Galhacktic Trendsetters

CTF writeups

33C3 CTF – Ichnixwisse

Posted on January 4, 2017January 5, 2017 by sajninredoc

In this crypto challenge, we're given an implementation of a zero-knowledge proof system (https://en.wikipedia.org/wiki/Zero-knowledge_proof) for 3-coloring a graph (https://en.wikipedia.org/wiki/Graph_coloring). We're also given a graph on 100 vertices with 4600 edges which is far from 3-colorable. Our goal is to fool the verifier on the server and prove to it that this graph is in fact 3-colorable (where upon doing so, we receive the flag).

Let's begin by reviewing some preliminaries on zero-knowledge proofs. There is a classic zero-knowledge proof protocol for 3-coloring a graph, which works as follows:

1. The prover takes his 3-colored graph, and randomly permutes the colors with one of the $3! = 6$ possible permutations (e.g. changes all reds to blues, all blues to greens, and all greens to red).
2. The prover then uses a bit-commitment scheme (https://en.wikipedia.org/wiki/Commitment_scheme) to commit to this specific coloring of the vertices.
3. The verifier then chooses a random edge of the graph, and asks the prover to reveal the colors of the two vertices of this edge.
4. The prover then decommits and reveals these colors to the verifier.
5. If the colors are the same, (or the prover fails at decommitting), the verifier rejects the proof. Otherwise, the protocol goes back to step 1, for as many iterations as is required to convince the verifier.

It's not too hard to show that this protocol is in fact zero-knowledge, but that's orthogonal to this challenge so I won't describe it here (if you're interested, you can read a proof in these lecture notes (http://www.cs.cornell.edu/courses/cs6830/2011fa/scribes/lecture18.pdf)). One thing that is important to observe is that if the prover has an almost 3-coloring of the graph that works for a p fraction of the edges, then the prover can fool the verifier in any given round with probability at least p (i.e. if the verifier happens to choose a correctly colored edge). Another useful thing to observe is that you can always 3-color a graph so that at least $2/3$ of its edges are colored correctly; in fact, if you just assign colors randomly, $2/3$ of the edges will be colored correctly in expectation (with some work you can also get this to work deterministically, but this isn't at all necessary). Unfortunately, this challenge requires us to pass 300 rounds of verification, where with a success probability of $2/3$ we would only have a $(2/3)^{300} \approx 10^{-53}$ chance of passing all the rounds. We could hope that maybe we could find a better coloring of the graph, but there are actually too many edges in the graph to do much better (any 3-colorable graph on 100 vertices can have at most $3(100/3)^2 = 3333$ edges). We'll end up using these facts later, however.

For now, let's look at two other aspects of the protocol: the bit-commitment scheme used, and how the verifier's requests are generated. In this scheme, we commit to a coloring by publishing a Merkle signature (https://en.wikipedia.org/wiki/Merkle_signature_scheme) of the coloring. The value of the ith leaf in this Merkle tree is the SHA1 hash of the color of the ith vertex concatenated with a

random nonce. Beyond this, the specific details of how the Merkle signature scheme works aren't too important; I think this challenge would also work if instead the prover just published a signature of each vertex's color individually.

Finally, we come to a very interesting feature of this protocol – this is actually a non-interactive zero-knowledge proof (https://en.wikipedia.org/wiki/Non-interactive_zero-knowledge_proof). In particular, instead of getting the verifier's challenges from, say, an external server, the verifier's challenges are generated pseudorandomly from your output so far. That is, to generate the next challenge, the "verifier" simply hashes your proof so far, and takes this hash modulo the number of edges to figure out which edge to challenge. This lets us construct a proof completely offline, and simply submit it to the server all at once. One important consequence of this is they force us to generate all our commitments for all of the rounds before the verifier generates its queries. If they didn't, then we could keep on trying new commitments until we find one that passes the round, and proceed from there.

Okay, so how can we construct a false proof for this graph that will fool the verifier? One problem is that, as I've described it, this scheme is (as far as I know) completely secure. More analytically, there seem to be two possible ways you could hope to increase your probability of success per round. The first is to somehow figure out how to decommit to two different colors, so you can answer a larger range of challenges successfully. Unfortunately, it's not too hard to show that this requires finding a SHA1 collision, which is pretty hard (https://www.schneier.com/blog/archives/2015/10/sha-1_freestart.html). The second approach is to somehow abuse the pseudorandom number generator to force the verifier to challenge you with queries which you can actually answer correctly. The problem with this approach is that, as pointed out in the previous paragraph, you have to declare all your commitments before the challenges are verified. Once you declare your commitments, your responses (decommitments) are fixed unless you can find SHA1 collisions, so somehow you need to find a sequence of commitments that generates a "good" sequence of queries. I don't know if this is equivalent to finding a SHA1 collision, but it does involve somehow forcing this PRNG to behave very non-randomly, which also seems quite hard (you can imagine this, for example, as similar to finding some x where $SHA1(x+i)$ is even for all $i \in \{1, 2, \ldots, 300\}$).

So now we're stuck! But since this challenge is solvable, one of our assumptions about this protocol must be wrong. Unlike crypto challenges we were used to solving from other CTFs, here the vulnerability was hidden in a tiny implementation detail in the verifier code. In particular, even though the color could only be one of 1, 2, or 3, the commitment scheme allocates a 10-digit string for the color information — that is, each leaf in the Merkle tree is the SHA1 hash of C concatenated with N, where C is a 10-digit string representing the color (which should be one of "000000001", "0000000002", or "0000000003"), and N is a 16-byte nonce. Now, it might seem weird to allocate 10 decimal digits for a number which is at most 3, and in fact, this is where the vulnerability lies. In the verifier code (which is written in C), when the prover decommits and reveals a color, this 10-digit string is read into a regular 32-bit integer. But $10^{10} > 2^{32}$! So because of integer overflow, there are actually three values that the verifier will treat as the same color for each color (for example, "4294967297" will also be read as the color 1).

How can we use this to our advantage? Well, recall in our earlier analysis, we stated "Once you declare your commitments, your responses (decommitments) are fixed unless you can find SHA1 collisions". This is no longer true! In fact, as long as we can answer the current challenge successfully, we can answer it successfully in 9 different ways, since there are 3 values we can give for the color of the first vertex and 3 values we can give for the color of the second vertex. Also, note that each such decommitment leads to a different next query. The probability that one specific next query is answerable is something around $2/3$, so the probability that none of these next queries is answerable is at most $(1/3)^9 \approx 5 \cdot 10^{-5}$.

Now we're in really good shape. With 300 rounds, the probability we ever run into a situation where none of the next queries is answerable is around $1-(1-(1/3)^9)^{300} \approx 0.01$, or around 1%, and we can easily generate a valid "proof".

Here is some code which implements this approach (modified from the supplied prover code):

```
from Crypto.Random import random as srandom
from itertools import product
import argparse
import hashlib

OVERFLOW = 4294967296

class Graph:
    def __init__(self, n, edges):
        self.edges = edges
        self.n = n


def generate_proof(outstream, graph, color, rounds):
    rnd = srandom
    hash_func = hashlib.sha1
    h = hash_func()
    S = []

    def emit(s):
        h.update(s)
        S.append(s)
        outstream.write(s)

    def rand():
        return int(h.hexdigest(), 16)

    def encode_int(num):
        return str(num).rjust(10, '0').encode('ascii') + b'\n'

    def subtree_hash(leafs, i, j):
        if i == j:
            return leafs[i]
        mid = (i + j) // 2
        res = hash_func(subtree_hash(leafs, i, mid) + subtree_hash(
        return res

    def commit(values):
        ''' Generate a merkle tree over v_i + r_i with randomly chos
        r = [rnd.getrandbits(128).to_bytes(16, byteorder='big') for
        values = [v + r for v, r in zip(values, r)]
        leafs = [hash_func(v).digest() for v in values]
        commitment = subtree_hash(leafs, 0, len(leafs)-1)
        return r, commitment



    def reveal(values, randoms, index, actual_val):
        leafs = [hash_func(v + r).digest() for v, r in zip(values,

        emit(actual_val)
        emit(randoms[index])

        def dfs(i, j):
```

```
            if i == j: return
            mid = (i + j) // 2
            if index <= mid:
                sibling_hash = subtree_hash(leafs, mid + 1, j)
                emit(sibling_hash)
                dfs(i, mid)
            else:
                sibling_hash = subtree_hash(leafs, i, mid)
                emit(sibling_hash)
                dfs(mid+1, j)
        dfs(0, len(leafs)-1)

    # commit but each time it commits it doesn't emit anything
    def reveal_emittance(values, randoms, index, actual_val):
        leafs = [hash_func(v + r).digest() for v, r in zip(values, r

        def dfs(i, j):
            if i == j: return b''
            mid = (i + j) // 2
            if index <= mid:
                sibling_hash = subtree_hash(leafs, mid + 1, j)
                return sibling_hash + dfs(i, mid)
            else:
                sibling_hash = subtree_hash(leafs, i, mid)
                return sibling_hash + dfs(mid+1, j)
        return actual_val + randoms[index] + dfs(0, len(leafs)-1)

    emit(encode_int(rounds))

    randoms = []
    colors = []
    for round in range(rounds):
        print('Commitment for round %d' % round)
        perm = list(range(3))
        rnd.shuffle(perm)
        color_perm = [encode_int(perm[color[x]]) for x in range(1, q
        r, commitment = commit(color_perm)

        randoms.append(r)
        colors.append(color_perm)

        emit(commitment)



    for round, (r, c) in enumerate(zip(randoms, colors)):
        print('Reveal for round %d' % round)
        challenge = rand() % len(graph.edges)
        x, y = graph.edges[challenge]
        assert c[x-1] != c[y-1]

        cx = int(c[x-1])
        cy = int(c[y-1])
        print(c[x-1], cx)
        xvals = [cx, cx+OVERFLOW, cx+2*OVERFLOW]
        yvals = [cy, cy+OVERFLOW, cy+2*OVERFLOW]

        xvals = [encode_int(xval) for xval in xvals]
        yvals = [encode_int(yval) for yval in yvals]

        xv, yv = -1, -1
```

```
116             for xval, yval in product(xvals, yvals):
117                 xemit = reveal_emittance(c, r, x-1, xval)
118                 yemit = reveal_emittance(c, r, y-1, yval)
119 
120                 cur_digest = b''.join(S) + xemit + yemit
121 
122                 next_challenge = int(hash_func(cur_digest).hexdigest(),
123                 nx, ny = graph.edges[next_challenge]
124                 if c[nx-1] != c[ny-1]:
125                     xv = xval
126                     yv = yval
127                     break
128 
129             if xv == -1:
130                 print('No choice good -- abort!')
131                 return
132             else:
133                 reveal(c, r, x-1, xv)
134                 reveal(c, r, y-1, yv)
135 
136     if __name__ == '__main__':
137         parser = argparse.ArgumentParser()
138         parser.add_argument('-g', required=True, dest='graph_file', help
139         parser.add_argument('-c', required=True, dest='color_file', help
140         parser.add_argument('-p', required=True, dest='proof_file', help
141         parser.add_argument('-r', type=int, dest='rounds', default=300,
142         args = parser.parse_args()
143 
144         n = 0
145         edges = []
146         with open(args.graph_file) as f:
147             for line in f:
148                 line = line.strip()
149                 if line.startswith('#') or not line: continue
150                 if line.startswith('p'):
151                     n = int(line.split()[2])
152                 else:
153                     assert line.startswith('e')
154                     x, y = map(int, line.split()[1:])
155                     assert 1 <= x <= n and 1 <= y <= n
156                     edges.append((x, y))
157         graph = Graph(n, edges)
158 
159         color = {}
160         with open(args.color_file) as f:
161             for line in f:
162                 line = line.strip()
163                 if line.startswith('#') or not line: continue
164                 assert line.startswith('c')
165                 x, c = map(int, line.split()[1:])
166                 color[x] = c
167                 assert c in [0,1,2], "Invalid color: %d" % c
168         for x in range(1, graph.n+1):
169             assert x in color, "Node %d does not have a color" % x
170 
171         print('Generating proof...')
172         with open(args.proof_file, 'wb') as f:
173             generate_proof(f, graph, color, rounds=args.rounds)
```

Posted in 33C3 CTFTagged crypto