifier used at the command line and in **mdxfind** output; the hx expression is the authoritative, machine-readable specification of what the algorithm computes. When a researcher asks "what does **e587** mean?" the answer is the hx expression `sha1(md5(pass) . salt)` — not a paragraph of English prose.

The expression states what the algorithm *computes*. Which of the values in that computation a given tool can *present* for matching is a separate question, settled by how that tool compares. **mdxfind** converts a loaded hash to binary, so two renderings of the same digest that differ only in hex case are the same bytes to it and cannot be told apart; a type whose only distinguishing feature is that case therefore has no comparable value at **x01**, and the first depth mdxfind can present is one hash further on. **hashpipe** compares the stored text and can distinguish the two, so it reports the same type at **x01**. Neither is wrong, and the catalog entry is unaffected: each tool reports the first value its own comparison can distinguish.

9.1 Catalog Stewardship

A single shared catalog binds the mdxfind family of tools together. **mdxfind** uses the catalog to assign type numbers and report matches. **hashpipe** uses the catalog to verify hashes and to evaluate ad-hoc expressions via the **−X** flag. **mdsplit** uses the catalog to associate matched output with the correct symbolic name. A correction applied to a catalog entry is therefore a correction that propagates uniformly across every tool that consumes the catalog.

The catalog is maintained as a plain-text troff document (**hx**(8)) in the same source tree as this specification. When a discrepancy is identified between a catalog entry and the implementation in **mdxfind**, the entry is corrected in **hx**(8) and republished. The expression text in the catalog is the canonical reference; the C implementation in **mdxfind** exists to compute the same result at higher throughput.

9.2 From Catalog to Bytecode

The catalog is compiled to a bytecode representation at build time by the **hx8_to_c** tool, which parses **hx**(8), emits a C source file (**codegen/hx_specs_data.c**) that contains the bytecode and metadata for every entry, and links it into both **mdxfind** and **hashpipe**.

The bytecode is the same instruction set described in Section 2.6: a flat array of **PUSH_VAR**, **PUSH_STR**, **CALL**, **CONCAT**, and the small number of control-flow opcodes that the language requires. A single VM implementation interprets the bytecode for every entry in the catalog. There is no per-algorithm C code in the hx runtime; adding a new algorithm to the catalog adds a new bytecode program, not a new function.

The bytecode representation supports tooling beyond straight execution. A bytecode walker can identify duplicates and near-duplicates across the catalog, compute structural fingerprints for algorithm families, and serve as the input to alternative backends.

10. Cross-Architecture Execution

The hx VM is platform-independent. The bytecode produced from a catalog entry runs identically on any CPU architecture that **mdxfind** or **hashpipe** supports: x86-64, ARM64, POWER, SPARC. Bit-for-bit reproducibility is a property of the language, not of any particular host.

For algorithms eligible for GPU acceleration, the same bytecode also drives the GPU execution path. A backend-specific code emitter consumes the bytecode and produces kernel source for the target accelerator: OpenCL source for NVIDIA, AMD, and Intel devices, and Metal source for Apple Silicon GPUs. The emitted kernel computes the same expression that the CPU VM interprets; the two outputs are byte-exact for the same inputs.

The user selects the algorithm by its **eN** number and the device by the **–G** flag; the backend choice is transparent. The expression `sha1(md5(pass) . salt)` — catalog entry **e587** — produces the same 40-character hex string whether the work runs on an x86-64 thread on a desktop, an ARM64 thread on Apple Silicon, an OpenCL workgroup on an NVIDIA GTX 1080, or a Metal threadgroup on an Apple M2 Max.

This architecture has a specific consequence for catalog authors: an algorithm needs only one definition. The hx expression in **hx**(8) is consumed by the CPU VM, by the OpenCL emitter, and by the Metal emitter. A correction to the expression takes effect on every backend on the next build.

11. Tooling

hx is supported by a small collection of host-side tools.

11.1 hx8_to_c

hx8_to_c is the catalog compiler. It parses **hx**(8), validates every expression against the grammar of Section 2.2 and the function registry of Section 3, and emits a C source file containing the bytecode, the variable slot map, the string constant table, and the metadata for each entry. The output is the single source of truth that **mdxfind** and **hashpipe** link against; both tools observe the same catalog and the same compiled bytecode.

A typical invocation:

```
hx8_to_c hx.8 > codegen/hx_specs_data.c
```

Diagnostics are written to standard error. If any entry fails to compile, the tool reports the **eN** number, the source line in **hx**(8), and a description of the error; the offending entry must be corrected before the build can proceed.

11.2 hashpipe –X

The **–X** flag to **hashpipe** accepts an hx expression on the command line, compiles it in process, and evaluates it for the supplied inputs. This is the recommended way to construct, debug, and verify hx expressions:

```
hashpipe -X 'sha1(md5(pass) . salt)' -p password -s test
```

–F takes the same role for multi-statement programs read from a file. Both flags accept the standard hashpipe input modes: a single password via **–p**, a stream of passwords on standard input, or a wordlist file.

11.3 hashpipe catalog smoke tests

hashpipe treats the catalog as a regression target. For every entry that has known test vectors, hashpipe can verify that the compiled bytecode produces the expected output for the documented inputs. This is the mechanism by which catalog drift is detected and corrected.

11.4 Standalone hx interpreter

For language development and pedagogy, a standalone **hx** interpreter is provided. It reads a program from a file or the command line, compiles it, and runs it against inputs supplied on the command line. The **–d** and **–b** flags described in Section 7 dump the AST and the compiled bytecode respectively, exposing the internal representation for inspection.

12. Summary

hx is a hash expression language designed for precision, clarity, and longevity. Its notation is unambiguous: the expression `sha1(md5(pass) . salt)` means exactly one thing, and evaluating it produces exactly one result, which matches the output of `mdxfind -m e587` on every supported CPU and GPU.

The language's deliberately limited scope — hash functions, concatenation, encoding, iteration, and simple control flow — is a strength, not a limitation. By constraining the domain, hx ensures that every expression is a complete, verifiable specification of a hash computation, suitable for documentation, for test vector generation, and for independent verification of results produced by high-performance tools.

The algorithm catalog (**hx**(8), published as *https://www.mdxfind.com/hx.pdf*) is the canonical reference for every type that **mdxfind** recognises. A catalog entry is its hx expression; the **eN** type number and the symbolic name are convenient identifiers, but the expression is the definition. The same expression drives the CPU VM in **hashpipe**, the high-throughput implementations in **mdxfind**, and the GPU kernels emitted for OpenCL and Metal backends.

Implementation source is published at *https://github.com/Cynosureprime/mdxfind* and *https://github.com/Cynosureprime/hashpipe*, and the troff source of this specification at *https://github.com/Cynosureprime/hx*. As the standard description format for hash computations, hx bridges the gap between the compressed type names used by mdxfind internally and the human-readable descriptions needed by researchers, documentation, and the broader security community.

13. Appendix A: mdxfind Type to hx Expression Reference

Type	Name	hx Expression
e1	MD5	md5(pass)
e2	MD5UC	md5_uc(pass)
e3	MD4	md4(pass)
e4	MD2	md2(pass)
e5	WRL	wrl(pass)
e6	HAV128	hav128(pass)
e7	SHA0	sha0(pass)
e8	SHA1	sha1(pass)
e9	SHA224	sha224(pass)
e10	SHA256	sha256(pass)
e11	SHA384	sha384(pass)
e12	SHA512	sha512(pass)
e13	GOST	gost(pass)
e14	GOST-CRYPTO	gost_crypto(pass)
e15	HAV256	hav256(pass)
e16	RMD128	rmd128(pass)
e17	RMD160	rmd160(pass)
e18	TIGER	tiger(pass)
e19	TTH	tth(pass)
e20	ED2K	ed2k(pass)
e21	AICH	aich(pass)
e22	HAS160	has160(pass)
e23	EDON256	edon256(pass)
e24	EDON512	edon512(pass)
e25	SNE128	sne128(pass)
e26	SNE256	sne256(pass)
e27	MD6	md6(pass)
e28	MD6128	md6128(pass)
e29	MD6256	md6256(pass)
e30	MD6512	md6512(pass)
e31	MD5SALT	md5(md5(pass) . salt)
e32	MDC2	mdc2(pass)
e33	MD5RAW	md5(md5_bin(pass))
e34	SHA1RAW	sha1(sha1_bin(pass))
e35	SHA224RAW	sha224(sha224_bin(pass))
e36	SHA256RAW	sha256(sha256_bin(pass))
e37	SHA384RAW	sha384(sha384_bin(pass))
e38	SHA512RAW	sha512(sha512_bin(pass))
e39	HAV128-4	hav128_4(pass)
e40	HAV128-5	hav128_5(pass)
e41	HAV160-3	hav160_3(pass)
e42	HAV160-4	hav160_4(pass)
e43	HAV160-5	hav160_5(pass)
e44	HAV192-3	hav192_3(pass)
e45	HAV192-4	hav192_4(pass)
e46	HAV192-5	hav192_5(pass)
e47	HAV224-3	hav224_3(pass)
e48	HAV224-4	hav224_4(pass)
e49	HAV224-5	hav224_5(pass)
e50	HAV256-4	hav256_4(pass)

Type	Name	hx Expression
e51	HAV256-5	hav256_5(pass)
e52	BLAKE224	blake224(pass)
e53	BLAKE256	blake256(pass)
e54	BLAKE384	blake384(pass)
e55	BLAKE512	blake512(pass)
e56	BMW224	bmw224(pass)
e57	BMW256	bmw256(pass)
e58	BMW384	bmw384(pass)
e59	BMW512	bmw512(pass)
e60	CUBE224	cube224(pass)
e61	CUBE256	cube256(pass)
e62	CUBE384	cube384(pass)
e63	CUBE512	cube512(pass)
e64	ECHO224	echo224(pass)
e65	ECHO256	echo256(pass)
e66	ECHO384	echo384(pass)
e67	ECHO512	echo512(pass)
e68	FUGUE224	fugue224(pass)
e69	FUGUE256	fugue256(pass)
e70	FUGUE384	fugue384(pass)
e71	FUGUE512	fugue512(pass)
e72	GROESTL224	groestl224(pass)
e73	GROESTL256	groestl256(pass)
e74	GROESTL384	groestl384(pass)
e75	GROESTL512	groestl512(pass)
e76	HAMSI224	hamsi224(pass)
e77	HAMSI256	hamsi256(pass)
e78	HAMSI384	hamsi384(pass)
e79	HAMSI512	hamsi512(pass)
e80	JH224	jh224(pass)
e81	JH256	jh256(pass)
e82	JH384	jh384(pass)
e83	JH512	jh512(pass)
e84	KECCAK224	keccak224(pass)
e85	KECCAK256	keccak256(pass)
e86	KECCAK384	keccak384(pass)
e87	KECCAK512	keccak512(pass)
e88	SHA3-224	sha3_224(pass)
e89	SHA3-256	sha3_256(pass)
e90	SHA3-384	sha3_384(pass)
e91	SHA3-512	sha3_512(pass)
e92	LUFFA224	luffa224(pass)
e93	LUFFA256	luffa256(pass)
e94	LUFFA384	luffa384(pass)
e95	LUFFA512	luffa512(pass)
e96	PANAMA	panama(pass)
e97	RADIOGATUN32	radiogatun32(pass)
e98	RADIOGATUN64	radiogatun64(pass)
e99	SHABAL224	shabal224(pass)
e100	SHABAL256	shabal256(pass)
e101	SHABAL384	shabal384(pass)
e102	SHABAL512	shabal512(pass)

Type	Name	hx Expression
e103	SHAVITE224	shavite224(pass)
e104	SHAVITE256	shavite256(pass)
e105	SHAVITE384	shavite384(pass)
e106	SHAVITE512	shavite512(pass)
e107	SIMD224	simd224(pass)
e108	SIMD256	simd256(pass)
e109	SIMD384	simd384(pass)
e110	SIMD512	simd512(pass)
e111	SKEIN224	skein224(pass)
e112	SKEIN256	skein256(pass)
e113	SKEIN384	skein384(pass)
e114	SKEIN512	skein512(pass)
e115	TIGER2	tiger2(pass)
e116	WRL0	wrl0(pass)
e117	WRL1	wrl1(pass)
e118	RIPEMD	ripemd(pass)
e119	MD2MD5	md2(md5(pass))
e120	MD2MD5PASS	md2(md5(pass).pass)
e121	MD4MD5	md4(md5(pass))
e122	MD4MD5PASS	md4(md5(pass).pass)
e123	MD5MD5PASS	md5(md5(pass).pass) *(Note [24])*
e124	GOSTMD5	gost(md5(pass))
e125	GOSTMD5PASS	gost(md5(pass).pass)
e126	HAV128MD5	hav128(md5(pass))
e127	HAV128MD5PASS	hav128(md5(pass).pass)
e128	HAV128-4MD5	hav128_4(md5(pass))
e129	HAV128-4MD5PASS	hav128_4(md5(pass).pass)
e130	HAV128-5MD5	hav128_5(md5(pass))
e131	HAV128-5MD5PASS	hav128_5(md5(pass).pass)
e132	HAV160-3MD5	hav160_3(md5(pass))
e133	HAV160-3MD5PASS	hav160_3(md5(pass).pass)
e134	HAV160-4MD5	hav160_4(md5(pass))
e135	HAV160-4MD5PASS	hav160_4(md5(pass).pass)
e136	HAV160-5MD5	hav160_5(md5(pass))
e137	HAV160-5MD5PASS	hav160_5(md5(pass).pass)
e138	HAV192-3MD5	hav192_3(md5(pass))
e139	HAV192-3MD5PASS	hav192_3(md5(pass).pass)
e140	HAV192-4MD5	hav192_4(md5(pass))
e141	HAV192-4MD5PASS	hav192_4(md5(pass).pass)
e142	HAV192-5MD5	hav192_5(md5(pass))
e143	HAV192-5MD5PASS	hav192_5(md5(pass).pass)
e144	HAV224-3MD5	hav224_3(md5(pass))
e145	HAV224-3MD5PASS	hav224_3(md5(pass).pass)
e146	HAV224-4MD5	hav224_4(md5(pass))
e147	HAV224-4MD5PASS	hav224_4(md5(pass).pass)
e148	HAV224-5MD5	hav224_5(md5(pass))
e149	HAV224-5MD5PASS	hav224_5(md5(pass).pass)
e150	HAV256MD5	hav256(md5(pass))
e151	HAV256MD5PASS	hav256(md5(pass).pass)
e152	HAV256-4MD5	hav256_4(md5(pass))
e153	HAV256-4MD5PASS	hav256_4(md5(pass).pass)
e154	HAV256-5MD5	hav256_5(md5(pass))

Type	Name	hx Expression
e155	HAV256-5MD5PASS	hav256_5(md5(pass).pass)
e156	RMD128MD5	rmd128(md5(pass))
e157	RMD128MD5PASS	rmd128(md5(pass).pass)
e158	RMD160MD5	rmd160(md5(pass))
e159	RMD160MD5PASS	rmd160(md5(pass).pass)
e160	SHA1MD5	sha1(md5(pass))
e161	SHA1MD5PASS	sha1(md5(pass).pass)
e162	SHA224MD5	sha224(md5(pass))
e163	SHA224MD5PASS	sha224(md5(pass).pass)
e164	SHA256MD5	sha256(md5(pass))
e165	SHA256MD5PASS	sha256(md5(pass).pass)
e166	SHA384MD5	sha384(md5(pass))
e167	SHA384MD5PASS	sha384(md5(pass).pass)
e168	SHA512MD5	sha512(md5(pass))
e169	SHA512MD5PASS	sha512(md5(pass).pass)
e170	TIGERMD5	tiger(md5(pass))
e171	TIGERMD5PASS	tiger(md5(pass).pass)
e172	WRLMD5	wrl(md5(pass))
e173	WRLMD5PASS	wrl(md5(pass).pass)
e174	SNE128MD5	sne128(md5(pass))
e175	SNE128MD5PASS	sne128(md5(pass).pass)
e176	SNE256MD5	sne256(md5(pass))
e177	SNE256MD5PASS	sne256(md5(pass).pass)
e178	MD5SHA1	md5(sha1(pass))
e179	MD5SHA256	md5(sha256(pass))
e180	MD5SHA512	md5(sha512(pass))
e181	SHA1PASSSHA1	sha1(pass . sha1(pass))
e182	SHA1UC	sha1_uc(pass)
e183	MD5HEXSALT	md5(pass . salt) *(Note [24])*
e184	SHA1HEXSALT	sha1(pass . salt) *(Note [24])*
e185	SHA256HEXSALT	sha256(pass . salt) *(Note [24])*
e186	GOSTHEXSALT	gost(pass . salt) *(Note [24])*
e187	HAV128HEXSALT	hav128(pass . salt) *(Note [24])*
e188	MD5SHA1MD5	md5(sha1(md5(pass)))
e189	MD5SHA1MD5SHA1	md5(sha1(md5(sha1(pass))))
e190	MD5SHA1MD5SHA1SHA1	md5(sha1(md5(sha1(sha1(pass)))))
e191	SHA1MD5SHA1	sha1(md5(sha1(pass)))
e192	SHA1MD5SHA1MD5	sha1(md5(sha1(md5(pass))))
e193	MD5HAV160-3	md5(hav160_3(pass))
e194	MD5SHA1HAV160-4	md5(sha1(hav160_4(pass)))
e195	MD5SHA1MD5SHA1MD5	md5(sha1(md5(sha1(md5(pass)))))
e196	MD5RMD160	md5(rmd160(pass))
e197	MD5SHA1SHA256	md5(sha1(sha256(pass)))
e198	SHA256SHA512	sha256(sha512(pass))
e199	MD5WRL	md5(wrl(pass))
e200	MD5SHA1SHA1RAW	md5(sha1(sha1_bin(pass)))
e201	MD5PASSMD5	md5(pass . md5(pass)) *(Note [24])*
e202	MD5-DBL-PASS	h=md5(pass); emit(md5(h.md5(h))); emit(md5(h.":".md5(h)))
e203	SHA1MD5UC	sha1(upper(md5(pass)))
e204	SHA1MD5SHA1MD5SHA1MD5	sha1(md5(sha1(md5(sha1(md5(pass))))))
e205	MD5SQL5	md5("*" . upper(sha1(sha1_bin(pass))))
e206	SHA1SQL5	*multi-emit SQL5 construction — see Note [1]*

Type	Name	hx Expression
e207	MD5MD5UCMD5	md5(upper(md5(md5(pass))))
e208	HMAC-MD2	hmac_md2(pass, user)
e209	HMAC-MD4	hmac_md4(pass, user)
e210	HMAC-RMD128	hmac_rmd128(pass, user)
e211	HMAC-RMD160	hmac_rmd160(pass, user)
e212	HMAC-RMD256	hmac_rmd256(pass, user)
e213	HMAC-RMD320	hmac_rmd320(pass, user)
e214	HMAC-MD5	hmac_md5(pass, user)
e215	HMAC-SHA1	hmac_sha1(pass, user)
e216	HMAC-SHA224	hmac_sha224(pass, user)
e217	HMAC-SHA256	hmac_sha256(pass, user)
e218	HMAC-SHA512	hmac_sha512(pass, user)
e219	HMAC-HAV128	hmac_hav128(pass, user)
e220	HMAC-HAV160	hmac_hav160(pass, user)
e221	HMAC-HAV192	hmac_hav192(pass, user)
e222	HMAC-HAV224	hmac_hav224(pass, user)
e223	HMAC-HAV256	hmac_hav256(pass, user)
e224	HMAC-TIGER128	hmac_tiger128(pass, user)
e225	HMAC-TIGER160	hmac_tiger160(pass, user)
e226	HMAC-TIGER192	hmac_tiger192(pass, user)
e227	HMAC-GOST	hmac_gost(pass, user)
e228	HMAC-WRL	hmac_wrl(pass, user) *(Note [30])*
e229	HMAC-SNE128	hmac_sne128(pass, user)
e230	HMAC-SNE256	hmac_sne256(pass, user)
e231	RMD128MD5MD5	rmd128(md5(md5(pass)))
e232	MD5BASE64	md5(base64(pass))
e233	MD5BASE64MD5	md5(base64(md5(pass)))
e234	MD5SHA1BASE64	md5(sha1(base64(pass)))
e235	MD5SHA1MD5BASE64	md5(sha1(md5(base64(pass))))
e236	SHA1BASE64	sha1(base64(pass))
e237	MD5BASE64MD5RAW	md5(base64(md5_bin(pass))) *(Note [24])*
e238	MD5BASE64SHA1RAW	md5(base64(sha1_bin(pass))) *(Note [24])*
e239	SHA1BASE64MD5RAW	sha1(base64(md5_bin(pass)))
e240	SHA1BASE64SHA1RAW	sha1(base64(sha1_bin(pass)))
e241	MD5BASE64SHA1RAWMD5	md5(base64(sha1_bin(md5(pass))))
e242	MD5BASE64MD5RAWSHA1	md5(base64(md5_bin(sha1(pass))))
e243	MD5BASE64MD5RAWMD5	md5(base64(md5_bin(md5(pass))))
e244	MD5BASE64MD5RAWMD5MD5	md5(base64(md5_bin(md5(md5(pass)))))
e245	MD5BASE64SHA1RAWBASE64SHA1RAW	md5(base64(sha1_bin(base64(sha1_bin(pass)))))
e246	MD5SHA1UC	md5(upper(sha1(pass)))
e247	MD5SHA1UCMD5	md5(upper(sha1(md5(pass))))
e248	MD5SHA1UCMD5UC	md5(upper(sha1(upper(md5(pass)))))
e249	SHA1SHA1u32	sha1(cut(sha1(pass), 0, 32))
e250	MD5SHA1u32	md5(cut(sha1(pass), 0, 32))
e251	SHA256SHA1	sha256(sha1(pass))
e252	MD5BASE64revMD5	md5(base64(rev(md5(pass))))
e253	MD5revMD5	md5(rev(md5(pass)))
e254	MD5revMD5MD5	md5(rev(md5(md5(pass))))
e255	MD5BASE64MD5MD5	md5(base64(md5(md5(pass))))
e256	MD5SQL5-40	md5(upper(sha1(sha1_bin(pass))))
e257	SHA1SQL5-40	sha1(upper(sha1(sha1_bin(pass))))
e258	MD5revSHA1	md5(rev(sha1(pass)))

Type	Name	hx Expression
e259	SQL5	"*" . upper(sha1(sha1_bin(pass)))
e260	MD5USERIDMD5	md5(user . md5(pass))
e261	MD5USERIDMD5MD5	md5(user . md5(md5(pass)))
e262	MD5MD5UC	md5(upper(md5(pass)))
e263	MD5USERnulPASS	md5(user . fromhex("00") . pass)
e264	RADMIN2	md5(pad(pass, 100))
e265	MD5RADMIN2	md5(md5(pad(pass, 100)))
e266	MD5RADMIN2SHA1	md5(md5(pad(sha1(pass), 100)))
e267	RADMIN2MD5	md5(pad(md5(pass), 100))
e268	RADMIN2MD5MD5	md5(pad(md5(md5(pass)), 100))
e269	RADMIN2MD5UC	md5(pad(upper(md5(pass)), 100))
e270	RADMIN2MD5SHA1	md5(pad(md5(sha1(pass)), 100))
e271	RADMIN2SHA1	md5(pad(sha1(pass), 100))
e272	RADMIN2SQL5-40	md5(pad(upper(sha1(sha1_bin(pass))), 100))
e273	RADMIN2BASE64	md5(pad(base64(pass), 100))
e274	RADMIN2SHA1MD5	md5(pad(sha1(md5(pass)), 100))
e275	RADMIN2SQL3	md5(pad(mysql3(pass), 100))
e276	RADMIN2MD5MD5MD5	md5(pad(md5(md5(md5(pass))), 100))
e277	MD5RADMIN2MD5	md5(md5(pad(md5(pass), 100)))
e278	MD51SALTMD5	*(complex: iterates over single-char salts)*
e279	SHA1RADMIN2	sha1(md5(pad(pass, 100)))
e280	SHA1RADMIN2MD5	sha1(md5(pad(md5(pass), 100)))
e281	SHA1RADMIN2BASE64	sha1(md5(pad(base64(pass), 100)))
e282	MD5-MULTISALT	md5(salt . md5(md5(salt . pass) . salt) . salt)
e283	MD52SALTMD5	md5(salt . salt . md5(pass)) *(Note [24])*
e284	MD51SALTMD5UC	*(complex: iterates single-char salts, UC)*
e285	MD51SALTMD5MD5	*(complex: iterates single-char salts, MD5MD5)*
e286	MD5MD5USER	md5(md5(pass).user) *(Note [24])*
e287	SHA1MD5MD5	sha1(md5(md5(pass)))
e288	SHA1SHA1RAWMD5	sha1(sha1_bin(md5(pass)))
e289	SHA1MD5MD5MD5	sha1(md5(md5(md5(pass))))
e290	SHA1MD5MD5SHA1MD5	sha1(md5(md5(sha1(md5(pass)))))
e291	SHA1MD5MD5SHA1	sha1(md5(md5(sha1(pass))))
e292	SHA1MD5MD5MD5SHA1	sha1(md5(md5(md5(sha1(pass)))))
e293	SHA1MD5MD5MD5MD5	sha1(md5(md5(md5(md5(pass)))))
e294	SHA1MD5MD5MD5MD5MD5	sha1(md5(md5(md5(md5(md5(pass))))))
e295	SHA1SHA1RAWMD5MD5	sha1(sha1_bin(md5(md5(pass))))
e296	SHA1SHA1RAWMD5MD5MD5	sha1(sha1_bin(md5(md5(md5(pass)))))
e297	SHA1MD51SALTMD5	*(complex: iterates over single-char salts)*
e298	SHA1MD5x	sha1(md5^N(pass)) *(N = iteration count)*
e299	MD5SHA1SHA1MD5	md5(sha1(sha1(md5(pass))))
e300	MD5SHA1SHA1	md5(sha1(sha1(pass)))
e301	MD5SQL5MD5	md5("*" . upper(sha1(sha1_bin(md5(pass)))))
e302	MD5SQL5-chop40	md5(cut("*" . upper(sha1(sha1_bin(pass))), 0, 40))
e303	MD5-2xMD5	h=md5(pass); md5(h . h)
e304	MD5-2xMD5-MD5	h=md5(md5(pass)); md5(h . h)
e305	MD5-2xMD5-SHA1	h=md5(sha1(pass)); md5(h . h)
e306	MD5-2xMD5-MD5MD5	h=md5(md5(md5(pass))); md5(h . h)
e307	MD5-2xMD5-MD5MD5MD5	h=md5(md5(md5(md5(pass)))); md5(h . h)
e308	SHA1MD5USER	emit(sha1(md5(pass) . user)) *(Note [24]; see Note [48])*
e309	SHA1MD5RADMIN2	sha1(md5(md5(pad(pass, 100))))
e310	SHA1SHA1USER	sha1(sha1(pass) . user)

Type	Name	hx Expression
e311	SHA11SALTMD5	*(complex: iterates over single-char salts)*
e312	MD5-3xMD5	h=md5(pass); md5(h . h . h)
e313	MD5-3xMD5-MD5	h=md5(md5(pass)); md5(h . h . h)
e314	MD5-3xMD5-SHA1	h=md5(sha1(pass)); md5(h . h . h)
e315	MD5-3xMD5-MD5MD5	h=md5(md5(md5(pass))); md5(h . h . h)
e316	MD5-3xMD5-MD5MD5MD5	h=md5(md5(md5(md5(pass)))); md5(h . h . h)
e317	SHA1-2xSHA1	h=sha1(pass); sha1(h . h)
e318	SHA1-2xSHA1-MD5	h=sha1(md5(pass)); sha1(h . h)
e319	SHA1-2xSHA1-SHA1	h=sha1(sha1(pass)); sha1(h . h)
e320	SHA1-2xSHA1-MD5MD5	h=sha1(md5(md5(pass))); sha1(h . h)
e321	SHA1-2xSHA1-MD5MD5MD5	h=sha1(md5(md5(md5(pass)))); sha1(h . h)
e322	MD5-1xMD5SHA1	md5(md5(pass) . sha1(pass))
e323	MD5-1xSHA1MD5	md5(sha1(pass) . md5(pass))
e324	SHA1-1xSHA1MD5	sha1(sha1(pass) . md5(pass))
e325	SHA1-1xMD5SHA1	sha1(md5(pass) . sha1(pass))
e326	MD5-1xMD5SHA1-MD5	md5(md5(md5(pass)) . sha1(md5(pass)))
e327	MD5-1xSHA1MD5-MD5	md5(sha1(md5(pass)) . md5(md5(pass)))
e328	MD5-1xMD5SHA1-MD5MD5	md5(md5(md5(md5(pass))) . sha1(md5(md5(pass))))
e329	MD5-1xSHA1MD5-MD5MD5	md5(sha1(md5(md5(pass))) . md5(md5(md5(pass))))
e330	MD5-1xSHA1MD5pSHA1p	md5(sha1(md5(pass)) . sha1(pass))
e331	SHA1-1xSHA1MD5pSHA1p	sha1(sha1(md5(pass)) . sha1(pass))
e332	MD5SHA1-1xSHA1MD5pSHA1p	md5(sha1(sha1(md5(pass)) . sha1(pass)))
e333	SHA1-1xSHA1psubp	*(complex: iterates over password substrings)*
e334	SHA1revp	sha1(rev(pass))
e335	MD5revp	md5(rev(pass))
e336	MD5TIGER2	md5(tiger2(pass))
e337	MD5SHA1MD5MD5	md5(sha1(md5(md5(pass))))
e338	MD5SHA1MD5RADMIN2	md5(sha1(md5(md5(pad(pass, 100)))))
e339	MD5SHA1MD5MD5MD5SHA1	md5(sha1(md5(md5(md5(sha1(pass))))))
e340	MD5SHA1RAW	md5(sha1_bin(pass))
e341	MD5SHA1MD5MD5MD5	md5(sha1(md5(md5(md5(pass)))))
e342	MD5SQL5-32	md5(cut("*" . upper(sha1(sha1_bin(pass))), 0, 32))
e343	MD51SALTMD5MD5MD5	*(complex: iterates single-char salts)*
e344	MD51SALTMD5MD5MD5MD5	*(complex: iterates single-char salts)*
e345	MD51SALTMD5MD5MD5MD5MD5	*(complex: iterates single-char salts)*
e346	MD5BASE64ROT13	md5(base64(rot13(pass)))
e347	MD5MD5SALT	md5(md5(md5(pass)) . salt)
e348	MD52SALTMD5MD5	*(complex: double-salt iterator)*
e349	MD52SALTMD5MD5MD5	*(complex: double-salt iterator)*
e350	MD5UCSALT	md5(upper(md5(pass)) . salt)
e351	MD5TIGER	md5(tiger(pass))
e352	MD5SHA1MD5PASS	md5(sha1(md5(pass)) . pass)
e353	MD5CAP	cap(md5(pass))
e354	MD5CAPMD5USER	emit(md5(cap(md5(pass)) . user)) *(Note [24]; see Note [48])*
e355	MD5CAPMD5MD5USER	emit(md5(cap(md5(cap(md5(pass)))) . user)) *(Note [24]; see Note [48])*
e356	MD5MD5MD5USER	emit(md5(md5(md5(pass)) . user)) *(Note [24]; see Note [48])*
e357	MD5MD5SALT-SALT	md5(md5(pass . salt) . ":" . salt)
e358	MD5CAPSHA1	md5(cap(sha1(pass)))
e359	MD5SHA1MD5x	md5(sha1(md5^N(pass))) *(N = iteration count)*
e360	MD5SHA1BASE64MD5RAW	md5(sha1(base64(md5_bin(pass))))
e361	MD5SHA1MD5MD5SHA1	md5(sha1(md5(md5(sha1(pass)))))
e362	MD5SHA1MD5UC	md5(sha1(upper(md5(pass))))

Type	Name	hx Expression
e363	MD5MD5HUM	emit(md5(md5^N(pass) . S)) *(Note [24])*
e364	SHA1MD5HUM	emit(sha1(md5^N(pass) . S)) *(Note [24])*
e365	SHA1SHA1HUM	emit(sha1(sha1^N(pass) . S)) *(Note [24])*
e366	MD5SHA1HUM	emit(md5(sha1^N(pass) . S)) *(Note [24])*
e367	MD5-MD5SALTMD5PASS	md5(md5(salt) . md5(pass))
e368	MD5NTLM	md5(md4(utf16le(pass)))
e369	NTLM	md4(utf16le(pass)) *(Note [29])*
e370	MD5MD4	md5(md4(pass))
e371	MD5NTLMUC	md5(upper(md4(utf16le(pass))))
e372	MD5USERPASS	md5(user . pass)
e373	MD5PASSSALT	md5(pass . salt)
e374	MD5SHA1RADMIN2MD5	md5(sha1(md5(pad(md5(pass), 100))))
e375	MD5UCBASE64MD5RAW	md5_uc(base64(md5_bin(pass))) *(Note [24])*
e376	MD5UCBASE64SHA1RAW	md5_uc(base64(sha1_bin(pass))) *(Note [24])*
e377	MD5SHA1MD5MD5UC	md5(sha1(md5(upper(md5(pass)))))
e378	MD5SHA1MD5HUM	emit(md5(sha1(md5^N(pass) . S))) *(Note [24])*
e379	LM	lm(pass)
e380	MD5LM	md5(lm(pass))
e381	MD5LMUC	md5(upper(lm(pass)))
e382	SHA256SHA256SALT	sha256(sha256(pass) . salt)
e383	MD5RAWUC	hex(upper(md5_bin(pass)))
e384	MD5SHA1UCu32	md5(cut(upper(sha1(pass)), 0, 32))
e385	SHA1SALTPASS	sha1(salt . pass)
e386	SHA512PASSSALT	sha512(pass . salt)
e387	SHA512SHA512SALT	sha512(sha512(pass) . salt)
e388	SHA512SALTPASS	sha512(salt . pass)
e389	SHA512SALTSHA512	sha512(salt . sha512(pass))
e390	WRLPASSSALT	wrl(pass . salt)
e391	WRLWRLSALT	wrl(wrl(pass) . salt)
e392	WRLSALTPASS	wrl(salt . pass)
e393	WRLSALTWRL	wrl(salt . wrl(pass))
e394	MD5SALTPASS	md5(salt . pass)
e395	SHA1SALTPASSSALT	sha1(salt . pass . salt)
e396	MD5MD5UCSHA1MD5MD5	md5(upper(md5(sha1(md5(md5(pass))))))
e397	MD5SHA256MD5	md5(sha256(md5(pass)))
e398	SHA1lsb32	"00000000" . cut(sha1(pass), 8)
e399	MD5-2xMD5UC	h=upper(md5(pass)); md5(h . h)
e400	MD5-4xMD5	h=md5(pass); md5(h . h . h . h)
e401	MD5-SHA1numSHA1	*(complex: iterates digits 1–9 as separator)*
e402	MD5-2xSHA1	h=sha1(pass); md5(h . h)
e403	SHA1-2xMD5	h=md5(pass); sha1(h . h)
e404	SHA1DRU	h=sha1_bin(pass); for i=1 to 1000000 { h=sha1_bin(sha1(h).pass) }; sha1(h)
e405	SHA1PASSSALT	sha1(pass . salt)
e406	SHA256MD5SALTPASS	sha256(md5(salt . pass))
e407	MD5padMD5	md5(" " . md5(pass) . " ")
e408	MD5DSALT	md5(md5(md5(pass) . salt) . salt2) *(see Note [50])*
e409	SHA1lsb35	hex(and(sha1_bin(pass), "\x00\x0f\xff...\xff")) *(nibble mask)*
e410	MD4SQL3	md4(mysql3(pass))
e411	MD5SALTPASSSALT	md5(salt . pass . salt)
e412	SHA256SALTPASS	sha256(salt . pass)
e413	SHA256PASSSALT	sha256(pass . salt)
e414	SMF	sha1(lower(user) . pass)

Type	Name	hx Expression
e415	MD5-MD5SHA1MD5SHA1MD5SHA1p	md5(md5(sha1(md5(sha1(md5(sha1(pass)))))) . " ")
e416	MD5MD5UCSQL3p	md5(upper(md5(mysql3(pass))) . " ")
e417	MD5MD5UCp	md5(upper(md5(pass)) . " ")
e418	MD5NTLMp	md5(md4(utf16le(pass)) . " ")
e419	MD5-MD5USERSHA1MD5PASS	md5(md5(user) . sha1(md5(pass)))
e420	MD5SHA1u32SALT	md5(cut(sha1(pass), 0, 32) . salt)
e421	MD5-LMNTLM	md5(lm(pass) . md4(utf16le(pass)))
e422	MD5-MD5psSHA1MD5psp	h=md5(pass.salt); md5(h . sha1(h . pass))
e423	MD5-MD5puSHA1MD5pup	h=md5(pass.user); md5(h . sha1(h . pass))
e424	MD5WRLMD5	md5(wrl(md5(pass)))
e427	GOST2012-32	gost2012_32(pass)
e428	GOST2012-64	gost2012_64(pass)
e429	POMELO	pomelo(pass, salt, t, m)
e430	STREEBOG-32	streebog_32(pass)
e431	STREEBOG-64	streebog_64(pass)
e432	MD5AM	md5(user . "::" . lower(pass)) *(Note [26])*
e433	MD5AM2	md5(user . "::" . lower(pass) . ":" . salt2 . ":73@^bhhs&#@&^@8@*$") *(Note [26])*
e434	MD5SWAP	h=md5(pass); md5(cut(h,16) . cut(h,0,16))
e435	MD5SPECAM	*parsing wrapper around MD5AM and MD5AM2 — see Note [2]*
e436	SHA1HESK	h=""; for i=0 to length(pass)-1 { h=h.sha1(cut(pass,i,1)) }; sha1(h)
e437	MD5HESK	h=""; for i=0 to length(pass)-1 { h=h.md5(cut(pass,i,1)) }; md5(h)
e438	SHA1SALTSHA1SALTSHA1PASS	sha1(salt . sha1(salt . sha1(pass)))
e439	MSCACHE	md4(md4_bin(utf16le(pass)) . utf16le(lower(user)))
e440	MD5SHA1SALTMD5PASS	md5(sha1(salt) . md5(pass))
e441	MD5SALTMD5PASS	md5(salt . md5(pass))
e442	MD5SHA1PASSMD5PASSSHA1PASS	md5(sha1(pass) . md5(pass) . sha1(pass))
e443	MD5SHA1PASSSALT	md5(sha1(pass . salt))
e444	LEET-SHA512-WRL-USER	emit(sha512(pass . user)); emit(wrl(user . pass))
e445	SHA1-SALT-SPECIAL	*iterated dash-delimited SHA-1 — see Note [4]*
e446	SHA1-SALT-UTF16-PEPPER	sha1(utf16le(salt . rev(lower(pass)) . pepper))
e447	SHA256RAWSALTPASS	sha256_bin(salt . pass)
e448	SHA512-CUSTOM1	sha512(md5(sha1(md5("$3dfhgjhgG65-" . pass . "23ewdfwGh5RG65?"))))
e449	MD5SQL3	md5(mysql3(pass))
e450	BCRYPT	bcrypt(pass, salt, 12)
e451	BCRYPTMD5	bcrypt(md5(pass), salt, 12)
e452	BCRYPTSHA1	bcrypt(sha1(pass), salt, 12)
e453	MD5sub8-24MD5	md5(cut(md5(pass), 8, 16))
e454	MD5sub8-24MD5sub8-24MD5	md5(cut(md5(cut(md5(pass), 8, 16)), 8, 16))
e455	PHPBB3	*phpass / itoa64-encoded MD5 chain — see Note [5]*
e456	MYSQL3	mysql3(pass)
e457	APACHE-SHA	*"{SHA}" . base64(sha1_bin(pass))*
e458	MD5-MD5PASSMD5SALT	md5(md5(pass) . md5(salt))
e459	YAF-SHA1	base64(sha1_bin(utf16le(pass) . frombase64(salt)))
e460	MD5revMD5SHA1	md5(rev(md5(sha1(pass))))
e461	APR1	apr1(pass, salt)
e462	MD5SHA1BASE64SHA1MD5	md5(sha1(base64(sha1(md5(pass)))))
e463	SHA1-8TRACK	sha1(salt . pass . "--") *(8track.com)*
e464	SHA1WRL	sha1(wrl(pass))
e465	SHA1revMD5	sha1(rev(md5(pass)))
e466	SHA1-MD5SALT	sha1(md5(md5(pass) . salt))
e467	SHA1-revMD5SALT	sha1(md5(rev(md5(pass)) . salt))
e468	SHA1revMD5PASSSALT	sha1(rev(md5(pass)) . salt)

Type	Name	hx Expression
e469	SHA1MD5PASSSALT	sha1(md5(pass . salt))
e470	SHA1-MD5MD5SALT	sha1(md5(md5(md5(pass)) . salt))
e471	SHA1MD5PASS-SALT	sha1(md5(pass) . salt)
e472	SHA1SALTMD5PASS	sha1(salt . md5(pass))
e473	SHA1SALTrevMD5PASS	sha1(salt . rev(md5(pass)))
e474	SHA1USERSQL3	sha1(user . ":" . mysql3(pass))
e475	SHA1SHA256	sha1(sha256(pass))
e476	SHA1SHA384	sha1(sha384(pass))
e477	SHA1SHA512	sha1(sha512(pass))
e478	SHA1SQL3	sha1(mysql3(pass))
e479	SHA1UCWRL	sha1(upper(wrl(pass)))
e480	SHA1WRLMD5	sha1(wrl(md5(pass)))
e481	SHA1MD5SHA1MD5SHA1MD5SHA1	sha1(md5(sha1(md5(sha1(md5(sha1(pass)))))))
e482	WRLSHA512	wrl(sha512(pass))
e483	MD5GOST	md5(gost(pass))
e484	MD5GOSTMD5	md5(gost(md5(pass)))
e485	MD5GOSTMD5UC	md5(gost(upper(md5(pass))))
e486	MD5WRLRAW	md5(wrl_bin(pass))
e487	MD5-4xMD5-SALT	h=md5(pass); md5(h . h . h . h . salt)
e488	SHA256UC	sha256_uc(pass)
e489	MD4SHA1MD5	md4(sha1(md5(pass)))
e490	MD5RAWMD5RAW	md5(md5_bin(md5_bin(pass)))
e491	MD5SHA1BASE64SHA1RAW	md5(sha1(base64(sha1_bin(pass))))
e492	MD5revMD5SHA1SHA1	md5(rev(md5(sha1(sha1(pass)))))
e493	MD5MD2	md5(md2(pass))
e494	MD5MD2RAW	md5(md2_bin(pass))
e495	MD5BASE64SHA256RAW	md5(base64(sha256_bin(pass)))
e496	MD4UTF16	md4(utf16le(pass))
e497	MD4UTF16MD5	md4(utf16le(md5(pass)))
e498	RMD128MD4	rmd128(md4(pass))
e499	MD5SHA0	md5(sha0(pass))
e500	DESCRYPT	descrypt(pass, salt) *(salt = 2-char crypt salt)*
e501	MD5DESCRYPT	md5(descrypt(pass, salt))
e502	MD4DESCRYPT	md4(descrypt(pass, salt))
e503	MD5PASSSHA1MD5	md5(pass . sha1(md5(pass)))
e504	MD5PASSMD5MD5PASS	md5(pass . md5(md5(pass) . pass))
e505	MD5PASSMD5MD5MD5	md5(pass . md5(md5(md5(pass))))
e506	MD5-MD5PASSMD5	md5(md5(pass) . pass . md5(pass))
e507	MD5PASSSHA1	md5(pass . sha1(pass))
e508	MD5MD5PASSSHA1	md5(md5(pass) . pass . sha1(pass))
e509	SHA1MD5MD5SHA1MD5SHA1SHA1MD5	sha1(md5(md5(sha1(md5(sha1(sha1(md5(pass))))))))
e510	SHA512SHA512RAWUSER	sha512(sha512_bin(pass) . user)
e511	MD5CRYPT	md5crypt(pass, salt)
e512	SHA256CRYPT	sha256crypt(pass, salt, rounds)
e513	SHA512CRYPT	sha512crypt(pass, salt)
e514	SHA512SALTMD5	sha512(salt . md5(pass))
e515	MD5-SALTMD5PASSSALT	md5(salt . md5(pass . salt))
e516	MD5-SALTMD5SALTPASS	md5(salt . md5(salt . pass))
e517	MD5-MD5PASS-SALT	md5(md5(pass) . salt)
e518	MD5-MD5SALT-PASS	md5(md5(salt) . pass)
e519	MD5-PASS-MD5SALT	md5(pass . md5(salt))
e520	SHA1SALTSHA1PASS	sha1(salt . sha1(pass))

Type	Name	hx Expression
e521	SHA1SALTCX	*iterated SHA-1 with "--" delimiters — see Note [16]*
e522	MD5SHA1MD5SHA1MD5SHA1MD5SHA1MD5SHA1MD5SHA1	md5(sha1(md5(sha1(md5(sha1(md5(sha1(md5(sha1(pass))))))))))
e523	SHA256SALTSHA256PASS	sha256(salt . sha256(pass))
e524	MD5dcab	hex(wperm(md5_bin(pass), 3, 2, 0, 1))
e525	MD5bcad	hex(wperm(md5_bin(pass), 1, 2, 0, 3))
e526	MD5BASE64BASE64	md5(base64(base64(pass)))
e527	MD5BASE64BASE64BASE64	md5(base64(base64(base64(pass))))
e528	MD5-SALTSHA1SALTPASS	md5(salt . sha1(salt . pass))
e529	CISCO8	pbkdf2_sha256(pass, salt, 20000, 32)
e530	PBKDF2-SHA256	pbkdf2_sha256(pass, salt, N, 32)
e531	PBKDF2-MD5	pbkdf2_md5(pass, salt, N, 32)
e532	PBKDF2-SHA1	pbkdf2_sha1(pass, salt, N, 20)
e533	PBKDF2-SHA512	pbkdf2_sha512(pass, salt, N, 64)
e534	PKCS5S2	pbkdf2_sha1(pass, salt, N, 20)
e535	SHA1-CUSTOMUSERSALT	*XOR construction — see Note [25]*
e536	PROGRESSENCODE	progress_encode(pass) *(Progress OpenEdge)*
e537	PHPBB3MD5	*iterated MD5 with phpass encoding — see Note [17]*
e538	SHA512CRYPTMD5	sha512crypt(md5(pass), salt)
e539	MYSQL5MD5	sha1(sha1_bin(md5(pass)))
e540	MANGOS	sha1(upper(user) . ":" . upper(pass)) *(Note [31])*
e541	MD5revMD5SALT	md5(rev(md5(pass)) . salt)
e542	MD5sub8-24SALT	md5(cut(md5(pass), 8, 16) . salt)
e543	HMAC-SHA384	hmac_sha384(pass, user)
e544	CRYPTEXT	*byte-swapped SHA-1 with "Cryptext" tag — see Note [6]*
e545	MD5SHA1lsb35	md5(hex(and(sha1_bin(pass), fromhex("00000fffffffffffffffffffffffffffffffff"))))
e546	MD5sub1-20MD5	md5(cut(md5(pass), 0, 20))
e547	MD5sub1-20MD5MD5	md5(cut(md5(md5(pass)), 0, 20))
e548	MD5SQL3SQL5MD5MD5	md5(md5(mysql3("*".upper(sha1(sha1_bin(pass))))))
e549	SHA1MD5SHA256	sha1(md5(sha256(pass)))
e550	SHA1SHA256MD5	sha1(sha256(md5(pass)))
e551	MD4UTF16MD5MD5	md4(utf16le(md5^2(pass)))
e552	MD5-2xSHA1MD5	h=sha1(md5(pass)); md5(h . h)
e553	MD5-6xMD5	h=md5(pass); md5(h . h . h . h . h . h)
e554	MD5-5xMD5	h=md5(pass); md5(h . h . h . h . h)
e555	MD5SHA1SALT	md5(sha1(pass) . salt)
e556	MD5SHA1u39	md5(cut(sha1(pass), 0, 39))
e557	MD5-MD5SHA1PASSSHA1MD5SALT	md5(md5(sha1(pass)) . sha1(md5(salt)))
e558	MD5UCMD5	md5_uc(md5(pass))
e559	MD4UTF16MD5UC	md4(utf16le(upper(md5(pass))))
e560	MD4UTF16MD5MD5MD5	md4(utf16le(md5^3(pass)))
e561	MD4UTF16SHA1	md4(utf16le(sha1(pass)))
e562	MD4UTF16SHA1SHA1	md4(utf16le(sha1^2(pass)))
e563	MD4UTF16SHA256	md4(utf16le(sha256(pass)))
e564	MD4UTF16SHA256SHA256	md4(utf16le(sha256^2(pass)))
e565	MD4UTF16SHA256SHA256SHA256	md4(utf16le(sha256^3(pass)))
e566	MD4UTF16SHA256SHA256SHA256SHA256	md4(utf16le(sha256^4(pass)))
e567	MD4UTF16SHA1MD5	md4(utf16le(sha1(md5(pass))))
e568	MD4UTF16MD5SHA1	md4(utf16le(md5(sha1(pass))))
e569	MD4UTF16SHA256UC	md4(utf16le(upper(sha256(pass))))
e570	MD4UTF16UC	upper(md4(utf16le(pass))) *(Note [28])*
e571	SHA1SHA256UC	sha1(upper(sha256(pass)))
e572	MD5BASE64SHA1MD5	md5(base64(sha1(md5(pass))))

Type	Name	hx Expression
e573	SHA1DESCRYPT	sha1(descrypt(pass, salt))
e574	MD4UTF16DESCRYPT	descrypt(md4(utf16le(pass)), salt)
e575	MD4UTF16MD5HUM	emit(md4(utf16le(md5^N(pass) . S))) *(Note [24])*
e576	MD4UTF16SHA1HUM	emit(md4(utf16le(sha1^N(pass) . S))) *(Note [24])*
e577	BCRYPT256	bcrypt(sha256(pass), salt, 12) *(Note [24])*
e578	SHA1-MD5PASSSALT	sha1(md5(pass) . salt)
e579	SHA1SHA1PASSSALT	sha1(sha1(pass) . salt)
e580	SHA1SHA256x	sha1(sha256^N(pass)) *(N = iteration count)*
e581	SHA1SHA256UCx	sha1(upper(sha256^N(pass))) *(N = iteration count)*
e582	SHA1SHA256UCxSHA256	sha1(sha256(upper(sha256^N(pass)))) *(N = iteration count)*
e583	SHA1SHA256UCSHA256	sha1(upper(sha256(sha256(pass))))
e584	SHA1SHA256UCSHA256SHA256	sha1(upper(sha256(sha256(sha256(pass)))))
e585	SHA1SHA256SHA512	sha1(sha256(sha512(pass)))
e586	SHA1MD5MD5SALT	sha1(md5(md5(pass)) . salt)
e587	SHA1MD5SALT	sha1(md5(pass) . salt)
e588	SHA1-MD5-MD5SALTMD5PASS	sha1(md5(md5(salt) . md5(pass)))
e589	SHA1-MD5UC-MD5SALT	sha1(upper(md5(md5(pass) . salt)))
e590	SHA1MD5DSALT	sha1(md5(pass . salt) . ":" . salt) *(Note [24])*
e591	SHA1MD5MD5DSALT	sha1(md5(md5(pass) . salt) . ":" . salt) *(Note [24])*
e592	SHA1-MD5-MD5SALTMD5PASS-SALT	sha1(md5(md5(salt) . md5(pass)) . ":" . salt)
e593	SHA1MD5MD5SHA1SHA1MD5	sha1(md5(md5(sha1(sha1(md5(pass))))))
e594	SHA1SHA1MD5PASSSALT	sha1(sha1(md5(pass)) . salt)
e595	SHA1MD5UCMD5	sha1(upper(md5(md5(pass))))
e596	SHA1-MD5PEPPER-MD5SALTMD5PASS	sha1(md5(md5(salt) . md5(pass)) . pepper)
e597	SHA1-MD5PEPPER-MD5SALT	sha1(md5(md5(pass) . salt) . pepper)
e598	SHA1MD5-PASSMD5SALT	sha1(md5(pass . md5(salt)))
e599	SHA1SHA256SHA1	sha1(sha256(sha1(pass)))
e600	SHA1MD5-SHA1PASSPASS	sha1(md5(sha1(pass) . pass))
e601	SHA1PASS-TRUNC	sha1(sha1(pass) . cut(pass, 0, 3))
e602	SHA1SALTMD5PASSPEPPER	sha1(salt . md5(pass) . pepper)
e603	SHA1MD5SALTMD5PASS	sha1(md5(salt) . md5(pass))
e604	SHA1SHA1u34	sha1(cut(sha1(pass), 0, 34))
e605	SHA1SHA1u36	sha1(cut(sha1(pass), 0, 36))
e606	SHA1SHA1u38	sha1(cut(sha1(pass), 0, 38))
e607	SHA1MD5SALTPASSPEPPER	sha1(md5(salt . pass) . pepper) *(Note [27])*
e608	SHA1SHA256u32	sha1(cut(sha256(pass), 0, 32))
e609	SHA1SHA256u40	sha1(cut(sha256(pass), 0, 40))
e610	SHA1SHA256u34	sha1(cut(sha256(pass), 0, 34))
e611	SHA1-MD5PEPPER-MD5MD5SALT	sha1(md5(md5(md5(pass)) . salt) . pepper)
e612	SHA1SHA256u42	sha1(cut(sha256(pass), 0, 42))
e613	SHA1SHA256SHA256	sha1(sha256(sha256(pass)))
e614	SHA1SHA256SHA256SHA256	sha1(sha256(sha256(sha256(pass))))
e615	SHA1SHA256u38	sha1(cut(sha256(pass), 0, 38))
e616	SHA1SHA256u36	sha1(cut(sha256(pass), 0, 36))
e617	SHA1NTLM	sha1(md4(utf16le(pass)))
e618	SHA1MD5SHA1MD5SHA1	sha1(md5(sha1(md5(sha1(pass)))))
e619	SHA1SALTSHA256	sha1(salt . sha256(pass))
e620	SHA1SHA1u35	sha1(cut(sha1(pass), 0, 35))
e621	SHA1SHA256u37	sha1(cut(sha256(pass), 0, 37))
e622	SHA1SHA256TRUNC	sha1(cut(sha256(pass), 0, N)) *(N = 21–64; see Note [24])*
e623	SHA1SHA256TRUNCMD5	sha1(cut(sha256(md5(pass)), 0, N)) *(N = 21–64; see Note [24])*
e624	SHA1SHA1u39	sha1(cut(sha1(pass), 0, 39))

Type	Name	hx Expression
e625	SHA1SHA1u37	sha1(cut(sha1(pass), 0, 37))
e626	SHA1SHA1TRUNC	sha1(cut(sha1(pass), 0, N)) *(N = 20–40; see Note [24])*
e627	SHA1-MD5SHA1PASSSHA1MD5SALT	sha1(md5(sha1(pass)) . sha1(md5(salt)))
e628	SHA1MD5CAP	sha1(cap(md5(pass)))
e629	SHA1MD5CAPMD5	sha1(cap(md5(md5(pass))))
e630	SHA1SHA256UCTRUNC	sha1(cut(upper(sha256(pass)), 0, N)) *(N = 21–64; see Note [24])*
e631	SHA1MD5MD5UCx	sha1(md5(upper(md5^N(pass)))) *(N = iteration count)*
e632	SHA1SHA256CAP	emit(sha1(cap(sha256(pass), N))) *(N = each lowercase position)*
e633	SHA1MD5SHA1-SALT	sha1(md5(sha1(pass)) . salt)
e634	SHA1SHA224	sha1(sha224(pass))
e635	SHA1WRLTRUNC	sha1(cut(wrl(pass), 0, N)) *(N = 21–128; see Note [24])*
e636	SHA1SHA512TRUNC	sha1(cut(sha512(pass), 0, N)) *(N = 21–128; see Note [24])*
e637	SHA1-SHA512PASSSHA512SALT	sha1(sha512(pass) . sha512(salt))
e638	SHA1SHA512TRUNC1SALT	sha1(cut(sha512(pass), 0, N) . salt) *(N = 21–128; salt is a generated single byte; see*
e639	SHA1SHA1PASS-TRUNC1SALT	sha1(sha1(sha1(pass) . cut(pass, 0, 3)) . salt) *(salt is a generated single byte; see Note*
e640	SHA1HAV128	sha1(hav128(pass))
e641	SHA1MD5RAW	sha1(md5(md5_bin(pass)))
e642	SHA1MD2	sha1(md2(pass))
e643	SHA1MD5BASE64	sha1(md5(base64(pass)))
e644	SHA1MD6TRUNC	sha1(cut(md6(pass), 0, N)) *(N = 20–32; see Note [24])*
e645	SHA1NTLMUC	sha1(upper(md4(utf16le(pass))))
e646	SHA1MD5SHA512	sha1(md5(sha512(pass)))
e647	SHA1MD5WRLSHA1	sha1(md5(wrl(sha1(pass))))
e648	MD5WRLSHA1	md5(wrl(sha1(pass)))
e649	SHA1MD5BASE641SALT	*single-byte salt search; mdxfind reports salt as "0xYY" — see Note [7]*
e650	SHA1-MD5PASSMD5MD5SALT	sha1(md5(pass) . md5(md5(salt)))
e651	SHA1SALTMD5UCPASSPEPPER	sha1(salt . upper(md5(pass)) . pepper)
e652	SHA1MD5UCSALT	sha1(upper(md5(pass)) . salt)
e653	SHA1WRLUCTRUNC	sha1(cut(upper(wrl(pass)), 0, N)) *(N = 21–128; see Note [24])*
e654	SHA1-MD5CAPPEPPER-MD5SALT	sha1(cap(md5(md5(pass) . salt)) . pepper)
e655	SHA1-MD5CAPSALT	sha1(cap(md5(md5(pass) . salt)))
e656	SHA1SHA1CAPTRUNC	sha1(cut(cap(sha1(pass)), 0, N)) *(N = 20–40; see Note [24])*
e657	SHA1MD6CAPTRUNC	sha1(cut(cap(md6(pass)), 0, N)) *(N = 20–32; see Note [24])*
e658	SHA1MD5UCMD5UC	sha1(upper(md5(upper(md5(pass)))))
e659	SHA1MD5CAPSALT	sha1(cap(md5(pass)) . salt)
e660	SHA1SQL5-32	sha1(cut(upper(sha1(sha1_bin(pass))), 0, 32))
e661	SHA1SHA1UCTRUNC	sha1(cut(upper(sha1(pass)), 0, N)) *(N = 20–40; see Note [24])*
e662	SHA1-MD5UCMD5UCPASSMD5UCSALT	sha1(upper(md5(upper(md5(pass)))) . upper(md5(salt)))
e663	SHA1MD5MD5PASS	emit(sha1(md5(md5(pass).pass))); emit(sha1(md5(md5(pass).":".pass) . fromhex("00
e664	SHA1MD51CAP	emit(sha1(cap(md5(pass), N))) *(N = each lowercase position)*
e665	SHA1SALTMD5UC	sha1(salt . upper(md5(pass)))
e666	SHA1SHA1CAPSALT	emit(sha1(cap(sha1(pass), N) . salt)) *(N = each lowercase position)*
e667	SHA1MD5UCMD5UCMD5UC	sha1(upper(md5(upper(md5(upper(md5(pass)))))))
e668	SHA1MD5UCMD5UCMD5UCMD5UC	sha1(upper(md5(upper(md5(upper(md5(upper(md5(pass))))))))
e669	SHA1MD51CAPSALT	emit(sha1(cap(md5(pass), N) . salt)) *(N = each lc position)*
e670	SHA1-MD5CAPMD5SALT	sha1(cap(md5(md5(md5(pass)) . salt)))
e671	SHA1SHA512UC	sha1(upper(sha512(pass)))
e672	SHA1WRLUCTRUNCSALT	sha1(cut(upper(wrl(pass)), 0, N) . salt) *(N = 21–128; see Note [24])*
e673	SHA1-MD5SHA256SALT	sha1(md5(md5(sha256(pass)) . salt))
e674	SHA256MD5SHA256MD5	sha256(md5(sha256(md5(pass))))
e675	SHA1SHA256MD5SHA256MD5	sha1(sha256(md5(sha256(md5(pass)))))
e676	SHA1-MD5sub8-24SALT	sha1(md5(cut(md5(pass), 8, 16) . salt))

Type	Name	hx Expression
e677	SHA1-PEPPER-MD5SALT	sha1(pepper . md5(md5(pass) . salt))
e678	SHA1SHA256TRUNCSALT	sha1(cut(sha256(pass), 0, N) . salt) *(N = 21–63; see Note [24])*
e679	SHA1SHA256TRUNCMD5SALT	sha1(cut(sha256(md5(pass)), 0, N) . salt) *(N = 21–63; see Note [24])*
e680	SHA1-HMAC-MD5	sha1(hmac_md5(pass, user))
e681	SHA1SALTSHA1PASSPEPPER	sha1(salt . sha1(pass) . pepper)
e682	SHA1SALTMD5MD5PASS	sha1(salt . md5(md5(pass)))
e683	SHA1MD5TRUNCSALT	sha1(cut(md5(pass), 0, N) . salt) *(N = 21–32; see Note [24])*
e684	SHA1SHA1TRUNC-SHA1PASS-3	sha1(cut(sha1(sha1(pass) . cut(pass, 0, 3)), 0, N)) *(N = 20–40; see Note [24])*
e685	SHA1MD5-SALTMD5PASS	sha1(md5(salt . md5(pass)))
e686	SHA1MD5-2xMD5-MD5	h=md5(md5(pass)); sha1(md5(h . h))
e687	SHA1SHA1MD5MD5PASS1SALT	sha1(sha1(md5(md5(pass))) . salt) *(Note [24])*
e688	SHA1MD5sub1-20MD5	sha1(md5(cut(md5(pass), 0, 20)))
e689	SHA1SHA512UCTRUNC	sha1(cut(upper(sha512(pass)), 0, N)) *(N = 21–128; see Note [24])*
e690	SHA1SALTSHA512UCTRUNC	sha1(salt . cut(upper(sha512(pass)), 0, N)) *(N = 21–128; see Note [24])*
e691	SHA1SALTSHA256UCTRUNC	sha1(salt . cut(upper(sha256(pass)), 0, N)) *(N = 21–64; see Note [24])*
e692	SHA1MD5UC-MD5UCSALT	sha1(upper(md5(upper(md5(pass)) . salt)))
e693	SHA1MD5PASSMD5	sha1(md5(pass . md5(pass)))
e694	MD5DECBASE64	md5(frombase64(pass))
e695	SHA1DECBASE64	sha1(frombase64(pass))
e696	SHA1MD5UCSHA1BASE64	sha1(upper(md5(sha1(base64(pass)))))
e697	SHA1MD5SHA1MD5MD5SHA1MD5	sha1(md5(sha1(md5(md5(sha1(md5(pass)))))))
e698	MD5SHA1MD5MD5SHA1MD5	md5(sha1(md5(md5(sha1(md5(pass))))))
e699	SHA1MD5UCSHA1UCMD5UC	sha1(upper(md5(upper(sha1(upper(md5(pass)))))))
e700	MD5MD5SHA1SALT	md5(md5(sha1(pass)) . salt)
e701	MD5MD5SHA256SALT	md5(md5(sha256(pass)) . salt)
e702	MD5SHA1x	md5(sha1^N(pass)) *(N = iteration count)*
e703	MD5DECBASE64MD5	md5(frombase64(md5(pass)))
e704	SHA1SALTMD5PASSMD5	sha1(salt . md5(pass . md5(pass)))
e705	SHA1MD5sub8-24MD5	sha1(md5(cut(md5(pass), 8, 16)))
e706	SHA1GOST	sha1(gost(pass))
e707	MD4UTF16SHA256x	md4(utf16le(sha256^N(pass))) *(N = iteration count)*
e708	MD4UTF16SHA256SHA256SHA256SHA256SHA256	md4(utf16le(sha256^5(pass)))
e709	SHA1MD5RAWUCMD5RAW	sha1(upper(hex(md5_bin^3(pass))))
e710	SHA1SHA3-256TRUNC	sha1(cut(sha3_256(pass), 0, N)) *(N = 21–64; see Note [24])*
e711	SHA1SHA3-256	sha1(sha3_256(pass))
e712	SHA1MD5SQL5	sha1(md5("*" . upper(sha1(sha1_bin(pass))))) *(see Notes [1], [24])*
e713	SHA1MD5MD5SQL5	sha1(md5(md5("*" . upper(sha1(sha1_bin(pass)))))) *(see Notes [1], [24])*
e714	SHA1RMD128	sha1(rmd128(pass))
e715	SHA1-SHA1SALTSHA1PASS	sha1(sha1(salt) . sha1(pass))
e716	SHA1MD5MD5UC	sha1(md5(upper(md5(pass))))
e717	SHA1MD5MD5UCMD5UC	sha1(md5(upper(md5(upper(md5(pass))))))
e718	SHA11SALTMD5UC	*(complex: iterates single-char salts with UC MD5)*
e719	SHA1SALTMD5MD5PASSPEPPER	sha1(salt . md5(md5(pass)) . pepper)
e720	SHA1SHA1UCPASSSALT	sha1(upper(sha1(pass)) . salt)
e721	SHA1MD5sub1-20MD5MD5	sha1(md5(cut(md5(md5(pass)), 0, 20)))
e722	SHA1MD4UTF16UCMD4UTF16UC	sha1(upper(md4(utf16le(upper(md4(utf16le(pass)))))))
e723	SHA1SHA512TRUNCMD5	sha1(cut(sha512(md5(pass)), 0, N)) *(N = 21–128; see Note [24])*
e724	SHA1MD5UCx	sha1(upper(md5^N(pass))) *(N = iteration count)*
e725	SHA1SALTSHA256TRUNC	sha1(salt . cut(sha256(pass), 0, N)) *(N = 21–64; see Note [24])*
e726	SHA1SALTSHA256TRUNCMD5	sha1(salt . cut(sha256(md5(pass)), 0, N)) *(N = 21–64; see Note [24])*
e727	SHA1UTF16LE	sha1(utf16le(pass))
e728	SHA1UTF16BEZ	sha1(utf16be(pass) . fromhex("00"))

Type	Name	hx Expression
e729	SHA1ZUTF16LE	sha1(fromhex("00") . utf16le(pass))
e730	SHA1UCUTF16LE	sha1_uc(utf16le(pass))
e731	SHA1MD5UC1LC	*uppercase MD5 hex with one alphabetic position lowercased — see Note [8]*
e732	SHA1UTF7	sha1(utf7(pass)) *(Note [9])*
e733	SHA1MD51CAPMD5	emit(sha1(cap(md5^(i+1)(pass), P))) *(as e749, one extra md5)*
e734	SHA1BASE64MD5	sha1(base64(md5(pass)))
e735	SHA1SALTSHA1UCPASS	sha1(salt . upper(sha1(pass)))
e736	SHA1MD5xSALT	sha1(md5^N(pass) . salt) *(N = iteration count)*
e737	SHA1SHA11CAP	sha1(cap(sha1(pass)))
e738	SHA1SALTMD5SHA1PASS	sha1(salt . md5(sha1(pass)))
e739	SHA1SHA1TRUNCSALT	sha1(cut(sha1(pass), 0, N) . salt) *(N = 21–40; see Note [24])*
e740	SHA1-MD5SALT-CR	sha1(md5(md5(pass) . salt) . fromhex("0d"))
e741	SHA1-MD5MD5SALT-CR	sha1(md5(md5(md5(pass)) . salt) . fromhex("0d"))
e742	SHA1SALTMD5SHA1PASSPEPPER	sha1(salt . md5(sha1(pass)) . pepper)
e743	SHA1MD5CAPMD5SALT	sha1(cap(md5(md5(pass))) . salt)
e744	SHA1SALTSHA1CAP	sha1(salt . cap(sha1(pass)))
e745	SHA1SHA384TRUNC	sha1(cut(sha384(pass), 0, N)) *(N = 21–96; see Note [24])*
e746	SHA1RMD160TRUNC	sha1(cut(rmd160(pass), 0, N)) *(N = 21–40; see Note [24])*
e747	SHA1BASE64MD5UC	sha1(base64(upper(md5(pass))))
e748	SHA1MD5CAPSHA1SALT	sha1(cap(md5(sha1(pass))) . salt)
e749	SHA1MD5x1CAP	emit(sha1(cap(md5^i(pass), P))) *(i = 1–q; P = each lowercase position, and each l a–f)*
e750	SHA1SHA1SHA1TRUNC	sha1(cut(sha1(sha1(pass)), 0, N)) *(N = 20–40; see Note [24])*
e751	SHA1SALTSHA1MD5	sha1(salt . sha1(md5(pass)))
e752	SHA1UTF16BE	sha1(utf16be(pass))
e753	SHA1SHA0	sha1(sha0(pass))
e754	SHA1MD4	sha1(md4(pass))
e755	SHA1SHA1TRUNCMD5	sha1(cut(sha1(md5(pass)), 0, N)) *(N = 20–40; see Note [24])*
e756	SHA1MD5MD5UCMD5MD5UC	sha1(md5(upper(md5(md5(upper(md5(pass)))))))
e757	SHA1SALTMD5UCMD5UC	sha1(salt . upper(md5(upper(md5(pass)))))
e758	SHA1MD51CAPMD5MD5	emit(sha1(cap(md5^(i+2)(pass), P))) *(as e749, two extra md5)*
e759	SHA1MD5SHA1PASSSALT	sha1(md5(sha1(pass) . salt))
e760	SHA1SQL5MD5	sha1("*" . upper(sha1(sha1_bin(md5(pass))))) *(see Notes [1], [24])*
e761	SHA1SQL5MD5MD5	sha1(md5("*" . upper(sha1(sha1_bin(md5(pass)))))) *(see Notes [1], [24])*
e762	SHA1revSHA1	sha1(rev(sha1(pass)))
e763	SHA11SALTMD5SHA256	*five-variant single-byte salt search — see Note [10]*
e764	SHA1SHA256MD5MD5	sha1(sha256(md5(md5(pass))))
e765	SHA1MD5SALTPASS	sha1(md5(salt . pass))
e766	SHA224SHA1	sha224(sha1(pass))
e767	MD5DECBASE64MD5BASE64MD5	md5(frombase64(md5(base64(md5(pass)))))
e768	SHA1revBASE64	sha1(rev(base64(pass)))
e769	SHA1revBASE64x	sha1(rev(base64^N(pass))) *(N = iteration count)*
e770	SHA1BASE64CUSTBASE64MD5	*four-variant {MD5} prefix base64 format — see Notes [11], [24]*
e771	SHA1MD5sub1-16	sha1(cut(md5(pass), 0, 16))
e772	SHA1MD5sub1-16MD5	sha1(cut(md5(md5(pass)), 0, 16))
e773	SHA1MD5sub1-16MD5MD5	sha1(cut(md5(md5(md5(pass))), 0, 16))
e774	SHA1SHA1sub1-16	sha1(cut(sha1(pass), 0, 16))
e775	MD4UTF16MD5MD5MD5MD5	md4(utf16le(md5^4(pass)))
e776	MD4UTF16SHA1UC	md4(utf16le(upper(sha1(pass))))
e777	MD4UTF16MD5x	md4(utf16le(md5^N(pass))) *(N = iteration count)*
e778	MD4UTF16SHA1x	md4(utf16le(sha1^N(pass))) *(N = iteration count)*
e779	MD4UTF16SHA256MD5	md4(utf16le(sha256(md5(pass))))

Type	Name	hx Expression
e780	MD4UTF16SHA256SHA1	md4(utf16le(sha256(sha1(pass))))
e781	MD4UTF16-2xMD5	md4(utf16le(md5(pass) . md5(pass)))
e782	MD4UTF16MD5PASSMD5SHA1PASS	md4(utf16le(md5(pass) . md5(sha1(pass))))
e783	MD4UTF16MD5PASSMD5SALT	md4(utf16le(md5(pass) . md5(salt)))
e784	MD4UTF16MD5MD5PASSMD5SALT	md4(utf16le(md5^2(pass) . md5(salt)))
e785	MD4UTF16MD5PASSMD5SHA1SALT	md4(utf16le(md5(pass) . md5(cut(sha1(salt), 0, 32))))
e786	NTLMH	md4(utf16le(pass)) *(Note [29])*
e787	MD4UTF16SQL3	md4(utf16le(mysql3(pass)))
e788	MD4UTF16BASE64	md4(utf16le(base64(pass)))
e789	MD4UTF16revBASE64x	md4(utf16le(rev(base64(pass)))) *(N = iteration count)*
e790	SHA1BASE64SHA256	sha1(base64(sha256(pass)))
e791	MD4UTF16BASE64SHA256	md4(utf16le(base64(sha256(pass))))
e792	HMAC-MD5-KPASS	hmac_md5(salt, pass)
e793	HMAC-SHA1-KPASS	hmac_sha1(salt, pass)
e794	HMAC-SHA224-KPASS	hmac_sha224(salt, pass)
e795	HMAC-SHA256-KPASS	hmac_sha256(salt, pass)
e796	HMAC-SHA384-KPASS	hmac_sha384(salt, pass)
e797	HMAC-SHA512-KPASS	hmac_sha512(salt, pass)
e798	HMAC-RMD160-KPASS	hmac_rmd160(salt, pass)
e799	HMAC-RMD320-KPASS	hmac_rmd320(salt, pass)
e800	MD5UTF16LE	md5(utf16le(pass))
e801	MD5UTF16LEPASSSALT	md5(utf16le(pass) . salt)
e802	MD5UTF16LESALTPASS	md5(salt . utf16le(pass))
e803	SHA1UTF16LEPASSSALT	sha1(utf16le(pass) . salt)
e804	SHA1UTF16LESALTPASS	sha1(salt . utf16le(pass))
e805	SHA256UTF16LE	sha256(utf16le(pass))
e806	SHA256UTF16LEPASSSALT	sha256(utf16le(pass) . salt)
e807	SHA256UTF16LESALTPASS	sha256(salt . utf16le(pass))
e808	SHA512UTF16LE	sha512(utf16le(pass))
e809	SHA512UTF16LEPASSSALT	sha512(utf16le(pass) . salt)
e810	SHA512UTF16LESALTPASS	sha512(salt . utf16le(pass))
e811	SHA384PASSSALT	sha384(pass . salt)
e812	SHA384SALTPASS	sha384(salt . pass)
e813	SHA384UTF16LE	sha384(utf16le(pass))
e814	SHA384UTF16LEPASSSALT	sha384(utf16le(pass) . salt)
e815	SHA384UTF16LESALTPASS	sha384(salt . utf16le(pass))
e816	RMD320	rmd320(pass)
e817	MD5-SHA1SALTPASS	md5(sha1(salt . pass))
e818	MD5-SALTMD5PASS-SALT	md5(salt . md5(pass) . salt)
e819	MD5-MD5MD5PASSSALT-PEP	md5(md5(md5(pass) . salt) . pepper)
e820	MD5-MD5SALT-MD5MD5PASS	md5(md5(salt) . md5(md5(pass)))
e821	MD5-MD5MD5PASSSALT-PEP2	md5(md5(md5(pass . salt)) . pepper)
e822	MD5-SALT-SHA1PEPPASS	md5(salt . sha1(pepper . pass))
e823	SHA1-SALTSHA1PASSSALT	sha1(salt . sha1(pass . salt))
e824	SHA1-SALTSHA1U16	sha1(fromhex(salt) . sha1_bin(utf16le(user) . ":" . utf16le(pass)))
e825	SHA256SALTPASSSALT	sha256(salt . pass . salt)
e826	SHA256-SALTSHA256RAW	sha256(salt . sha256_bin(pass))
e827	WRLSALTPASSSALT	wrl(salt . pass . salt)
e828	HMAC-BLAKE2S	hmac_blake2s(pass, salt)
e829	MURMUR64A	murmur64a(pass)
e830	MURMUR64AZERO	murmur64a(pass) *(zero seed)*
e831	SHA224PASSSALT	sha224(pass . salt)

Type	Name	hx Expression
e832	SHA224SALTPASS	sha224(salt . pass)
e833	SSHA1BASE64	"{SSHA}" . base64(sha1_bin(pass . salt) . salt)
e834	SHA1PASSHEXSALT	sha1(pass . fromhex(salt))
e835	SSHA256BASE64	"{SSHA256}" . base64(sha256_bin(pass . salt) . salt)
e836	SSHA512BASE64	"{SSHA512}" . base64(sha512_bin(pass . salt) . salt)
e837	HMAC-STREEBOG256-KPASS	hmac_streebog256(salt, pass)
e838	HMAC-STREEBOG256	hmac_streebog256(pass, salt)
e839	HMAC-STREEBOG512-KPASS	hmac_streebog512(salt, pass)
e840	HMAC-STREEBOG512	hmac_streebog512(pass, salt)
e841	BLAKE2B512	blake2b512(pass)
e842	BLAKE2B512PASSSALT	blake2b512(pass . salt)
e843	BLAKE2B512SALTPASS	blake2b512(salt . pass)
e844	BLAKE2S256	blake2s256(pass)
e845	BLAKE2B256	blake2b256(pass)
e846	BLAKE2B256PASSSALT	blake2b256(pass . salt)
e847	BLAKE2B256SALTPASS	blake2b256(salt . pass)
e848	DESENCRYPT	des(pass, fromhex(salt))
e849	DES3ENCRYPT	des3(pass, fromhex(salt))
e850	MSSQL2000	sha1(utf16le(pass) . salt) *(MSSQL 2000, case-sensitive half)*
e851	MSSQL2005	sha1(utf16le(pass) . salt) *(MSSQL 2005)*
e852	MSSQL2012	sha512(utf16le(pass) . salt) *(MSSQL 2012+)*
e853	MACOSX	sha1(fromhex(salt) . pass) *(macOS 10.4–10.6; salt is hex-decoded)*
e854	MACOSX7	pbkdf2_sha512(pass, salt, N, 128) *(macOS 10.7+)*
e855	POSTGRESQL	md5(pass . user) *(PostgreSQL)*
e856	JUNIPERSSG	juniper_encode(md5_bin(user . ":Administration Tools:" . pass))
e857	SKYPE	md5(user . fromhex("0a") . "skyper" . fromhex("0a") . pass) *(Skype)*
e858	PEOPLESOFT	base64(sha1_bin(utf16le(pass))) *(PeopleSoft)*
e859	EPISERVER	base64(sha1_bin(salt . pass)) *(EPiServer v0)*
e860	HMAILSERVER	sha256(salt . pass) *(hMailServer)*
e861	CISCOPIX	cisco_pix_encode(md5_bin(pad(pass, 16))) *(Cisco PIX)*
e862	CISCOASA	cisco_pix_encode(md5_bin(pad(pass . salt, 16)))
e863	MEDIAWIKI	md5(salt . "-" . md5(pass)) *(MediaWiki B-type)*
e864	DAHUA	md5(salt . upper(md5(salt2 . pass))) *(Dahua)*
e865	CISCO4	sha256(pass) *(Cisco Type 4)*
e866	CISCOISE	pbkdf2_sha256(pass, salt, 1000, 32) *(Cisco ISE)*
e867	SAMSUNGSHA1	*Note [33]*
e868	AIX-MD5	md5crypt(pass, salt) *(AIX {smd5})*
e869	AIX-SHA1	pbkdf2_sha1(pass, salt, N, 20) *(AIX {ssha1})*
e870	AIX-SHA256	pbkdf2_sha256(pass, salt, N, 32) *(AIX {ssha256})*
e871	AIX-SHA512	pbkdf2_sha512(pass, salt, N, 64) *(AIX {ssha512})*
e872	IPMI2-SHA1	hmac_sha1(pass, rakp_data) *(IPMI 2.0 RAKP)*
e873	IPMI2-MD5	hmac_md5(pass, rakp_data) *(IPMI 2.0 RAKP)*
e874	KRB5PA23	rc4_hmac_md5(pass, realm, user, timestamp) *(see Note [51])*
e875	MYSQL-SHA256CRYPT	sha256crypt(pass, salt) *(MySQL)*
e876	DRUPAL7	phpass(pass, salt, N) *(Drupal 7)*
e877	SYBASE-ASE	sha256(salt . pass) *(Sybase ASE)*
e878	NETSCALER	sha1(salt . pass . fromhex("00")) *(NetScaler; trailing NUL byte)*
e879	NSEC3	sha1^N(pass . salt) *(DNSSEC NSEC3, N = iteration count)*
e880	WBB3	sha1(salt . sha1(salt . sha1(pass))) *(WBB3)*
e881	RACF	"$racf$*" . user . "*" . upper(racf_encrypt(pass, user)) *(IBM RACF)*
e882	DOMINO5	domino5(pass)
e883	DOMINO6	domino6(pass, fromhex(salt))

Type	Name	hx Expression
e884	SCRYPT	scrypt(pass, salt, N, r, p)
e885	JUNIPERIVE	md5(salt . md5(pass)) *(Juniper IVE)*
e886	PHPS	md5(salt . md5(pass)) *(PHPS)*
e887	ARUBAOS	md5(salt . pass) *(ArubaOS — salt has vendor prefix)*
e888	ISCSI-CHAP	*Note [34]*
e889	WERKZEUG-MD5	pbkdf2_md5(pass, salt, N, 32) *(Werkzeug)*
e890	WERKZEUG-SHA256	pbkdf2_sha256(pass, salt, N, 32) *(Werkzeug)*
e891	AUTHME	sha256(sha256(pass) . salt) *(AuthMe)*
e892	NETWITNESS	*(RSA NetWitness authentication derivation)*
e893	NETSCALER-SHA512	sha512(salt . pass . fromhex("00")) *(NetScaler; trailing NUL byte)*
e894	SHA1SHA1SALTPASSSALT	sha1(sha1(salt . pass . salt))
e895	NETSCALER-PBKDF2	pbkdf2_sha256(pass, salt, N, 32) *(NetScaler)*
e896	ORACLE7	upper(oracle7(pass, salt)) *(Oracle 7 / H: Type)*
e897	NETNTLMV1	des_block(md4(utf16le(pass)), challenge) *(NetNTLMv1)*
e898	NETNTLMV2	hmac_md5(md4(utf16le(pass)), challenge) *(NetNTLMv2)*
e899	LASTPASS	pbkdf2_sha256(pass, salt, N, 32) *(LastPass)*
e900	FORTIGATE	sha1(salt . pass . salt) *(FortiGate)*
e901	DOMINO8	*(Lotus Notes 8/9: iterated hash derivation with salt)*
e902	SIPHASH	siphash(pass, fromhex(salt)) *(Note [35])*
e903	CRAMMD5	hmac_md5(pass, challenge) *(CRAM-MD5 RFC 2195)*
e904	SAPCODVNH	sha1(pass . salt) *(SAP CODVN H/iSCHF)*
e905	REDHAT389DS	pbkdf2_sha256(pass, salt, N, 32) *(389-DS)*
e906	POSTGRESCRAM	md5(md5(pass . user) . fromhex(salt)) *(salt = 8-hex challenge)*
e907	MYSQLCRAM	upper(sha1(sha1_bin(pass))) *(MySQL 4.1+ password verifier)*
e908	SHIRO1	pbkdf2_sha1(pass, salt, N, 16) *(Apache Shiro)*
e909	ECRYPTFS	sha512^65536(pass . salt) *(Linux eCryptfs)*
e910	ORACLE12	pbkdf2_sha512(pass, salt, 4096, 64) *(Oracle 12c)*
e911	COLDFUSION10	sha256(salt . upper(sha1(pass))) *(Adobe ColdFusion 10)*
e912	AZURESYNC	pbkdf2_sha256(pass, salt, 1000, 32) *(Azure AD Sync; see Note [51])*
e913	ANDROIDFDE	pbkdf2_sha1(pass, salt, 2000, 16) *(Android FDE master key)*
e914	KRB5TGS23	rc4_hmac_md5(pass, realm, spn) *(Kerberos 5 TGS etype 23; see Note [51])*
e915	AXCRYPT	axcrypt(pass, fromhex(salt), iter) *(AES Key Wrap)*
e916	AXCRYPTSHA1	cut(sha1(pass), 0, 32)
e917	CISCO9	scrypt(pass, salt, 16384, 1, 1) *(Cisco Type 9)*
e918	DCC2	pbkdf2_sha1(md4(utf16le(pass)), utf16le(username), 10240, 16) *(see Note [51])*
e919	PWSAFE3	sha256^N(pass . salt) *(Password Safe v3, N = iteration count)*
e920	IKEPSK-MD5	hmac_md5(psk, ike_payload) *(IKE PSK authentication, MD5)*
e921	IKEPSK-SHA1	hmac_sha1(psk, ike_payload) *(IKE PSK authentication, SHA1)*
e922	SAP-BCODE	upper(sap_bcode(pass, salt))
e923	SAP-BCODE4	cut(upper(sap_bcode(pass, salt)), 0, 8) . "00000000"
e924	SAP-PASSCODE	*Note [36]*
e925	SAP-PASSCODE5	*Note [36]*
e926	AS400-DES	upper(as400_des(pass, salt)) *(salt = userid)*
e927	PS-TOKEN	sha1(fromhex(salt) . utf16le(pass)) *(PeopleSoft PS_TOKEN)*
e928	WINPHONE	sha256(utf16le(pass) . fromhex(salt)) *(salt = 256-hex / 128 bytes)*
e929	RACF-KDFAES	*Note [18]*
e930	TACACS	*Note [19]*
e931	APPLE-SECURE-NOTES	pbkdf2_sha256(pass, salt, 20000, 32) *(Apple Secure Notes)*
e932	CRAMMD5-DOVECOT	hmac_md5(pass, challenge) *(Dovecot CRAM-MD5)*
e933	JWT	hmac_sha256(pass, header . "." . payload) *(JWT HS256 signature)*
e934	QNX-MD5	md5crypt(pass, salt) *(QNX)*
e935	QNX-SHA256	sha256crypt(pass, salt, rounds) *(QNX)*

Type	Name	hx Expression
e936	QNX-SHA512	sha512crypt(pass, salt, rounds) *(QNX)*
e937	QNX7-SHA512	sha512crypt(pass, salt, rounds) *(QNX 7)*
e938	SHA1-S1PS2	sha1(s1 . pass . s2) *(dual-salted SHA1)*
e939	RAILS-RESTFUL	sha1^10(site_key . salt . pass) *(Rails restful_authentication)*
e940	KRB5PA-17	aes128_cts_hmac_sha1(pass, realm, user) *(Kerberos 5 PA etype 17)*
e941	KRB5PA-18	aes256_cts_hmac_sha1(pass, realm, user) *(Kerberos 5 PA etype 18)*
e942	WPA-PMKID	*WPA PMKID; see Note [12]*
e943	WPA-EAPOL	*WPA 4-way handshake MIC; see Note [13]*
e944	ANSIBLE-VAULT	*Note [20]*
e945	APFS	*Note [14]*
e946	OTM-SHA256	*Note [21]*
e947	TELEGRAM-SHA256	pbkdf2_sha512(pass, salt, 100000, 56) *(Telegram local passcode)*
e948	WEB2PY-SHA512	pbkdf2_sha512(pass, salt, 1000, 64) *(web2py)*
e949	SOLARWINDS	*Note [22]*
e950	SOLARWINDS2	*Note [23]*
e951	SIMPLACMS	md5(salt . pass . md5(pass)) *(Simpla CMS)*
e952	APPLE-KEYCHAIN	pbkdf2_sha1(pass, salt, 1000, 24) *(macOS login keychain unlock)*
e953	APPLE-IWORK	pbkdf2_sha1(pass, salt, 50000, 16) *(Apple iWork)*
e954	BITWARDEN	pbkdf2_sha256(pass, salt, N, 32) *(Bitwarden)*
e955	MONGODB-SHA1	pbkdf2_sha1(pass, salt, N, 20) *(MongoDB SCRAM-SHA-1 stored key)*
e956	MONGODB-SHA256	pbkdf2_sha256(pass, salt, N, 32) *(MongoDB SCRAM-SHA-256 stored key)*
e957	FORTIGATE256	sha256(salt . pass . salt) *(FortiGate256)*
e958	UMBRACO	hmac_sha1(pass, salt) *(Umbraco)*
e959	DAHUA-AUTH	dahua_encode(md5_bin(pass)) *(Dahua Auth)*
e960	BESDER-AUTH	besder_encode(md5_bin(pass))
e961	SQLCIPHER	pbkdf2_sha1(pass, salt, N, 32) *(SQLCipher)*
e962	RORAILS-SHA1	sha1("--" . salt . "--" . pass . "--") *(Rails Restful Auth)*
e963	AES128-NOKDF	aes_ecb_encrypt(pad(pass, 16), fromhex(salt))
e964	AES192-NOKDF	aes_ecb_encrypt(pad(pass, 24), fromhex(salt))
e965	AES256-NOKDF	aes_ecb_encrypt(pad(pass, 32), fromhex(salt))
e966	VMWARE-VMX	pbkdf2_sha1(pass, salt, N, 32) *(VMware VMX encryption)*
e967	BCRYPTSHA512	bcrypt(sha512(pass), salt, cost)
e968	POSTGRESSCRAM256	pbkdf2_sha256(pass, salt, 4096, 32) *(PostgreSQL SCRAM-SHA-256)*
e969	AWSSIGV4	*AWS SigV4 4-step signing key — see Note [3]*
e970	KRB5DB17	aes128_cts_hmac_sha1(pass, realm, user) *(Kerberos 5 KDB etype 17)*
e971	KRB5DB18	aes256_cts_hmac_sha1(pass, realm, user) *(Kerberos 5 KDB etype 18)*
e972	MURMUR3	murmur3(pass, seed) *(MurmurHash3_x86_32; seed stored beside the digest)*
e973	VEEAM-VBK	*Note [15]*
e974	MSSNTP	*(MS-SNTP: MD5 of NTP packet with md4(utf16le(pass)) as key)*
e975	SSPR-MD5	md5^N(pass) *(SSPR; N from the $sspr$ format, binary chaining)*
e976	SSPR-SHA1	sha1^N(pass) *(SSPR; N from the $sspr$ format, binary chaining)*
e977	SSPR-SHA1S	sha1(pass . salt) *(SSPR salted)*
e978	SSPR-SHA256	sha256^N(salt . pass) *(Note [37])*
e979	SSPR-SHA512	sha512^N(salt . pass) *(Note [37])*
e980	EMPIRECMS	md5(md5(salt) . md5(pass)) *(EmpireCMS)*
e981	PBKDF1-SHA1	pbkdf1_sha1(pass, salt, iter)
e982	MSONLINE	pbkdf2_sha256(pass, salt, N, 32) *(Microsoft Online; see Note [51])*
e983	WBB4	sha1(salt . sha1(salt . sha1(pass))) *(WBB4 = WBB3)*
e984	SAPCODVNH512	pbkdf2_sha512(pass, salt, N, 64) *(SAP CODVN H SHA512)*
e985	SM3CRYPT	sm3crypt(pass, salt, rounds)
e986	AS400SSHA1	*Note [38]*
e987	ARGON2	argon2(pass, salt, m, t, p)

Type	Name	hx Expression
e988	SM3	sm3(pass)
e989	BCRYPTHMACSHA256	bcrypt(base64(hmac_sha256_bin(salt, pass)), salt, N)
e990	WPA-PMK	*candidate is the 32-byte PMK as 64 hex — no PBKDF2*
e991	MD5SALT1SALT2	md5(salt . pass . salt2) *(also John dynamic_1560, SocialEngine)*
e992	SYMFONY256	*Note [39]*
e993	WPBCRYPT	bcrypt(base64(hmac_sha384_bin(pass, "wp-sha384")), salt, N) *(WordPress)*
e994	GOST12512CRYPT	gost12_512crypt(pass, salt, rounds)
e995	YESCRYPT	yescrypt(pass, salt, N, r, p)
e996	MD5SHA256SHA256	md5(sha256(sha256(pass)))
e997	BSDICRYPT	bsdicrypt(pass, salt)
e998	GOST-YESCRYPT	gost_yescrypt(pass, salt)
e999	SHA1CRYPT	sha1crypt(pass, salt, rounds)
e1000	7ZIP	sevenzip(pass, salt)
e1001	CMIYC	cmiyc(pass, salt)
e1002	RMD256	rmd256(pass)
e1003	MD4SALTPASS	md4(salt . pass)
e1004	MD4PASSSALT	md4(pass . salt)
e1005	SHA1UCUSERPASS	sha1(upper(user) . ":" . pass) *(John dynamic_35)*
e1006	SHA1USERCOLONPASS	sha1(user . ":" . pass) *(John dynamic_36)*
e1007	MD5PASSSALTMD5PASSSALT	md5(pass . salt . md5(pass . salt))
e1008	QAS-VASAUTH	sha256("#" . salt . "-" . pass) *(Quest QAS vas_auth)*
e1009	POSTOFFICE	md5(salt . "Y" . pass . fromhex("f7") . salt) *(Post.Office)*
e1010	IPB2	md5(md5(fromhex(salt)) . md5(pass)) *(Invision Power Board 2)*
e1011	MD5USERMD5PASSSALT	md5(user . md5(pass) . salt) *(John dynamic_15)*
e1012	RVARY	*Note [32]*
e1013	SHA512RAWPASSSALT	sha512_bin(pass . salt) *(BlackBerry ES10 at -i 100)*
e1014	EPISERVER-SID	sha1(cut(fromhex(salt), 0, 29) . pass . fromhex("00")) *(EPiServer tblSID)*
e1015	ORACLE11	upper(sha1(pass . fromhex(salt))) *(Oracle 11g, stored form — see Note [40])*
e1016	SAPCODVNH256	*SAP CODVN H SHA-256 — see Note [41]*
e1017	SAPCODVNH384	*SAP CODVN H SHA-384 — see Note [41]*
e1018	MONGODB	md5(user . ":mongo:" . pass) *(MongoDB system credential)*
e1019	H3C	*H3C/HPE/Huawei — see Note [42]*
e1020	ARGON2MD5	argon2(md5(pass), salt, m, t, p) *(vBulletin 6 — see Note [43])*
e1021	DRAGONFLY3-32	sha256(pass . "3" . fromhex("00") . salt) *(DragonFly BSD — see Note [44])*
e1022	DRAGONFLY3-64	sha256(pass . "3" . fromhex("00") . "sha5" . salt) *(DragonFly BSD — see Note [44*
e1023	DRAGONFLY4-32	sha512(pass . "4" . fromhex("00") . salt) *(DragonFly BSD — see Note [44])*
e1024	DRAGONFLY4-64	sha512(pass . "4" . fromhex("00") . "/etc" . salt) *(DragonFly BSD — see Note [44]*
e1025	GOST12256CRYPT	gost12_256crypt(pass, salt, rounds) *(see Note [45])*
e1026	GOST94CRYPT	gost94crypt(pass, salt, rounds) *(see Note [45])*
e1027	SUNMD5	sunmd5(pass, salt, rounds) *(Solaris — see Note [46])*

13.1 Notes

13.1.1 Note [1]

e206 SHA1SQL5 — multi-emit construction that produces three candidates per password, capturing variants of the MySQL SQL5 hash format including the leading * prefix and a leading-space variant:

```
s = "*" . upper(sha1(sha1_bin(pass)))
emit(sha1(s))
emit(sha1("*" . sha1(sha1_bin(pass))))
emit(sha1(" " . s))
```

13.1.2 Note [2]

e435 MD5SPECAM — a parsing wrapper, not a distinct hash primitive. The input record is colon-separated of the form

```
username:email@domain[:password]
```

The wrapper lower-cases each field and extracts:

```
user  = lower( field 0 of input )
email = lower( field 1 of input )         # the "name@host" part
cand  = lower( field 2 of input ) if present
cand  = user                              # otherwise
```

It then computes **both** of the following, emitting both hashes for the parsed candidate:

```
emit(md5(user . "::" . cand))                    # = e432 MD5AM
emit(md5(user . "::" . cand . email . ":73@^bhhs&#@&^@8@*$"))    # = e433 MD5AM2
```

The **salt** position in **MD5AM** is the parsed user; the **salt2** position in **MD5AM2** is the parsed email. What distinguishes **MD5SPECAM** from a direct invocation of **MD5AM** or **MD5AM2** is the parsing of the input record — the hashing itself reuses the e432 and e433 expressions.

13.1.3 Note [3]

e969 AWSSIGV4 — AWS Signature Version 4 signing-key derivation, a four-step HMAC-SHA256 chain that mixes the password with the request date, region, and service name to produce a key that is then used to sign HTTP requests:

```
k1 = hmac_sha256("AWS4" . pass, date)
k2 = hmac_sha256(k1, region)
k3 = hmac_sha256(k2, service)
sk = hmac_sha256(k3, "aws4_request")
```

The final signing key **sk** is what is verified.

13.1.4 Note [4]

e445 SHA1-SALT-SPECIAL — a 10-round iterated SHA-1 over a dash-delimited frame. The initial input wraps the salt in two dashes on each side and pads the password with two dashes on each side; each subsequent round inserts the previous SHA-1 hex digest into the same frame, separated from the salt and the password by two dashes.

```
h = sha1("--".salt."----".pass."----")
for i = 1 to 9 {
    h = sha1("--".salt."--".h."--".pass."----")
}
emit(h)
```

The hashing primitive is plain SHA-1; what is unusual is the buffer framing and the per-round substitution of the previous hex digest between the two salt-dashes and the two password-dashes.

13.1.5 Note [5]

e455 PHPBB3 — the standard phpass construction used by phpBB3, WordPress, and Drupal 7. The MD5 chain operates on raw binary digests (not hex), hence the use of **md5_bin**. The final 16-byte digest is encoded with the phpass itoa64 alphabet via **phpass_encode**.

```
h = md5_bin(salt . pass)
for i = 1 to 2047 { h = md5_bin(h . pass) }
"$H$9" . salt . phpass_encode(h)
```

The **H9** prefix encodes the variant (H) and the iteration-count exponent (9 ⇒ 2^11 = 2048 rounds). The salt is appended literally; the phpass-encoded digest follows.

13.1.6 Note [6]

e544 CRYPTEXT — SHA-1 of the password, with the resulting 20 bytes byte-swapped within each 32-bit group, the literal string "Cryptext" appended, then SHA-1 again and a final per-group byte-swap. There is no special hashing primitive at work — the construction is fully expressible with **sha1_bin** and **cut**:

```
b = sha1_bin(pass)
sw = ""
for j = 0 to 4 {
    sw = sw . cut(b, j*4+3, 1) . cut(b, j*4+2, 1)
            . cut(b, j*4+1, 1) . cut(b, j*4,   1)
}
b2 = sha1_bin(sw . "Cryptext")
out = ""
for j = 0 to 4 {
    out = out . cut(b2, j*4+3, 1) . cut(b2, j*4+2, 1)
              . cut(b2, j*4+1, 1) . cut(b2, j*4,   1)
}
emit(hex(out))
```

The per-group byte-swap reverses byte order within each 4-byte word; equivalent to applying **bswap32()** to the entire SHA-1 digest:

```
hex(bswap32(sha1_bin(pass)))
```

13.1.7 Note [7]

e649 SHA1MD5BASE641SALT — a single-byte salt search. The base value is computed once per password and reused for every candidate salt byte:

```
base = md5(base64(pass))              # 32-char hex string
```

mdxfind then iterates a single salt byte **y** through the range **9 .. 127** (119 candidate values) and computes

```
sha1(base . single_byte(y))           # 33-byte input total
```

emitting each result and checking it against the target. A match is reported with the matching salt rendered as **0xYY** (for example **0x42**).

Reconstructing a found hash. Given an mdxfind result line such as

```
SHA1MD5BASE641SALTx01 13204d41af878472790a713b8f95f581b0700bcf:0x41:password
```

the verifying hx expression is

```
sha1(md5(base64(pass)) . fromhex(salt))
```

invoked with **−p** *password* and **−s 41** (the two hex digits of the reported salt byte; **0x41** is the ASCII letter **A**):

```
$ hashpipe -X 'sha1(md5(base64(pass)) . fromhex(salt))' -p password -s 41
13204d41af878472790a713b8f95f581b0700bcf
```

The reproduced digest matches the **SHA1MD5BASE641SALTx01** field from the original result, confirming the

find.

Full per-byte loop in hx. The complete search can be written by precomputing the 119 single-byte salts as a binary literal and indexing into it:

```
bytes = fromhex("090a0b0c0d0e0f10111213141516171819" .
                "1a1b1c1d1e1f202122232425262728292a2b2c2d2e2f" .
                "303132333435363738393a3b3c3d3e3f40414243" .
                "4445464748494a4b4c4d4e4f505152535455565758 59" .
                "5a5b5c5d5e5f606162636465666768696a6b6c6d6e6f" .
                "707172737475767778797a7b7c7d7e7f")
base = md5(base64(pass))
for i = 0 to 118 { emit(sha1(base . cut(bytes, i, 1))) }
```

With **–i**N the per-salt SHA-1 chain extends to **N** rounds, hex-encoding and re-hashing between rounds.

13.1.8 Note [8]

e731 SHA1MD5UC1LC — SHA-1 of the uppercase MD5 hex string, with exactly one of the alphabetic positions (**A**ne**F**) lowercased. The 32-character MD5 hex contains some number of alphabetic characters (*N*), and the type emits *N* candidate hashes per password — one for each A–F position taken in turn. Hex digits 0–9 are unaffected by case and are skipped. The mdxfind output line carries the matching SHA-1 digest and the password but **does**not record which position was lowercased; reconstruction therefore re-runs the full per-position search and identifies the matching variant by matching SHA-1 against the target.

The hx expression demonstrates the language's **for**, **if**, and string-slice features:

```
b = upper(md5(pass));
for y = 0 to 31 {
    c = cut(b, y, 1);
    if lower(c) != c {
        v = cut(b, 0, y) . lower(c) .
            cut(b, add(y, 1), sub(31, y));
        emit(sha1(v))
    }
}
```

Worked example. For the password **password**, `upper(md5(pass))` is
`5F4DCC3B5AA765D61D8327DEB882CF99`
which contains 14 alphabetic positions (positions 1, 3, 4, 5, 7, 9, 10, 14, 17, 22, 23, 24, 28, 29). Running the expression above against **–p password** emits these 14 SHA-1 digests, in order:

```
8230320f88f5f6e5746f7511cc1c3abd5ce437ed
0867ad7acd326a09c9e3f99c88c1f524dc4cef8b
d9c9670782f1d9781d991b6b1a7e4f78e5062571
8d75ebd9d8795964754b6e0a5421463c5ab2a17d
e724c34add8b6d76072ea683b3b0f771851882e6
35285b8778ec8ee97db5b9587bcec5d6e780a809
ab8e1e6b77f59957c6b9db40cb5fe6adc00233bd
b4962ea3571a518cea8e589629b8406e2d25b723
83ad12dea7e7a0965e564f4b6f1984be4afe75ff
cbc46bedfedd79aade4babdac8bd8c9fff79de63
10880623525198324bc9313c6991fad10ceac218
78eab8a2d1ad15f28791c13b960a4959c3a0971b
55964fa55355f138d027ba96438f0b725a795f8b
6571f71314f2303cd6ae3ceb71599322a31b0116
```

A given mdxfind find of one of these digests with the password **password** is verified by checking the digest against this list. The **add** and **sub** helpers are required because hx expressions are not arithmetic; they take an integer iteration variable and a constant offset and return an integer.

13.1.9 Note [9]

e732 SHA1UTF7 — SHA-1 of the UTF-7 encoding of the password. The hx primitive **utf7()** performs the UTF-7 conversion directly via iconv.

mdxfind quirk. The mdxfind compute path appends the literal character **X** to the password before iconv-converting to UTF-7, then drops the trailing converted byte from the result. This was added to work around an iconv bug where a UTF-7 escape sequence at the end of the input would not be flushed without a following character. mdxfind also short-circuits the SHA-1 step when the UTF-7 output is the same as the input (i.e., when the password contains no characters that require UTF-7 escaping) on the grounds that the result would equal a plain SHA-1 hash and not contribute new information. **Neither workaround is reproduced in hx**: **utf7()** in hx works correctly for trailing-escape inputs and is used unconditionally regardless of whether the conversion changes the input.

Worked example. For the password **pass@word**, mdxfind converts to **pass+AEA-word**, SHA-1s the converted bytes, and reports

```
SHA1UTF7x01 72987265467cb8001facc60fd620442e4c5994fe:pass@word
```

The hx expression reproduces this directly:

```
$ hashpipe -X 'sha1(utf7(pass))' -p 'pass@word'
72987265467cb8001facc60fd620442e4c5994fe
```

A password with no UTF-7-escaping characters (e.g. plain ASCII alphanumeric) will produce a digest equal to **sha1(pass)**; the hx expression returns this naturally, while mdxfind suppresses the result.

13.1.10 Note [10]

e763 SHA11SALTMD5SHA256 — a five-variant single-byte salt search. The base value is **md5(sha256(pass))** (MD5 of the SHA-256 hex of the password) and the salt is a single byte *y* in the range **9 .. 127** (skipping **10**, which mdxfind substitutes with the space character **0x20**). For each *y*, mdxfind computes **five** SHA-1 variants over different placements of *y* relative to the base value, emitting all five and reporting matches with the salt rendered as the single character *y*.

The five variants in hx form, where **salt** is the single salt byte, are:

```
sha1(salt . md5(sha256(pass)))                # variant 1
sha1(md5(sha256(pass)) . salt)                # variant 2
sha1(salt . md5(sha256(pass)) . salt)         # variant 3
sha1(salt . salt . md5(sha256(pass)))         # variant 4
sha1(md5(sha256(pass)) . salt . salt)         # variant 5
```

Worked example. For **pass = "password"**, **salt = "A"** (byte 0x41), the five hx invocations produce:

```
$ hashpipe -X 'sha1(salt . md5(sha256(pass)))' -p password -s A
a391f6e587080f1dd25121ce41f1d8e03c9585f7
$ hashpipe -X 'sha1(md5(sha256(pass)) . salt)' -p password -s A
2ff7a3b90fa7777349049f0576aa7edbeeea8283
$ hashpipe -X 'sha1(salt.md5(sha256(pass)).salt)' -p password -s A
a0d12e54500a36a09749e653aee20ff457986054
$ hashpipe -X 'sha1(salt . salt . md5(sha256(pass)))' -p password -s A
d9cbf46c2950da6f8339a314bbf830810e43af90
$ hashpipe -X 'sha1(md5(sha256(pass)) . salt . salt)' -p password -s A
d3ab8996254c8552ef9f82a8bd4486a603425396
```

matching the five mdxfind output lines for that salt byte.

Reconstructing a found hash. mdxfind reports the salt as the single byte *y*, but does **not** record which of the five placements produced the match. To verify a found hash, compute all five variants for the reported salt and select the one that matches the digest in the **SHA11SALTMD5SHA256** field.

The full per-byte search loop in hx requires a **for** loop over the 119 candidate salts (with the **y==10** substitution to space) and a five-fold **emit** inside. mdxfind's **–i**N mode extends each per-variant SHA-1 chain to **N** rounds, hex-

encoding and re-hashing between rounds.

13.1.11 Note [11]

e770 SHA1BASE64CUSTBASE64MD5 — despite the **CUST** in the name, the base64 encoding uses the standard RFC 4648 alphabet (**A–Z, a–z, 0–9, +**, **/**, padding with **=**). The four variants differ in two orthogonal ways: outer-format choice (binary base64 with newline vs hex string) and trailing-character handling (full-length vs trimmed).

The base value is built from the binary MD5 of the password, prefixed with the literal string **{MD5}** (commonly seen in LDAP and OpenLDAP password fields):

```
inner_b64 = base64(md5_bin(pass))           # e.g. X03MO1qn...mQ==
hex_form  = md5(pass)                       # e.g. 5f4dcc3b...cf99
```

The four mdxfind variants in hx form are:

```
# Variant 1: outer base64 of {MD5}<inner>\n, full length
sha1(base64("{MD5}" . inner_b64 . fromhex("0a")))

# Variant 2: same as 1, with last char stripped
b = base64("{MD5}" . inner_b64 . fromhex("0a"))
sha1(cut(b, 0, sub(length(b), 1)))

# Variant 3: outer base64 of {MD5}<hex>, full length
sha1(base64("{MD5}" . hex_form))

# Variant 4: same as 3, with the two = padding chars stripped
b = base64("{MD5}" . hex_form)
sha1(cut(b, 0, sub(length(b), 2)))
```

Worked example — for **pass = "password"**:

```
$ hashpipe -X 'sha1(base64("{MD5}".base64(md5_bin(pass)).fromhex("0a")))' -p password
17f4f61432f73b94afd0c81738cd2300368dbb00
$ hashpipe -X 'b = base64("{MD5}".base64(md5_bin(pass)).fromhex("0a")); sha1(cut(b, 0, sub
034f2e510f9fd0968cc5ced57a751522b622c129
$ hashpipe -X 'sha1(base64("{MD5}".md5(pass)))' -p password
260c19672dc393c96e1cda85290c0fafc87f7b53
$ hashpipe -X 'b = base64("{MD5}".md5(pass)); sha1(cut(b, 0, sub(length(b), 2)))' -p passwo
6773f3c257e06c1784e949ce16bb8deeeeec3c2d
```

matching the four mdxfind output lines.

hx string-literal note. hx string literals do not interpret backslash escapes — **"0(dq** in an hx expression is the two-character sequence backslash, n, not a newline byte. For the literal LF (0x0a) used in variants 1 and 2, use **fromhex("0a")** to get a single-byte string. The trailing-character truncation in variants 2 and 4 uses **cut(s, 0, sub(length(s), N))** to drop the last *N* characters of the string — a general pattern that does not require any new mapping function.

13.1.12 Note [12]

e942 WPA-PMKID — the WPA/WPA2 PMKID derivation, used by the PMKID attack on WPA-PSK. The PMK is the standard PSK-mode pairwise master key, and the PMKID is the first 16 bytes of HMAC-SHA1 over a fixed string and the two MAC addresses.

```
PMK   = PBKDF2-HMAC-SHA1(pass, essid, 4096, 32)
PMKID = HMAC-SHA1(PMK, "PMK Name" || ap_mac || sta_mac)[0..15]
```

This is fully expressible in hx with the existing primitives:

```
cut(hmac_sha1(pbkdf2_sha1_bin(pass, fromhex(essid), 4096, 32),
              "PMK Name" . fromhex(ap_mac) . fromhex(sta_mac)),
    0, 32)
```

where **essid**, **ap_mac**, **sta_mac** are passed as hex strings. mdxfind reports the find as **WPA*01*** *pmkid*ap_mac*sta_mac*essid**.

13.1.13 Note [13]

e943 WPA-EAPOL — the WPA/WPA2 4-way-handshake MIC. Like PMKID, the PMK is the standard PSK-mode key. The PTK derivation builds a 100-byte *PKE* buffer from the literal label **Pairwise key expansion**, the lexicographically-ordered MACs and nonces, and an iterative PRF expansion based on HMAC-SHA1. The first 16 (WPA1, KCK) or 16 (WPA2, KCK) bytes of the PRF output are the *KCK*, which is then used as the key in either HMAC-MD5 (key version 1, WPA-TKIP) or HMAC-SHA1 (key version 2, WPA-CCMP) over the EAPOL frame to produce the MIC.

```
PMK = PBKDF2-HMAC-SHA1(pass, essid, 4096, 32)

PKE = "Pairwise key expansion"     /* 23 bytes */
    || min(ap_mac, sta_mac)        /*  6 bytes */
    || max(ap_mac, sta_mac)        /*  6 bytes */
    || min(ap_nonce, sta_nonce)    /* 32 bytes */
    || max(ap_nonce, sta_nonce)    /* 32 bytes */
    || 0x00                        /*  1 byte counter */
                                   /* 100 bytes total */

/* IEEE 802.11i PRF-512 expands KCK from PMK */
KCK = HMAC-SHA1(PMK, PKE)[0..15]

if keyver == 1:         /* WPA1 / TKIP */
    MIC = HMAC-MD5(KCK, eapol)[0..15]
elif keyver == 2:       /* WPA2 / CCMP-AES */
    MIC = HMAC-SHA1(KCK, eapol)[0..15]
```

The **eapol** buffer is the 802.1X EAPOL-Key frame from message-2 of the 4-way handshake with the MIC field zeroed out. hashcat presents the data in mode-22000 format (**WPA*** *keyver*MIC*mac_ap*mac_sta*essid*anonce*eapol***).

The conditional MIC primitive can be expressed in hx with an **if** on the integer **keyver** variable, but PKE-frame construction requires the lexicographic **min/max** ordering of MAC pairs and nonces, which is straightforward with the existing **< >** relops once the inputs are bound. For verification of mdxfind output, working byte-for-byte against the hashcat 22000 reference data is recommended.

13.1.14 Note [14]

e945 APFS — an Apple File System volume key check. The on-disk format (**$fvde$2$16$**salt$iter$wkey) encodes a 16-byte salt, an iteration count, and a 40-byte AES-wrapped 32-byte volume key. A password is correct iff the unwrap succeeds, which is detected by the standard RFC 3394 IV check (**0xa6a6a6a6a6a6a6a6**).

```
DK  = PBKDF2-HMAC-SHA256(pass, salt, iter, 32)  /* 32-byte AES-256 KEK */
VOL = AES-Key-Unwrap(DK, wkey)                  /* 32 bytes on success,
                                                   empty on IV mismatch */
```

The hx primitive **aes_unwrap(kek,**wrapped) implements RFC 3394 directly: it accepts 16-byte (AES-128) or 32-byte (AES-256) KEKs and any *8+8***n** wrapped-input length, returning the *8n*-byte plaintext on a successful IV match or the empty string on failure. For verification it is sufficient to test **length(aes_unwrap(\..., wkey)) > 0**; the unwrapped volume key itself is not the published hash and is not normally compared. See **mdxfind.c** JOB_APFS for the reference implementation.

13.1.15 Note [15]

e973 VEEAM-VBK — a Veeam backup-archive password check. The on-disk format is **vbk***salt128hex*iter*ct32hex**.** The password is encoded as UTF-16LE, run through PBKDF2-HMAC-SHA1 to produce a 48-byte derived key, and the AES-CBC ciphertext is decrypted with the first 32 bytes as the AES-256 key and the next 16 bytes as the IV. A correct password leaves the decrypted block in the form *plaintext4*‖*PKCS7***12**, i.e. the last twelve bytes are all **0x0c**.

```
DK  = PBKDF2-HMAC-SHA1(UTF-16LE(pass), salt, iter, 48)
key = DK[0..31]                  /* AES-256 key */
iv  = DK[32..47]                 /* CBC IV       */
PT  = AES-CBC-Decrypt(key, iv, ct)            /* 16 bytes */
ok  = (PT[4..15] == 0x0c × 12)
```

The hx primitive **aes_cbc_decrypt(key,**iv,**ct)** returns the raw decrypted bytes (no padding strip); use the **_bin** suffix when chaining. A canonical hx verification is

```
dk = pbkdf2_sha1_bin(utf16le(pass), fromhex(salt), iter, 48);
pt = aes_cbc_decrypt_bin(cut(dk,0,32), cut(dk,32,16), fromhex(ct));
if cut(pt,4,12) == fromhex("0c0c0c0c0c0c0c0c0c0c0c0c") { emit("ok") }
```

which prints **ok** when the password is correct and nothing otherwise. See **mdxfind.c** JOB_VEEAM_VBK for the reference implementation.

13.1.16 Note [16]

e521 SHA1SALTCX — iterated SHA-1 with double-dash delimiters. Each round emits a separate hash:

```
h = sha1("--" . salt . "----" . pass . "----")
emit(h)
for i = 1 to N-1 {
    h = sha1("--" . salt . "--" . h . "--" . pass . "----")
    emit(h)
}
```

where N is the iteration count from the hash format.

13.1.17 Note [17]

e537 PHPBB3MD5 — iterated MD5 with phpass-style encoding:

```
h = md5_bin(salt . md5(pass))
for i = 1 to 2047 {
    h = md5_bin(h . md5(pass))
}
result = phpass_encode(h)
```

13.1.18 Note [18]

e929 RACF-KDFAES — IBM RACF KDFAES password hash.

```
racf_kdfaes(pass, user, fromhex(salt2), fromhex(salt))
```

where **salt** is 16 bytes (32 hex) and **salt2** is the 16-byte header block.

13.1.19 Note [19]

e930 TACACS — TACACS+ authentication response:

```
hex(cut(xor(md5_bin(fromhex(session) . pass
    . fromhex(seq)), fromhex("010000000000")), 0, 6))
```

Generates a 6-byte ciphertext from session ID, password, and sequence number.

13.1.20 Note [20]

e944 ANSIBLE-VAULT — Ansible Vault key derivation:

```
pbkdf2_sha256(pass, salt, 10000, 80)
```

The 80-byte derived key is split: 32 bytes AES key + 32 bytes HMAC key + 16 bytes IV.

13.1.21 Note [21]

e946 OTM-SHA256 — Oracle Transportation Management iterated SHA-256:

```
h = sha256_bin(salt . pass)
for i = 1 to iter-1 { h = sha256_bin(h) }
result = base64(h)
```

Iteration count and salt are taken from the **otm_sha256:iter:salt:hash** format.

13.1.22 Note [22]

e949 SOLARWINDS — SolarWinds Orion v0:

```
base64(sha512_bin(pbkdf2_sha1_bin(pass,
    cut(lower(salt) . "1244352345234", 0, 8),
    1000, 1024)))
```

The salt is the username, lowercased and right-padded to 8 bytes with the literal string "1244352345234".

13.1.23 Note [23]

e950 SOLARWINDS2 — SolarWinds Orion v1:

```
base64(sha512_bin(pbkdf2_sha1_bin(pass,
    frombase64(salt), 1000, 1024)))
```

The salt is 16 random bytes, base64-encoded in the on-disk **$solarwinds$1$salt$hash** format.

13.1.24 Note [24]

Multi-emit families. Several mdxfind hash types emit **more**than**one**digest per input password rather than the single canonical expression shown in their type row. The Appendix A expression captures the primary form; this note enumerates the additional digests mdxfind also computes and checks against the target table for each input.

Colon-variant emit (PASS suffix). The types **e123** (MD5MD5PASS), **e201** (MD5PASSMD5), **e202** (MD5-DBL-PASS), and **e663** (SHA1MD5MD5PASS) each emit **both**

```
H(hex32(md5(pass)) . pass)
H(hex32(md5(pass)) . ":" . pass)
```

where **H** is the outer hash. The colon variant captures real-world hashes where the implementation separates the inner digest from the password with a literal **:**. The e202 and e663 rows already use **emit()** syntax in Appendix A; e123 and e201 do not, and the multi-emit semantics there are implicit.

Colon-variant emit (USER suffix). The types **e286** (MD5MD5USER), **e308** (SHA1MD5USER), **e354** (MD5CAPMD5USER), and **e355** (MD5CAPMD5MD5USER) similarly emit **both H(hex32(md5(pass)) . user)** and **H(hex32(md5(pass)) . ":" . user)**. The **user** slot is iterated against the **−u** user list (or the salt list, depending on options).

Single-byte salt search with colon variants. The HEXSALT family (**e183** MD5HEXSALT, **e184** SHA1HEXSALT, **e185** SHA256HEXSALT, **e186** GOSTHEXSALT, **e187** HAV128HEXSALT) iterates a single 2-hex-digit salt across all 256 candidate bytes and, for each candidate, emits **three** digests:

```
H(hex32(md5(pass)) . hex32(salt) . ":")
H(hex32(md5(pass)) . hex32(salt))
H(hex32(md5(pass)) . hex32(salt) . ":" . hex32(salt))
```

The matching salt is reported as a 2-hex-digit value in the output.

The **e687** (**SHA1SHA1MD5MD5PASS1SALT**) and **e649** (**SHA1MD5BASE641SALT**, already covered by Note

[7]) types also perform a 1-byte salt search and emit multiple candidate digests per salt.

Truncation-length search emit (TRUNC suffix). The TRUNC family (**e622** SHA1SHA256TRUNC, **e626** SHA1SHA1TRUNC, **e636** SHA1SHA512TRUNC, **e644** SHA1MD6TRUNC, **e656** SHA1SHA1CAPTRUNC, **e661** SHA1SHA1UCTRUNC, **e678** SHA1SHA256TRUNCSALT, **e689** SHA1SHA512UCTRUNC, **e690** SHA1SALTSHA512UCTRUNC, **e691** SHA1SALTSHA256UCTRUNC, **e745** SHA1SHA384TRUNC, **e746** SHA1RMD160TRUNC, **e750** SHA1SHA1SHA1TRUNC, **e755** SHA1SHA1TRUNCMD5, and related) sweeps a truncation length **N** across a range (typically 21–40 or 21–64) and for each **N** emits **two** digests — one truncating the inner digest from the **left** (taking the first **N** bytes) and one truncating from the **right** (taking the last **N** bytes). The Appendix A expression shows the left-truncation form only; the right-truncation form is also checked. The reported salt position carries the matching **N** as a decimal string.

Multi-salt-placement emit (DSALT suffix). The types **e590** (**SHA1MD5DSALT**) and **e591** (**SHA1MD5MD5DSALT**) combine a multi-byte salt with a single-byte salt search. For each candidate multi-byte salt **s**, mdxfind emits four colon-aware variants:

```
sha1(md5(pass . s) . ":" . s)
sha1(md5(pass . s) . s)
sha1(md5(pass . s) . s . s)
sha1(md5(pass . s) . ":" . s . s)
```

then iterates a fifth variant over 119 single-byte salt values and emits an additional digest per byte.

Multi-form SQL5 family (SHA1SQL5, SHA1SQL5MD5, etc.). e342 (**MD5SQL5-32**) belongs to this family too: it renders the inner digest in both upper and lower case and emits eight digests per password, over each case with and without the leading **1** and ***** characters.

The SHA1SQL5 multi-emit construction is documented in Note [1]. The related types **e712** (**SHA1MD5SQL5**), **e713** (**SHA1MD5MD5SQL5**), **e760** (**SHA1SQL5MD5**), and **e761** (**SHA1SQL5MD5MD5**) share the same three-variant SQL5 emit pattern, applied after one or more MD5 wrappings of the password.

Iterated salt-set emit (MD52SALTMD5 family). The **e283** (**MD52SALTMD5**), **e348** (**MD52SALTMD5MD5**), and **e349** (**MD52SALTMD5MD5MD5**) types iterate **i** and **j** each through the printable byte range 9–127 and for each (**i**, **j**) pair emit three salt-placement variants of the digest (**i j** prefix, raw **unprefixed,** and **suffix variants).**

Base64-RAW padding-variant emit. The **e237** (**MD5BASE64MD5RAW**) type, its UC counterpart **e375** (**MD5UCBASE64MD5RAW**), and the three composites that share its code path — **e242** (**MD5BASE64MD5RAWSHA1**), **e243** (**MD5BASE64MD5RAWMD5**) and **e244** (**MD5BASE64MD5RAWMD5MD5**) — emit **both** the full-length base64 string and a 2-character-shorter variant (the base64 encoding without trailing **==** padding) per password.

The SHA1RAW pair (**e238** MD5BASE64SHA1RAW, **e376** MD5UCBASE64SHA1RAW) emit **once** each and are not part of this family. The variant exists because an md5 digest is 16 bytes, whose base64 encoding ends in two **=** characters; a sha1 digest is 20 bytes and its base64 encoding ends in one, so there is no two-character variant to take.

PHPBB3 colon-prefix emit. The **e770** (**SHA1BASE64CUSTBASE64MD5**) type emits four variants of the **{MD5}**.base64**.digest** construction, with and without the trailing newline and with and without the closing **=** padding character on the base64 output.

TRUNC families: two cuts per length. The **TRUNC** types truncate an intermediate **hex** digest to **N** characters and hash the result. For each **N**, most of these types emit **two** digests — one taking the **leading N** hex characters and one taking the **trailing N** — so a single password yields roughly twice as many candidate digests as there are values of **N**:

```
sha1(cut(H(pass), 0, N))      /* leading  */
sha1(cut(H(pass), -N, N))     /* trailing */
```

N counts down from the full hex width of the inner digest to the lower bound given in each type row; at **N** = the full width the two cuts coincide and only one distinct digest results. The value of **N** is reported in the salt field of mdxfind output, and for the salt-bearing types as "**salt N**".

The following emit the **leading** cut only: **e635** (SHA1WRLTRUNC), **e638** (SHA1SHA512TRUNC1SALT), **e644** (SHA1MD6TRUNC), **e653** (SHA1WRLUCTRUNC), **e657** (SHA1MD6CAPTRUNC), **e672** (SHA1WRLUCTRUNCSALT), **e683** (SHA1MD5TRUNCSALT), and **e739** (SHA1SHA1TRUNCSALT).

Single-byte salt (1SALT). The **1SALT** types **e638** (SHA1SHA512TRUNC1SALT) and **e639** (SHA1SHA1PASS-TRUNC1SALT) do **not** take an external salt. The **salt** shown in their expressions is a single byte that mdxfind generates internally, iterated across the printable range **0x09–0x7f** (119 values), in the same manner as **e348** and **e349**. Any **–s** salt file is ignored. The byte is reported in the salt field as "**N 0xKK**" for **e638** and "**0xKK**" for **e639**, which has no truncated digest of its own: it hashes the complete **e601** (**SHA1PASS-TRUNC**) digest with the byte appended.

BCRYPT256 (e577): two constructions, selected by the salt form. The type row gives the **$2a$** form. A **$2k$** salt selects a **different** algorithm entirely — Python **passlib.hash.bcrypt_sha256** — and mdxfind computes both:

```
$2a$ $2b$ $2y$   bcrypt(sha256(pass), salt)
$2k$             bcrypt(base64(hmac_sha256(key=salt, msg=pass)), salt)
```

The **$2k$** form is **59** characters, not 60:

```
$2k$ <cost> $ <21 salt chars> <31 checksum chars>
```

Only 21 of the 22 salt characters are stored. The 22nd carries just two significant bits, so it is dropped and recovered by trying the four canonical values **.euO**, which is why a single **$2k$** salt yields four candidate digests. Accordingly **–z** emits **five** hashes per password for this type: one **$2a$** and four **$2k$**.

Source for the **$2k$** form is KoreLogic's own CMIYC 2024 write-up, which names **passlib.hash.bcrypt_sha256** directly; samples are archived under **contest/history/2024**. Those samples carry no plaintexts, so the construction was confirmed instead against vectors generated independently of both tools.

HUM families: two placements, and an inner iteration. The seven **HUM** types append a short line-ending sequence **S** to the hex of an inner digest and hash the result. Two things about them are not visible in the type row.

First, each emits **two** digests per (**S**, **N**) pair — the separator is placed **both** after and before the hex:

```
OUTER(hex(inner) . S)      /* append  */
OUTER(S . hex(inner))      /* prepend */
```

S runs over eight values: empty, a space, and the line-ending combinations **\ r**, **\ n**, **\ r\ n**, **\ n\ r**, **\ r\ r** and **\ n\ n**.

Second, **N** is an **inner iteration index**, not a repeat count for **S**. mdxfind computes **inner** once from the password and then re-hashes the **hex** of that digest, emitting at every round:

```
inner(0) = INNER(pass)
inner(k) = INNER(hex(inner(k-1)))
```

The salt field records the pair as "**<sephex>- x N**", so a salt of **0a- x 2** means separator **0x0a**, with the inner hash applied twice. Reading **N** as a repeat count for the separator bytes does not reproduce any emitted digest.

Two expressions in this family were previously recorded inverted and are corrected above: **e575** and **e576** hash the UTF-16LE expansion with **MD4**, so the outer hash is the NTLM-shaped one their outer-first names already imply, not the inner **md5/sha1**. **e378** places **S** inside the **sha1**, before the outer **md5**.

13.1.25 Note [25]

SHA1-CUSTOMUSERSALT (e535). The username is carried *inside* the stored hash, after the base64 field, and is hashed together with a fixed 15-byte pepper baked into mdxfind. Writing **A** for the SHA-1 of the password:

```
A = sha1(pass)
B = sha1(A)
C = sha1(user . "r2chYO214w>1a32" . B)
C = C ^ A
emit(base64(C) . ":" . user)
```

The XOR covers the whole 20-byte digest: mdxfind writes it as **md5buf.v[0..2] ^= curin.v[0..2]** and **v** is an array of

unsigned long long, so three elements span 24 bytes, not 12.

13.1.26 Note [26]

MD5AM (e432) and MD5AM2 (e433). Both read the **–u** channel, not **–s**, and both fold the password to lower case before hashing. A single **–u** entry supplies *two* fields, split at the first colon; the rows above spell the first field **user** and the second **salt2** only because the language has no operator that selects a field of a slot. Writing the entry as **u1:u2**:

```
e432  md5(u1 . "::" . lower(pass))
e433  md5(u1 . "::" . lower(pass) . ":" . u2 . ":73@^bhhs&#@&^@8@*$")
```

The doubled colon is not a separator between two supplied values: mdxfind stages the lowered password in a buffer whose two preceding bytes are both set to **':'**, then copies the first user field in behind them. For e433 the second field is appended after the password together with the colon that delimited it, followed by the 19-byte literal shown, which begins with a colon of its own. An entry with no colon leaves e433 with the same value as e432, since the append is guarded on the delimiter being present.

13.1.27 Note [27]

SHA1MD5SALTPASSPEPPER (e607). The type name is what settles the intended form. In this family a **SALT** token immediately after **SHA1** marks a salt in the outer concatenation, as in **SHA1SALTMD5PASSPEPPER** (e602), **sha1(salt . md5(pass) . pepper)**. e607 carries no such token, so its only salt is the one inside the inner digest, and the outer hash covers the digest and the pepper alone. The sibling types in the same block confirm it: they anchor the digest at a fixed point in the buffer, place the salt before it and the pepper after it, and hash the whole span.

13.1.28 Note [28]

MD4UTF16UC (e570). The expression is the first iteration only. Under **–i** this type re-applies the ENCODING as well as the digest, so the chain is

```
x(n+1) = upper(md4(utf16le(x(n))))
```

and not the uppercase digest chain that **md4_uc^N** would give. Verified against all five iterations of **–i 5**. The **^N** operator iterates a single function rather than a composite, so no output-role form expresses this and the row is deliberately left describing x01 alone. Worth contrasting with **SHA1UCUTF16LE (e730)** which looks like the same shape but does *not* re-encode: its chain is plain **sha1_uc^N(utf16le(pass))**, and it is written that way. The two are not interchangeable.

13.1.29 Note [29]

MD4UTF16 (e496), NTLM (e369) and NTLMH (e786). All three rows give the same primary form, **md4(utf16le(pass))**, and the three compile to identical bytecode. That identity is real only in the degenerate ASCII single-variant case. The types differ in four independent ways, none of which an hx expression carries.

Emissions. Each password produces a different number of candidates: e496 one, e786 two, e369 three. This holds even for pure ASCII, where the values coincide but the multiplicity does not, so the three are distinguishable in output at every input.

Encoding. e369 applies three encodings through **iconv**, UTF-8, CP1251 and CP1252, and dedups identical results with a **memcmp**. e786 reproduces the mapping of UTF-8 to UTF-16 that *hashcat* performs, which is not correct, and exists precisely so that mdxfind agrees with hashcat on inputs where hashcat is wrong. It is a compatibility type, not a specification type. For ASCII every variant collapses to one digest, which is why the three test identical; they separate as soon as a byte above 0x7f appears. With the password **$HEX[636166c3a9]** — the letters c, a, f followed by the two UTF-8 bytes of an e-acute — e369 emits three distinct digests and e786 emits two.

Iteration. e496 honours **–i**, producing x01, x02, x03 and so on. e369 and e786 do not: both call the check at a fixed iteration of one, so they emit x01 only however high **–i** is set, and the repeated x01 lines are the separate encoding variants rather than successive rounds.

13.1.30 Note [30]

HMAC-WRL (e228). The row gives the construction, the full Whirlpool HMAC of 64 bytes. A comparison against **–z** output can show fewer hex characters than that, and the difference is NOT a truncation in the algorithm. The row must not describe one.

Match width in mdxfind is a property of the MATCHER, not of the hash. Two things set it. The HMAC cases assign **hmac_len** beside **hmac_type**, and the shared **HMAC_start** label hands it to **checkhashkey**, where it bounds the span searched; across the family it is twice the digest width in bytes, MD5 32, SHA-1 40, RIPEMD-320 80, SHA-512 128, while **JOB_HMAC_WRL** carries 64. The digest itself is always computed in full.

On top of that, **Minhashlen** tracks the SHORTEST hash in the loaded list — it starts at 256 and each hash read lowers it — and **checkhashkey** slides a window of that width along the computed digest. When the input contains short hashes mdxfind therefore reduces the size of the match on purpose, so that a short hash is found wherever it sits inside a longer digest. Overlap detection is the point.

This is why an oracle comparison against **–z** must compare the CONSTRUCTION and treat a width difference as expected. An earlier revision of this row encoded the observed width as **cut(hmac_wrl(pass, user), 0, 64)** and was wrong to: that asserts the algorithm yields 32 bytes, which it does not.

13.1.31 Note [31]

MANGOS (e540), and why John dynamic_35 must not be mapped to it. The row is correct and uppercases BOTH operands. It is the inner SHA-1 of the WoW and ManGOS SRP6 credential, the value that is then salted and exponentiated to make the verifier, so it is a component rather than a whole stored credential.

John the Ripper carries **dynamic_35** commented **sha1(uc($u).:.$p) (ManGOS)**, and the resemblance is misleading. That row does NOT uppercase the password: its function list appends the userid, the colon constant, then **DynamicFunc__append_keys** unmodified. Checked against John's own three preload vectors for **ELEV__CHARS/test1**, **U1/thatsworking** and **NINECHARS/test3**, all of which reproduce only with the password left as typed. For **SOLAR** and **PASSWORD** this row gives ed03969e..., while John's construction gives eae82942..., so the two are different algorithms that happen to share a name in a comment.

The real ManGOS construction is the one implemented here, and John implements it too, in **wow_srp_fmt_plug.c** rather than in dynamic_35: that file states U and P are both upper case and calls **enc_strupper** on each before hashing. Its inner stage reproduces this row exactly.

Neither **dynamic_35** nor **dynamic_36**, which is the same shape with no uppercasing at all, has any equivalent in this catalog. Their absence from the John map is therefore correct and deliberate, not an omission waiting to be filled. Mapping either onto e540 would attribute cracks to the wrong algorithm, and the failure would be silent, since both produce a well-formed 40-character digest for the same inputs.

13.1.32 Note [32]

RVARY (e1012). IBM DES-ECB of the EBCDIC password, keyed by the RACF key schedule derived from that same password: each byte is mapped through the EBCDIC table, XORed with 0x55, shifted left one bit and given odd parity. The password is truncated to 8 characters, the plaintext padded with EBCDIC space 0x40 and the key with 0x2a. It differs from **e881 RACF** only in that the plaintext is the password rather than the username, and is *not* uppercased. No hx expression is given because the key derivation is not a composable primitive; note also that **racf_encrypt()** uppercases its user operand, so **upper(racf_encrypt(pass, pass))** agrees with the published test vectors only because both use uppercase passwords, and diverges for any lowercase password.

13.1.33 Note [33]

SAMSUNGSHA1 (e867). Samsung Android. 1024 rounds of SHA-1 over the previous digest, the decimal round index, the password and the salt:

```
round 0 : sha1("0" . pass . salt)
round i : sha1(prev . str(i) . pass . salt)
```

str(i) is the decimal representation of the round index and **prev** the digest of the preceding round.

13.1.34 Note [34]

ISCSI-CHAP (e888). iSCSI CHAP.

```
md5(id_byte . pass . fromhex(challenge))
```

The salt field packs two values, the challenge and the identifier, as **challenge_hex**:**id_hex**.

13.1.35 Note [35]

SIPHASH (e902). SipHash-2-4. The row gives the computation. In the stored form the **c** and **d** round counts precede the key.

13.1.36 Note [36]

SAP-PASSCODE (e924) and SAP-PASSCODE5 (e925). SAP CODVN F/G. The digest of the first SHA-1 selects both a length and an offset into a fixed magic string, and the second SHA-1 runs over the password, that slice and the uppercased salt:

```
h = sha1(pass . upper(salt))
n = 32 + sum(h[0..9] mod 6)
o = sum(h[10..19] and 7)
    sha1(pass . magic[o..o+n] . upper(salt))
```

e925 truncates that result and pads it back out to the stored width:

```
cut(SAP-PASSCODE, 0, 20) . "00000000000000000000"
```

RFC_READ_TABLE returns the first 10 bytes. The type name says 5, but the width is 10.

13.1.37 Note [37]

SSPR-SHA256 (e978) and SSPR-SHA512 (e979). SSPR. The salt is the base64 TEXT as stored, not its decoded bytes; the iteration count N comes from the stored format; and the chaining is on the binary digest rather than its hex.

13.1.38 Note [38]

AS400SSHA1 (e986). IBM AS/400 Salted SHA-1. The salt is the userid, uppercased and padded with spaces to 10 characters, and both operands are converted to UTF-16 big-endian:

```
sha1(utf16be(cut(upper(salt) . "          ", 0, 10))
     . utf16be(pass))
```

The padding literal between the quotes is ten spaces.

13.1.39 Note [39]

SYMFONY256 (e992). Symfony. **sha512(pass)** followed by 9999 rounds of SHA-512 over the 128-character hex of the previous digest. The last 4999 of those rounds append the salt.

13.1.40 Note [46]

SUNMD5 (e1027). Solaris crypt. The initial digest is md5(pass . salt); thereafter 4096 plus the rounds= value are run, each computing md5 of the previous digest, optionally a fixed 1517-byte phrase, and the decimal round number in ASCII.

Whether the phrase is included is decided per round by a coin flip derived from bits of the previous digest — two seven-bit indices are assembled from the digest bytes and the XOR of the bits they address gives the flip. Roughly half the rounds therefore hash an extra 1517 bytes, so the cost is data dependent and varies with the candidate rather than being fixed by the rounds= field alone.

The phrase is 1516 characters of Hamlet's soliloquy *plus* its terminating NUL: Sun's implementation passed sizeof(), so 1517 bytes are hashed, and dropping the NUL yields a plausible but wrong digest for every candidate.

The salt is everything before the LAST dollar sign. That definition also covers the variant whose salt ends in a dollar sign and so shows a doubled separator in the stored form, with no special case. The digest is emitted with the md5crypt transposition, 22 characters. All 8 of John the Ripper's vectors verify, including its 120-character password, and mdxfind -z output cracks back.

13.1.41 Note [48]

SHA1MD5USER (e308), MD5CAPMD5USER (e354), MD5CAPMD5MD5USER (e355), MD5MD5MD5USER (e356). Four entry points into one loop in mdxfind, reached by fallthrough, differing only in how the intermediate hex is built: one md5 or two, with the leading hex character capitalised or not. The capitalisation of the *first* hex happens in the e355 pre-block, so a reading that caps only the last md5 agrees with mdxfind whenever that character is a digit and disagrees otherwise — about five times in eight.

For every user, two digests are emitted: one over hex . user and one over hex . ":" . user. Both are iterated, each over its own hex. Registering the pair so that only one of them seeds the iteration chain leaves the type complete at its base depth and exactly half strength at every depth below, which is how the defect presented before it was found.

Earlier revisions of this page gave these expressions as md5(md5(user)) and the like, with no password term at all. That was a transcription error; the password is the inner operand and the username is appended.

13.1.42 Note [50]

MD5DSALT (e408). The two salts are reported in a single field, concatenated with no separator, so a line carrying the salt **Kpa** does not say whether that is **Kp** then **a**, or **K** then **pa**. Anything consuming this type has to try each split; mdxfind bounds each salt at three characters, so at most three positions need testing. A reader that assumes an even split agrees only when the two salts are the same length.

13.1.43 Note [51]

Two UTF-16LE conventions. mdxfind turns a password into UTF-16LE in two different ways, and which one a type uses is not visible from its expression.

The first is a byte expansion: each input byte is followed by a NUL, so the bytes are treated as Latin-1. The second is a real UTF-8 to UTF-16LE conversion through iconv, opened with //IGNORE so that input which is not valid UTF-8 has the bad sequences dropped rather than failing the conversion.

They agree for ASCII and differ for everything else. **DCC2**, **AZURESYNC**, **KRB5PA23** and **KRB5TGS23** use the byte expansion; **MSONLINE** and plain **NTLM** use iconv. An implementation that picks one convention for all of them will verify ASCII passwords and quietly fail every other one.

13.1.44 Note [45]

GOST12256CRYPT (e1025), GOST94CRYPT (e1026). The two remaining members of the GOST crypt family, siblings of **GOST12512CRYPT** (e994). All three are the Drepper schedule — the same one **SHA256CRYPT** (e512) and **SHA512CRYPT** (e513) use — with the SHA replaced by a GOST digest, defaulting to 5000 rounds and accepting an explicit rounds= field.

e1025 substitutes Streebog-256 and e1026 substitutes GOST R 34.11-94. Both produce a 32-byte digest and so use the sha256crypt transposition and a 43-character stored hash, where e994 uses the sha512crypt transposition and 86.

e1026 uses the TEST parameter S-box, not CryptoPro — i.e. the same set as **GOST** (e13), not **GOST-CRYPTO** (e14). John the Ripper's gost94hash carries a **john_gost_cryptopro_init** override but leaves it commented out, and the bundled gosthash used here is the test-parameter build, so the two agree.

Streebog output is byte-reversed relative to the published GOST vectors by the bundled Saarinen implementation, so e1025 reverses each digest exactly as e994 does. All of John's vectors for both types verify, and mdxfind -z output cracks back.

13.1.45 Note [44]

DRAGONFLY3-32, DRAGONFLY3-64, DRAGONFLY4-32, DRAGONFLY4-64 (e1021– e1024). DragonFly BSD's own crypt schemes. Both set up like **SHA256CRYPT** (e512) and **SHA512CRYPT** (e513) but return after a *single* digest of pass . magic . salt — there is no round loop, which makes these among the cheapest salted types in the catalogue.

The magic is the format tag *including* its NUL terminator. On 64-bit builds its length was taken as 8 rather than 4, so four adjacent rodata bytes ride along after the NUL: "sha5" for 3 and "/etc" for 4. That is the bug the variant names refer to, and both variants are in the wild. Because the NUL is interior to the hashed string, neither can be expressed as an ordinary salted type with a textual salt.

The digest is emitted with the sha256crypt/sha512crypt transposition, but 4 encodes only 62 of the 64 digest bytes — bytes 62 and 63 never appear in the stored form, so a 4 hash cannot be decoded back to a full digest. mdxfind and hashpipe therefore encode forward and compare strings.

The stored forms of the 32- and 64-bit variants are *identical* — nothing in a 3 or 4 line says which produced it. A line is loaded into both types of its pair and only the correct one verifies. All 24 of John the Ripper's vectors were reproduced by an independent implementation before this code was written, and **mdxfind −z** regenerates John's published strings byte for byte.

13.1.46 Note [43]

ARGON2MD5 (e1020). vBulletin 6. Argon2 over the MD5 of the password rather than the password itself, the direct successor to what **BCRYPTMD5** (e451) models for vBulletin 5. The inner digest is the 32-character lowercase HEX, not the 16 raw bytes, matching e451 and vBulletin's own convention. The stored form is ordinary Argon2 MCF and carries no marker distinguishing it from e987 ARGON2, so a line can be offered to either type and only the correct one verifies.

Neither John the Ripper nor hashcat ships a vector for this scheme. The registered one was produced by **argon2-cffi**, a different implementation from the argon2 library linked here, so the check is across implementations rather than self-generated.

13.1.47 Note [42]

H3C (e1019). H3C, HPE and Huawei device passwords. The stored form is **h6$<salt>$<base64>** and the digest is **sha512(pass . NUL . salt . pass . NUL)** — the password is hashed INCLUDING its terminating NUL, twice, with the salt between the two copies. The 64-byte digest is stored base64-encoded. Verified against all three of John the Ripper's own **h3c** test vectors, base64 exact.

13.1.48 Note [41]

SAPCODVNH256 (e1016), SAPCODVNH384 (e1017). SAP CODVN H in its SHA-256 and SHA-384 flavours. The stored form is **{x-isSHA256, ITER}** or **{x-isSHA384, ITER}** followed by base64 of the digest immediately followed by the salt. The computation is **h = H(pass . salt)** for the first round and **h = H(pass . h)** for each of the remaining **ITER-1** rounds, with the password truncated at 40 characters. The iteration count is carried in the hash rather than fixed by the type, so it is a runtime parameter and these rows are documentation rather than a compilable expression.

These complete the family: e904 SAPCODVNH is the SHA-1 flavour and e984 SAPCODVNH512 the SHA-512 one. All four of John the Ripper's **saph** SHA-256 and SHA-384 test vectors were reproduced by an independent implementation before either type was written.

13.1.49 Note [40]

ORACLE11 (e1015). Oracle 11g. The digest is **upper(sha1(pass . fromhex(salt)))** — the same value **SHA1PASSHEXSALT** (e834, hashcat mode 112) computes. The two differ only in stored form, and that difference is the reason both exist. e834 takes the digest and the salt as two separate fields; e1015 owns the form Oracle actually stores, a single 60-character field holding the 40-hex digest immediately followed by the 20-hex salt. No other type in the catalogue has a 30-byte digest, so that width identifies the format on its own, with no wrapper. Found lines are reported in the same concatenated form, so a result can be handed straight back to Oracle.

14. Appendix B: Intrinsic Function Reference

This appendix documents every built-in function available in hx, with its default output format and a concrete example using the password `password` and, where applicable, the salt `salt` (the four bytes 0x73, 0x61, 0x6c, 0x74).

All examples can be reproduced with:

```
echo password | hashpipe -X 'expression'
```

14.1 Output Format Suffixes

Every digest-producing function (hash, HMAC, KDF, cipher) supports role suffixes that control the output encoding. The bare name uses the function's default role; the suffixed form overrides it:

Suffix	Role	Description
(none)	default	Function's canonical output (usually hex)
_bin	ROLE_BIN	Raw binary bytes
_hex	ROLE_HEX	Lowercase hexadecimal
_uc	ROLE_UC	Uppercase hexadecimal
_b64	ROLE_B64	RFC 4648 base64 with = padding
_mcf	ROLE_MCF	Modular Crypt Format (crypt-family only)

TABLE 16. Output format suffixes

Not every suffix is valid for every function. The supported combinations are:

- **ROLES_DIGEST** (hash, HMAC, KDF) — supports _bin, _hex, _b64, _uc. Default is _hex.
- **ROLES_CRYPT** (bcrypt, yescrypt) — supports _bin, _hex, _b64, *and* _mcf. Default is _mcf.
- **ROLES_NONE** (string transforms, encoders) — no suffix accepted. The function returns a string directly.

To illustrate why suffixes matter, consider **bcrypt**. The default (MCF) output is the familiar crypt string that encodes the algorithm, cost, salt, and hash together:

```
$ echo password | hashpipe -X 'bcrypt(pass, fromhex("6162636465666768696a6b6c6d6e6f70"), 5)'
$2b$05$WUHhXETkX0fnYkrqZU3ta.xKSDhHZGYxrgOUh/C3/852ZM1RjPsCa
```

This is self-contained: any bcrypt implementation can verify the password against this string. The alternative forms strip the metadata:

```
bcrypt_hex(...)  ccc5058c96c86b3b624168c113907eef86cedd3951b847
bcrypt_b64(...)  xKSDhHZGYxrgOUh/C3/852ZM1RjPsCa
bcrypt_bin(...)  (23 raw bytes)
```

These contain only the 23-byte hash output. Without the salt and cost factor, they are *useless* for verification — you cannot reconstruct the bcrypt input from the raw hash alone. The _hex and _bin forms exist for compositions where bcrypt's output feeds into another function (e.g., a site-specific wrapper that computes `sha256(bcrypt_bin(pass, salt, cost))`), not for standalone storage.

14.2 Cryptographic Hash Functions

These are the core message digest functions. Each takes a single byte-string argument and returns the digest. Default output is lowercase hex.

All hash functions registered by hashpipe (approximately 1000 types) are available as hx intrinsics. The table below lists the most commonly used primitives.

Function	Digest Size	Example Output (pass = "password")
md4(pass)	128 bit	8a9d093f14f8701df17732b2bb182c74
md5(pass)	128 bit	5f4dcc3b5aa765d61d8327deb882cf99
sha1(pass)	160 bit	5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
sha224(pass)	224 bit	d63dc919e201d7bc4c825630d2cf25fdc93d4b2f0d46706d29038d01
sha256(pass)	256 bit	5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
sha384(pass)	384 bit	a8b64babd8acb29e42e9a9862daa23bb†
sha512(pass)	512 bit	b109f3bbbc244eb82441917ed06d618b†

TABLE 17. Core hash functions (default hex output)

† Output truncated for display; full output is 96 or 128 hex characters.

The same value in all three encodings:

```
md5(pass)       5f4dcc3b5aa765d61d8327deb882cf99
md5_b64(pass)   X03MO1qnZdYdgyfeuILPmQ==
md5_bin(pass)   (16 raw bytes)
md5_hex(pass)   5f4dcc3b5aa765d61d8327deb882cf99
```

Note that **md5(pass)** and **md5_hex(pass)** produce identical output: the bare name defaults to hex for all digest functions. The explicit _hex suffix is available for clarity in complex expressions.

Additional hash functions available include (but are not limited to): md2, sha0, gost, gost_crypto, rmd128, rmd160, tiger, tth, ed2k, aich, snefru128, snefru256, hav128, hav160, hav192, hav224, hav256, whirlpool, sha3_224, sha3_256, sha3_384, sha3_512, keccak224, keccak256, keccak384, keccak512, blake2b_256, blake2b_512, blake2s_128, blake2s_256, skein256, skein512, skein1024, streebog256, streebog512, panama, sm3, and all BLAKE variants. Every hash type known to hashpipe is automatically registered as an hx function.

14.3 HMAC Functions

HMAC functions take two arguments: the key and the message. The key is typically the password; the message is typically the salt or a derived value.

Function	Digest Size	Example (key = "password", msg = "salt")
hmac_md5(pass, salt)	128 bit	dda73d480bbd86b0da55b1480e7d0b30
hmac_sha1(pass, salt)	160 bit	c1d0e06998305903ac76f589bbd6d4b61a670ba6
hmac_sha256(pass, salt)	256 bit	84ec44c7d6fc41917953a1dafca3c7d7†
hmac_sha512(pass, salt)	512 bit	(128 hex characters)

TABLE 18. HMAC functions (default hex output)

† Truncated for display. HMAC functions support _bin, _hex, and _b64 suffixes.

14.4 Key Derivation Functions

KDFs take additional parameters for iteration count and output length.

Function	Parameters	Default Role
pbkdf2_sha1(pass, salt, iter, dklen)	4	hex
pbkdf2_sha256(pass, salt, iter, dklen)	4	hex
pbkdf2_sha512(pass, salt, iter, dklen)	4	hex
pbkdf2_md5(pass, salt, iter, dklen)	4	hex
pbkdf1_sha1(pass, salt, iter, dklen)	4	hex
bcrypt(pass, salt, cost)	3	mcf
yescrypt(pass, salt)	2	mcf
scrypt(pass, salt, N, r, p, dklen)	6	hex
argon2id(pass, salt, t, m, p, dklen)	6	hex
argon2i(pass, salt, t, m, p, dklen)	6	hex
argon2d(pass, salt, t, m, p, dklen)	6	hex
argon2(pass, salt, t, m, p, dklen)	6	hex (alias for argon2id)
pomelo(pass, salt, t_cost, m_cost)	4	hex

TABLE 19. Key derivation functions

Example:

```
$ echo password123 | hashpipe -X 'pbkdf2_sha256(pass, fromhex("73616c74"), 1000, 32)'
632c2812e46d4604102ba7618e9d6d7d2f8128f6266b4a03264d2a0460b7dcb3

$ echo password123 | hashpipe -X 'pbkdf1_sha1(pass, fromhex("73616c74"), 1000, 20)'
c5baaad10eb03e9602037df936ccf3f5e8d9b4e8

$ echo password123 | hashpipe -X 'pomelo(pass, fromhex("73616c74"), 2, 3)'
79bfb3e29e6be446560e1c52a06690734ef28634f662a31ff11b9973514c6be0
```

pomelo() is the PHC (Password Hashing Competition) submission POMELO v2 by Hongjun Wu. The **t_cost** and **m_cost** parameters control time and memory hardness; mdxfind uses t_cost=2, m_cost=3 by default.

The crypt-family KDFs (bcrypt, yescrypt) default to MCF output, which includes the algorithm identifier, cost parameters, and salt in a single self-verifying string. See the bcrypt example in the Output Format Suffixes section above.

14.5 Crypt-Family Functions

These functions implement the Drepper-pattern crypt algorithms (and related MCF-producing functions). Default output is MCF (Modular Crypt Format); all support _hex, _b64, _bin suffixes to extract the raw hash digest.

Function	Parameters	MCF Prefix
md5crypt(pass, salt)	2	1
apr1(pass, salt)	2	$apr1$
sha256crypt(pass, salt [, rounds])	2–3	5
sha512crypt(pass, salt [, rounds])	2–3	6
sm3crypt(pass, salt [, rounds])	2–3	$sm3$
gost12_512crypt(pass, salt [, rounds])	2–3	$gost12512hash$
descrypt(pass, salt)	2	(13-char)
phpass(pass, salt, log2count)	3	H

TABLE 20. Crypt-family functions (default MCF output)

Examples (pass = "password123", salt = "salt"):

```
md5crypt(pass, salt)
  $1$salt$/3NHsNrNmNbOO90IOW9dw/

sha256crypt(pass, salt)
  $5$salt$XLWJeVAbH3W178D74IkDX6.dlQByMbBzb/KgZVmw486

sha512crypt(pass, salt)
  $6$salt$cevuzTZ/QBjzuZG0/ebEeedm...

sm3crypt(pass, salt)
  $sm3$salt$8ZfxTI2EU8nZHmzZjhleiDc7S/jAjfTrsRQv1bXp2l5

gost12_512crypt(pass, salt)
  $gost12512hash$salt$6ORzFltcwpBfn0QJn9XWHk.7q3Wi2...

descrypt(pass, "ab")
  abJnggxhB/yWI

phpass(pass, "saltsalt", 11)
  $H$9saltsaltngurp8A.5QxWvhTtS2J2I0
```

The _hex suffix extracts the raw hash digest as hex, stripping the MCF wrapper:

```
md5crypt_hex(pass, salt)
  4d676a50a49a91767622b87c417872da

md5crypt_b64(pass, salt)
  TWdqUKSakXZ2Irh8QXhy2g==
```

For crypt functions with a **rounds** parameter, the default is 5000. Custom round counts are embedded in the MCF output:

```
sha256crypt(pass, salt, 10000)
  $5$rounds=10000$salt$2L6cVpxJfG/se8rqocEPWP1W9Lh4ma7rj9nLxMmPb0A
```

14.6 Kerberos and Protocol Functions

These functions implement the key derivation steps used by Kerberos authentication protocols.

Function	Args	Description
rc4_hmac_md5(pass)	1	Kerberos etype 23 key (HMAC-MD5 of NTLM hash)
aes128_cts_hmac_sha1(pass, salt)	2	Kerberos etype 17 key (PBKDF2-SHA1, 16 bytes)
aes256_cts_hmac_sha1(pass, salt)	2	Kerberos etype 18 key (PBKDF2-SHA1, 32 bytes)

TABLE 21. Kerberos and protocol functions

Examples (pass = "password123"):

```
rc4_hmac_md5(pass)
  358bd5d2c85ea4cddfa181278c361fdd

aes128_cts_hmac_sha1(pass, salt)
  06ca0b78b558933164fc00313fe02a10

aes256_cts_hmac_sha1(pass, salt)
  06ca0b78b558933164fc00313fe02a10\
  52738b98c736e77da8f3080a84f5c3bc
```

rc4_hmac_md5() computes the Kerberos etype 23 base key: HMAC-MD5(MD4(UTF16LE(pass)), usage=1). The

AES functions compute the string-to-key derivation: PBKDF2-SHA1(pass, salt, 4096, keylen). These are the password-derived keys used in KRB5PA23, KRB5TGS23, KRB5PA-17, KRB5PA-18, KRB5DB17, and KRB5DB18 hash types.

14.7 Non-Cryptographic Hash Functions

Function	Args	Example (pass = "password123")
siphash(pass, key)	2	4e9e3a4211a94859 (key = 00..0f)
murmur3(pass, seed)	2	43d88d27 (seed = 0)

TABLE 22. Non-cryptographic hash functions

siphash() implements SipHash-2-4 with a 16-byte key. **murmur3()** implements MurmurHash3_x86_32 with a 32-bit seed (default 0). Both support _bin, _hex, _b64 suffixes.

14.8 String Transform Functions

These functions operate on byte strings and return strings. They do not support role suffixes.

Function	Args	Description
hex(x)	1	Encode bytes as lowercase hex
upper(x)	1	Convert to uppercase
lower(x)	1	Convert to lowercase
rev(x)	1	Reverse byte sequence
bswap32(x)	1	Reverse bytes within each 4-byte group
rotate(x, N)	2	Rotate string by N positions
cap(x)	1	Capitalize first letter
cap(x, N)	2	Capitalize letter at position N
rot13(x)	1	ROT13 substitution cipher
cut(x, off, len)	3	Extract substring
pad(x, len, byte)	3	Pad to length with byte
trunc(x, len)	2	Truncate to length

TABLE 23. String transform functions

Examples:

```
upper(md5(pass))     5F4DCC3B5AA765D61D8327DEB882CF99
cut(sha512(pass), 0, 32)
                     b109f3bbbc244eb82441917ed06d618b
rev(md5(pass))       99fc288bed7238d16d567aa5b3cc4d5f
hex(bswap32(sha1_bin(pass)))
                     e461aa5b3f3fb9c90b2582061b33f86cd88fe67e
```

14.9 Encoding Conversion Functions

These convert between byte encodings and do not support role suffixes.

Function	Args	Description
base64(x)	1	RFC 4648 base64 encode
frombase64(x)	1	RFC 4648 base64 decode
fromhex(x)	1	Hex string to raw bytes
utf16le(x)	1	UTF-8 to UTF-16 little-endian
utf16be(x)	1	UTF-8 to UTF-16 big-endian
utf7(x)	1	UTF-8 to UTF-7
zext16(x)	1	Zero-extend each byte to 16 bits LE
from_cp1252(x)	1	UTF-8 to Windows-1252
from_cp1251(x)	1	UTF-8 to Windows-1251
ebcdic(x)	1	ASCII to IBM EBCDIC CP037

TABLE 24. Encoding conversion functions

Example — NTLM is MD4 over the UTF-16LE encoding of the password:

```
$ echo password | hashpipe -X 'md4(utf16le(pass))'
8846f7eaee8fb117ad06bdd830b7586c
```

14.10 Custom Encoding Functions

These produce application-specific encodings used by particular hash schemes. They do not support role suffixes, and return ASCII text.

Function	Args	Description
phpass_encode(x)	1	PHPass/Drupal custom base64
sha512crypt_encode(x)	1	SHA-512/256-crypt permuted base64
bcrypt_encode(x)	1	Bcrypt Radix-64 encoding
progress_encode(x)	1	Progress/OpenEdge encoding
juniper_encode(x)	1	Juniper 9 encoding
cisco_pix_encode(x)	1	Cisco PIX base64 (no padding)
lotus64_encode(x)	1	Lotus Notes/Domino base64
besder_encode(x)	1	Besder/Dahua encoding
dahua_encode(x)	1	Dahua legacy encoding

TABLE 25. Custom encoding functions

These are needed because several password hash schemes use non-standard base64 alphabets or permuted byte orderings that cannot be expressed as a simple **base64()** call.

14.11 Cipher Primitives

Low-level cipher operations for hash schemes that include encryption or decryption steps.

Function	Args	Description
des(key, pt)	2	Single DES-ECB encrypt (8-byte key, 8-byte block)
des_block(key7, pt)	2	DES-ECB with 7→8 byte key expansion (NTLMv1)
des3(key, pt)	2	Triple-DES ECB encrypt
aes_ecb_encrypt(key, pt)	2	AES-ECB encrypt
aes_cbc_encrypt(key, iv, pt)	3	AES-CBC encrypt
aes_cbc_decrypt(key, iv, ct)	3	AES-CBC decrypt
aes_unwrap(kek, wrapped)	2	RFC 3394 AES Key Unwrap

TABLE 26. Cipher primitive functions

These support _bin, _hex, and _b64 suffixes. Default is hex. The _bin suffix is typically needed when the cipher output feeds into another function:

```
pt = aes_cbc_decrypt_bin(key, iv, ct)
```

14.12 Proprietary Algorithm Primitives

Opaque implementations of vendor-specific password mixing algorithms. These support _bin, _hex, _b64 suffixes (default hex) except for the encoder-style functions (domino6, lotus64_encode) which return ASCII text directly.

Function	Args	Description
domino5(pass)	1	Lotus Domino 5 mixer
domino6(pass)	1	Lotus Domino 6 (returns ASCII)
oracle7(user, pass)	2	Oracle 7–10 DES chain
sap_bcode(user, pass)	2	SAP CODVN B (BCODE)
as400_des(user, pass)	2	IBM AS/400 DES hash
axcrypt(pass)	1	AxCrypt key-file mixer
racf_encrypt(user, pass)	2	IBM RACF DES hash
racf_kdfaes(pass, user, hdr, salt)	4	IBM RACF KDFAES

TABLE 27. Proprietary algorithm primitives

14.13 Bitwise and Arithmetic Functions

Function	Args	Description
xor(a, b)	2	Byte-wise XOR
and(a, b)	2	Byte-wise AND
or(a, b)	2	Byte-wise OR
wperm(x, w0, w1, ...)	2+	Permute 4-byte words
add(a, b)	2	Integer addition
sub(a, b)	2	Integer subtraction
mul(a, b)	2	Integer multiplication
mod(a, b)	2	Integer modulo
length(x)	1	Byte length of argument

TABLE 28. Bitwise and arithmetic functions

14.14 Output Function

Function	Args	Description
emit(x)	1	Write x to stdout, return x (passthrough)

TABLE 29. Output function

emit() is used for hash types that produce multiple output values from a single password. The final expression result is always emitted automatically; **emit()** is needed only for intermediate values.