

Vulnerabilità nei software registrabili tramite Serial Key

Introduzione

L'obiettivo dell'analisi proposta è esaminare le vulnerabilità presenti all'interno dei software che richiedono l'attivazione tramite Serial Key. L'analisi di tali vulnerabilità è fondamentale per comprendere come i meccanismi di protezione possono essere aggirati, consentendo l'uso non autorizzato del software.

La dimostrazione pratica riportata illustrerà concretamente come un attaccante potrebbe approfittarne per attivare il software in modo illecito.

Premesse

Alcuni software vengono distribuiti inizialmente in versione demo e successivamente possono essere attivati su un dispositivo tramite l'acquisto di una Serial Key fornita dai produttori. Solitamente, esistono due scenari di attivazione:

Online: Nel momento in cui l'utente inserisce la Serial Key, il software comunica con i server del produttore per verificare la correttezza della chiave e restituisce l'esito all'utente.

Offline: Nel momento in cui l'utente inserisce la Serial Key, il software esegue un controllo interno verificando che la chiave sia valida rispetto al nome utente specificato, utilizzando un algoritmo.

È importante notare che alcuni software non richiedono una registrazione e non distinguono tra utenti; il controllo eseguito sulla Serial Key in input può limitarsi a verificare che la sintassi della chiave rispetti un determinato pattern, individuando così un altro tipo di vulnerabilità che non sarà oggetto di discussione in questo approfondimento.

Vulnerabilità nello scenario offline

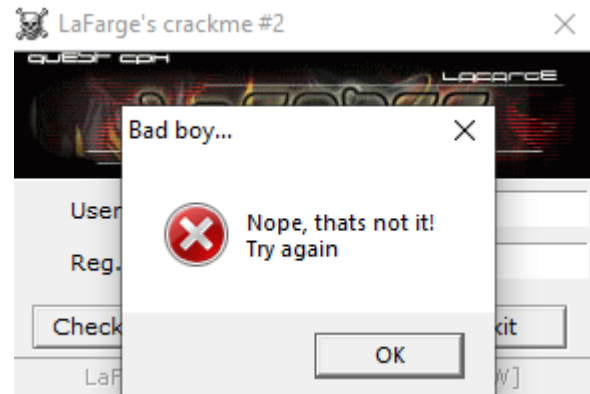
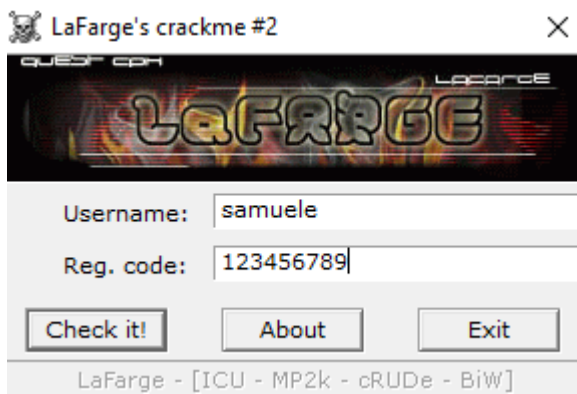
Il processo di computazione della Serial Key coinvolge 3 passaggi principali:

Input: *nome_utente, serial_key*

Computazione: $\text{func}(k, \text{nome_utente}) = \text{correct_key}$ (dove k è un valore segreto)

Check: $\text{correct_key} == \text{serial_key}$

Per la dimostrazione pratica riportata di seguito, si fa uso di **LaFarge's crackme #2**, un software open source utilizzato per sfide di reverse engineering che simula lo scenario appena descritto.



LaFarge's crackme #2

Obiettivi:

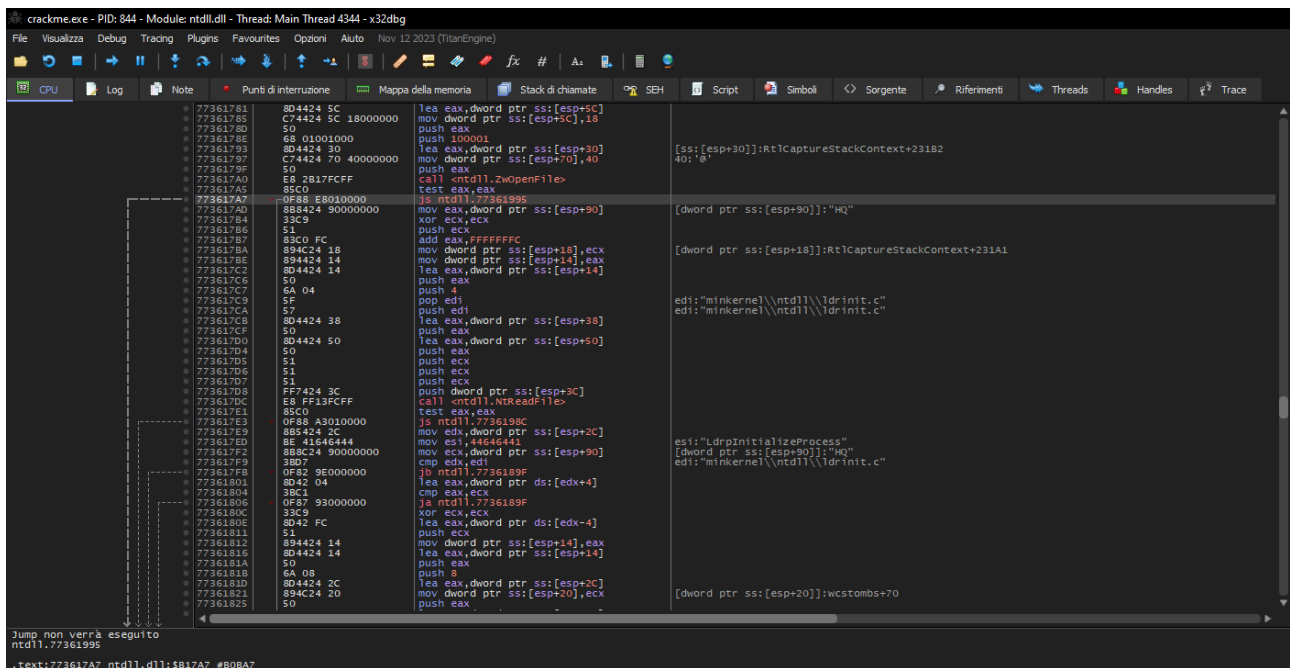
- acquisizione *serial_key* per *nome_utente* = 'samuele'
- sviluppo di un generatore di chiavi (**keygen**) valido per qualsiasi *nome_utente*

Acquisizione Serial Key

Per poter indentificare eventuali vulnerabilità in un software, è necessario studiarne il comportamento. Nel caso specifico, osserviamo che inserendo una *serial_key* errata, il software restituisce una finestra di dialogo contenente un messaggio di errore. Utilizzando un debugger, possiamo individuare la parte di codice responsabile di questa situazione con l'obiettivo di ottenere informazioni sulla gestione e il confronto delle chiavi.

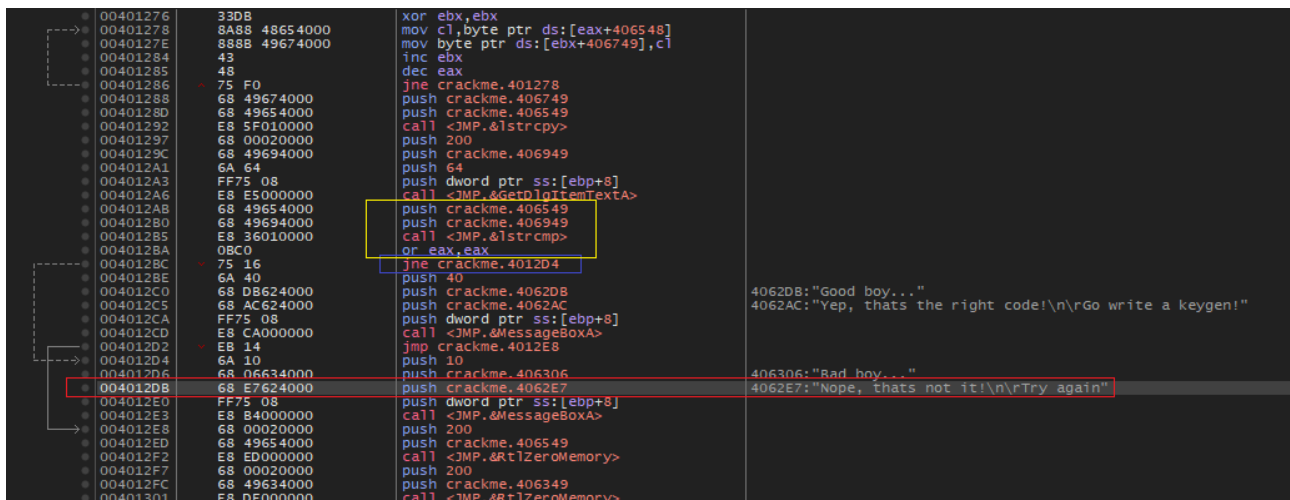
Per condurre un'analisi statica e dinamica del software ci avvaliamo di x32dbg, un debugger gratuito in grado di disassemblare l'eseguibile in codice assembly x86 e di esaminare la memoria a basso livello.

Questo strumento fornisce numerose funzionalità di reverse engineering che ci permettono di comprendere il funzionamento interno del software.



x32dbg

x32dbg offre la possibilità di ricercare stringhe all'interno di un modulo, consentendo di identificare la porzione di codice associata. Sfruttando questa funzionalità e conoscendo il messaggio di errore restituito, siamo in grado di analizzare in modo più dettagliato il codice responsabile della finestra di dialogo.



Ricerca stringa in x32dbg e visualizzazione nel suo contesto

Attraverso una conoscenza basilare dell'assembly x86, notiamo che il **messaggio di errore della finestra di dialogo** (contenuto nel registro **4062E7**) viene caricato su stack in seguito al risultato di un'operazione **'jump not equal' (jne)**. Analizziamo le righe di codice precedenti per comprendere cosa influenzi questa decisione:

`push crackme.406549`: Carica un primo valore sullo stack.

`push crackme.405959`: Carica un secondo valore sullo stack.

`call <JMP.&Istrcmp>`: Indica la chiamata a una funzione delle API di Windows per il confronto tra stringhe, deducendo che i valori caricati su stack sono stringhe.

`or eax, eax`: Operazione logica per manipolare il valore del registro **Zero Flag (ZF)**. Nell'assembly, il registro ZF determina il risultato di un'operazione *jne* in base al risultato dell'operazione logica compiuta in precedenza. In particolare, ZF = 1 quando il risultato è 0 e viceversa.

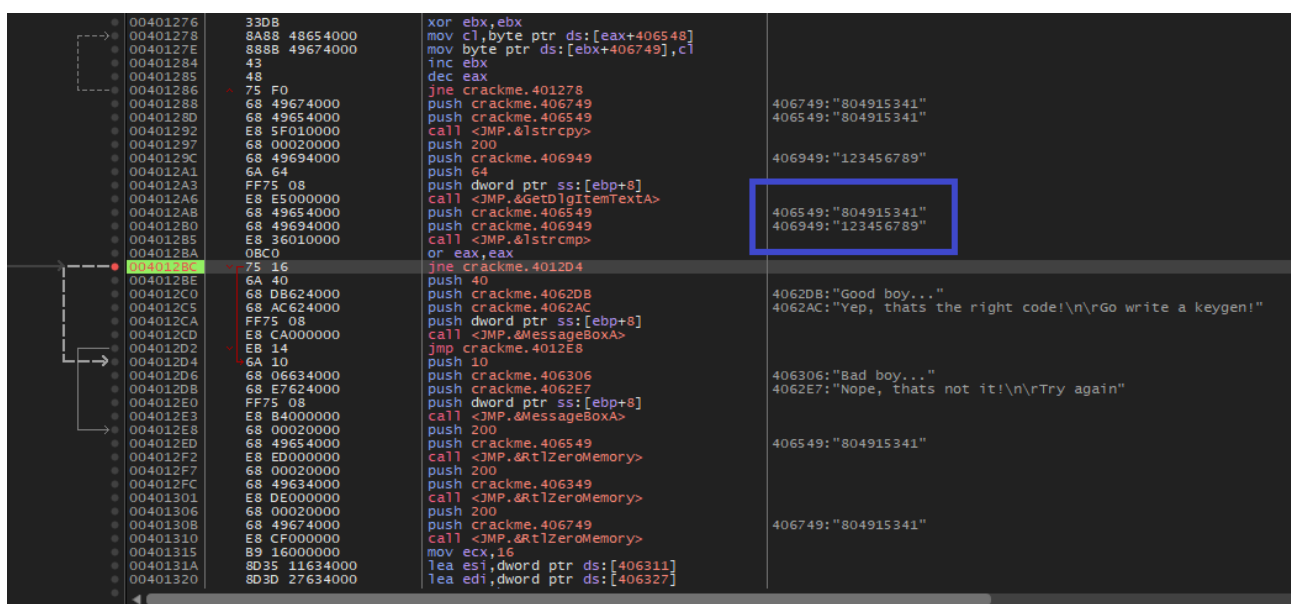
Deduciamo che quando le stringhe caricate sullo stack sono diverse:

or eax, eax = 1 → ZF = 0 → jump → messaggio di errore

quando sono uguali:

or eax, eax = 0 → ZF = 1 → no jump → messaggio corretto

Impostando un breakpoint sull'operazione *jne*, possiamo analizzare dinamicamente cosa accade all'interno del software e leggere i valori contenuti nei registri di nostro interesse.

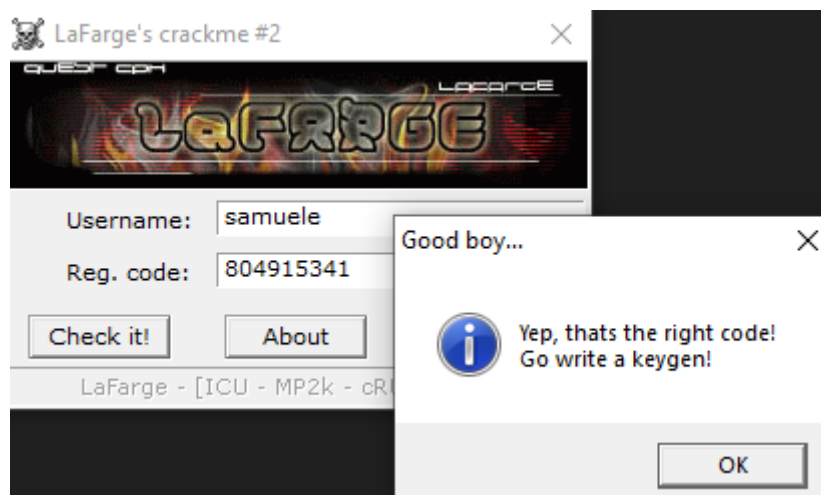


Address	Disassembly	Comment
00401276	33D8	xor ebx,ebx
00401278	8A88 48654000	mov cl,byte ptr ds:[eax+406548]
0040127E	8888 49674000	mov byte ptr ds:[ebx+406749],cl
00401284	43	inc ebx
00401285	48	dec eax
00401286	75 F0	jne crackme.401278
00401288	68 49674000	push crackme.406749
0040128D	68 49654000	push crackme.406549
00401292	E8 5F010000	call <JMP.&Istrcpy>
00401297	68 00020000	push 200
0040129C	68 49694000	push crackme.406949
004012A1	6A 64	push 64
004012A3	FF75 08	push dword ptr ss:[ebp+8]
004012A6	E8 E5000000	call <JMP.&GetDlgItemTextA>
004012AB	68 49654000	push crackme.406549
004012B0	68 49694000	push crackme.406949
004012B5	E8 36010000	call <JMP.&Istrcmp>
004012BA	0BC0	or eax,eax
004012BC	75 16	jne crackme.401204
004012BE	6A 40	push 40
004012C0	68 DB624000	push crackme.4062DB
004012C5	68 AC624000	push crackme.4062AC
004012CA	FF75 08	push dword ptr ss:[ebp+8]
004012CD	E8 CA000000	call <JMP.&MessageBoxA>
004012D2	EB 14	jmp crackme.4012E8
004012D4	6A 10	push 10
004012D6	68 06634000	push crackme.406306
004012DB	68 E7624000	push crackme.4062E7
004012E0	FF75 08	push dword ptr ss:[ebp+8]
004012E3	E8 B4000000	call <JMP.&MessageBoxA>
004012E8	68 00020000	push 200
004012ED	68 49654000	push crackme.406549
004012F2	E8 ED000000	call <JMP.&RtlZeroMemory>
004012F7	68 00020000	push 200
004012FC	68 49634000	push crackme.406349
00401301	E8 DE000000	call <JMP.&RtlZeroMemory>
00401306	68 00020000	push 200
0040130B	68 49674000	push crackme.406749
00401310	E8 CF000000	call <JMP.&RtlZeroMemory>
00401315	B9 16000000	mov ecx,16
0040131A	8D35 11634000	lea esi,dword ptr ds:[406311]
00401320	8D3D 27634000	lea edi,dword ptr ds:[406327]

Address	Value
406749	"804915341"
406549	"804915341"
406949	"123456789"
4062DB	"Good boy..."
4062AC	"Yep, thats the right code!\n\rGo write a keygen!"
406306	"Bad boy..."
4062E7	"Nope, thats not it!\n\rTry again"
406549	"804915341"
406749	"804915341"

Breakpoint e visualizzazione dei valori contenuti nei registri usati dal software

Le stringhe contenute nei registri 406549 e 406949 sono *correct_key* computata e *serial_key* di input. Verifichiamo la correttezza della prima rispetto al *nome_utente* = 'samuele':



Verifica correttezza della chiave computata

Sviluppo Keygen

Una volta identificata la vulnerabilità nel software possiamo sviluppare un keygen con l'intento di distribuirlo in rete. Questo consentirebbe a chiunque di utilizzare il software in questione senza la necessità di utilizzare un debugger.

Dalla precedente discussione abbiamo dedotto che:

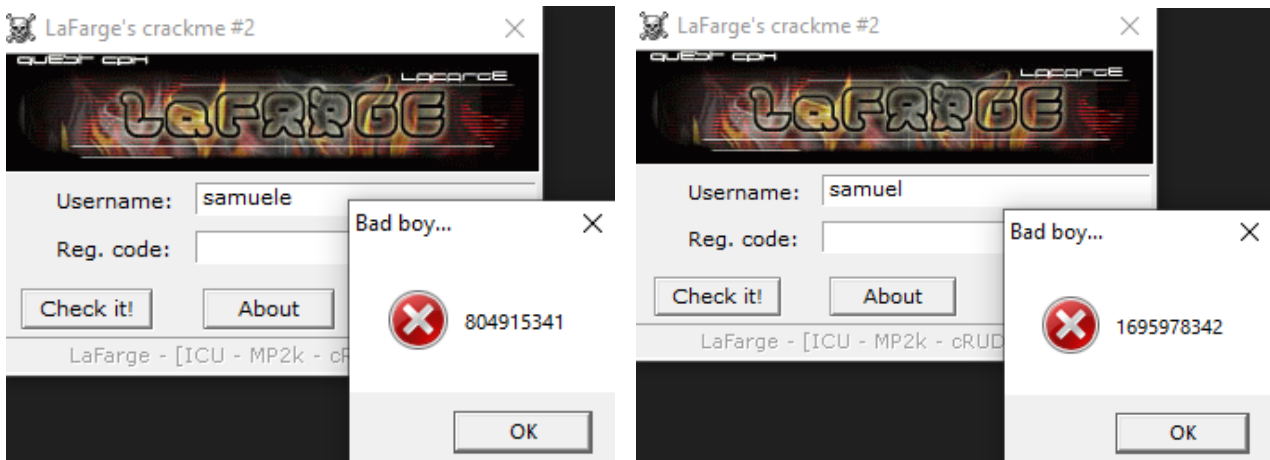
- nel registro 406549 viene memorizzata *correct_key* computata
- nel registro 4062E7 viene memorizzato il messaggio di errore visualizzato nella finestra di dialogo

x32dbg offre la possibilità di creare una copia di un eseguibile le cui istruzioni assembly sono state modificate. L'obiettivo è modificare il comportamento del software sfruttando la finestra di dialogo per far apparire a schermo il valore del registro in cui è contenuta la chiave computata. Per fare ciò, sarà sufficiente sostituire il registro contenente il messaggio di errore con il registro contenente la *correct_key*.

Salvando il nuovo eseguibile, otteniamo un software identico all'originale con l'unica differenza che il pulsante 'Check it!' produrrà una finestra di dialogo contenente la *correct_key* computata a partire da *nome_utente*. Una volta generata, sarà possibile utilizzare la chiave come input nel software originale.

00401285	48	dec eax	
00401286	75 F0	jne crackme.401278	
00401288	68 49674000	push crackme.406749	
0040128D	68 49654000	push crackme.406549	
00401292	E8 5F010000	call <JMP.&lstrncpy>	
00401297	68 00020000	push 200	
0040129C	68 49694000	push crackme.406949	
004012A1	6A 64	push 64	
004012A3	FF75 08	push dword ptr ss:[ebp+8]	
004012A6	E8 E5000000	call <JMP.&getDlgItemTextA>	
004012AB	68 49654000	push crackme.406549	
004012B0	68 49694000	push crackme.406949	
004012B5	E8 36010000	call <JMP.&lstrcmp>	
004012BA	0BC0	or eax,eax	
004012BC	75 16	jne crackme.4012D4	
004012BE	6A 40	push 40	
004012C0	68 DB624000	push crackme.4062DB	
004012C5	68 AC624000	push crackme.4062AC	
004012CA	FF75 08	push dword ptr ss:[ebp+8]	
004012CD	E8 CA000000	call <JMP.&MessageBoxA>	4062DB:"Good boy..."
004012D2	EB 14	jmp crackme.4012E8	4062AC:"Yep, thats the right code!\n\rGo write a keygen!"
004012D4	6A 10	push 10	
004012D6	68 06634000	push crackme.406306	
004012DB	68 49654000	push crackme.406549	406306:"Bad boy..."
004012E0	FF75 08	push dword ptr ss:[ebp+8]	
004012E3	E8 B4000000	call <JMP.&MessageBoxA>	
004012E8	68 00020000	push 200	
004012ED	68 49654000	push crackme.406549	
004012F2	E8 ED000000	call <JMP.&rtlZeroMemory>	
004012F7	68 00020000	push 200	
004012FC	68 49634000	push crackme.406349	
00401301	E8 DE000000	call <JMP.&rtlZeroMemory>	
00401306	68 00020000	push 200	
0040130B	68 49674000	push crackme.406749	
00401310	E8 CF000000	call <JMP.&rtlZeroMemory>	
00401315	B9 16000000	mov ecx,16	
0040131A	8D35 11634000	lea esi,dword ptr ds:[406311]	esi:"LdrpInitializeProcess"
00401320	8D3D 27634000	lea edi,dword ptr ds:[406327]	edi:"minkernel\\ntdll\\ldrinit.c"
00401326	F3A4	rep movsb	
00401328	5B	pop ebx	
00401329	58	pop eax	
0040132A	EB 38	jmp crackme.401364	
0040132C	663D EE03	cmp ax,3EE	
00401330	75 14	jne crackme.401346	
00401332	6A 00	push 0	
00401334	E8 08010000	call crackme.401441	
00401339	50	push eax	
0040133A	53	push ebx	
0040133B	6A 00	push 0	
0040133D	E8 72000000	call <JMP.&ExitProcess>	

Sostituzione registro caricato su stack



Generazione di diverse correct_key per diversi nome_utente