Firstly, to figure out the exit gate order, we noticed we're going right whenever we're getting a new set of hex numbers. All this while, we kept saving them and converted them to ASCII. They seemed to form a sentence: "You see a Gold-Bug in one corner. It is the key to a treasure found by" We thought this must be getting used somewhere ahead. Then, on pressing read we got our puzzle: You see the following written on the panel: n = 84364443735725034864402554533826279174703893439763343343863260342756678609216895093779263028809246505955647572176682669445270008816481771701417554768871285020442403001649254405058303439906229201909599348669565697534331652019516409514800265887388539283381053937433496994442146419682027649079704982600857517093*n*=84364443735725034864402554533826279174703893439763343343863260342756678609216895093779263028809246505955647572176682669445270008816481771701417554768871285020442403001649254405058303439906229201909599348669565697534331652019516409514800265887388539283381053937433496994442146419682027649079704982600857517093 Crewmates: This door has RSA encryption with exponent 55 and the password is: 2370178774682911039678909490731983030553818037642728322629590658530188954399653341053938177968436688097089627901880710053017665162508698865521085855413334590627256102779817144092314796016509489198045275785268570702028938469832266534760990574458224815724693200797833912963006702298796670695548259886980015169323701787746829110396789094907319830305538180376427283226295906585301889543996533410539381779684366880970896279018807100530176651625086988655210858554133345906272561027798171440923147960165094891980452757852685707020289384698322665347609905744582248157246932007978339129630067022987966706955482598869800151693 =c=*c* \large{\text{\underline{ANALYSIS}}}ANALYSIS​ It is given in the question that it is an RSA encryption with public exponent e=5*e*=5, Encrypted text c \equiv m^e \mod n*c*≡*me*mod*n* , Decrypted text m \equiv c^d \mod n*m*≡*cd*mod*n*, and d*d* being the private key satisfying ed \equiv 1 \mod \phi (n)*ed*≡1mod*ϕ*(*n*). We need to decrypt the password to clear the level for which we require d*d*. Since the given n*n* is very large we cannot find \phi (n)*ϕ*(*n*) and hence d*d* by factorization or any other such method efficiently. However, we can use small exponent attacks. We checked if n*n* was divisible by any of the known Mersenne primes, but it wasn't. Then we moved on to the \textbf{Coppersmith attack}**Coppersmith attack** which employs \textit{LLL}*LLL* and \textit{Minkowski's Theorem}*Minkowski’s Theorem* as taught in the lectures. First of all, we need to check whether or not the actual message consists of padding. If there is no padding then c^{\frac{1}{e}}*ce*1​ will be an integer , but it came out to be a non-integer. Hence, let's say the padding is p*p* then , ( p+m\_0 )^e \equiv c \mod n(*p*+*m*0​)*e*≡*c*mod*n*. \textit{Coppersmith's Theorem}:*Coppersmith’s Theorem*: Given N \in \Z*N*∈Z and f \in \Z[x] *f*∈Z[*x*] be an integral polynomial in x*x* of degree (say) d*d*, then we can find all x\_0 \in \Z*x*0​∈Z such that x\_0 < N^{\frac{1}{d}}*x*0​<*Nd*1​ and f(x\_0) \equiv 0 \mod N*f*(*x*0​)≡0mod*N* in reasonable time. Moreover, the complexity of one such algorithm (Coppersmith's algorithm) is less than the complexity of the LLL algorithm on a lattice of dimension D*D*, with the required D=\log\_2(N)*D*=log2​(*N*). Hence, in order to get m*m*, we can just find the roots of f(m\_0) = (p + m\_0)^e-c *f*(*m*0​)=(*p*+*m*0​)*e*−*c* in the congruence class of n*n*, using the aforementioned Coppersmith's algorithm and lattice reduction. However, we first have to guess the padding used and the length of the unknown message m\_0*m*0​. Clearly, the padding can't be all zeroes as in the previous assignments. We recalled that we had saved a sentence while solving the game of this puzzle. We quickly converted it back to ASCII codes and then to byte values. Call this p\_0*p*0​, then we have to form p=p\_0 \cdot 2^k*p*=*p*0​⋅2*k* for some k \in \N*k*∈N which we have to iterate upon. Note that this k*k* is the length of m\_0*m*0​ effectively, hence must be less than n^{\frac{1}{e}}*ne*1​. Also note that this algorithm will give an m\_0 < n^{\frac{1}{e}}*m*0​<*ne*1​ ( \approx 205≈205 bits) if it exists. Thus, we iterated over k*k* and checked all possible m\_0 < 2^k*m*0​<2*k* putting them into the RSA encryption function. This yielded the solution: m= 01011001 01101111 01110101 00100000 01110011 01100101 01100101*m*=01011001011011110111010100100000011100110110010101100101 00100000 01100001 00100000 01000111 01101111 01101100 0110010000100000011000010010000001000111011011110110110001100100 00101101 01000010 01110101 01100111 00100000 01101001 01101110 00101101010000100111010101100111001000000110100101101110 00100000 01101111 01101110 01100101 00100000 01100011 01101111 00100000011011110110111001100101001000000110001101101111 01110010 01101110 01100101 01110010 00101110 00100000 01001001 01110010011011100110010101110010001011100010000001001001 01110100 00100000 01101001 01110011 00100000 01110100 01101000 01110100001000000110100101110011001000000111010001101000 01100101 00100000 01101011 01100101 01111001 00100000 01110100 01100101001000000110101101100101011110010010000001110100 01101111 00100000 01100001 00100000 01110100 01110010 01100101 01101111001000000110000100100000011101000111001001100101 01100001 01110011 01110101 01110010 01100101 00100000 01100110 01100001011100110111010101110010011001010010000001100110 01101111 01110101 01101110 01100100 00100000 01100010 01111001 01101111011101010110111001100100001000000110001001111001 00100000 01000010 01000000 01101000 01110101 01100010 001000000100001001000000011010000111010101100010 01000001 01101100 00100001010000010110110000100001 which when converted to ASCII gives: M=\textbf{You see a Gold-Bug in one corner. It is the key to a treasure}*M*=**You see a Gold-Bug in one corner. It is the key to a treasure** \textbf{found by B@hubAl!}**found by B@hubAl!** The root was m\_0=\textbf{01000010 01000000 01101000 01110101}*m*0​=**01000010 01000000 01101000 01110101** \textbf{ 01100010 01000001 01101100 00100001}**01100010 01000001 01101100 00100001**. Hence, the password was M\_0=\textbf{B@hubAl!}*M*0​=**B@hubAl!**. For the above process involving Coppersmith's attack, theorem and algorithm, lattice reduction method, and for running the code as well, we used the following resources: https://en.wikipedia.org/wiki/Coppersmith\_method https://web.eecs.umich.edu/~cpeikert/lic13/lec04.pdf https://sagecell.sagemath.org/ https://en.wikipedia.org/wiki/Coppersmith%27s\_attack -------------------------------------------------------------------------------------------------------------------------------------- PS- Figuring out the exit gates was an adventure too :P - >exit2 61 20 47 6f 6c 64 2d 42 >exit1 59 6f 75 20 73 65 65 20 >exit3 59 6f 75 20 73 65 65 20 >exit4 same exit1,3,4 give same "\textbf{You see}**You see**" exit2 gives "\textbf{a Gold-B}**a Gold-B**" exit5 is closed exit2 then exit 4 gives 75 67 20 69 6e 20 6f 6e = \textbf{ug in on}**ug in on** then exit3 gives 65 20 63 6f 72 6e 65 72 = \textbf{e corner}**e corner** then exit1 gives 2e 20 49 74 20 69 73 20 = \textbf{. It is}**. It is** then exit4 gives 74 68 65 20 6b 65 79 20 = \textbf{the key}**the key** again exit4 gives 74 6f 20 61 20 74 72 65 = \textbf{to a tre}**to a tre** exit2 gives 61 73 75 72 65 20 66 6f = \textbf{asure fo}**asure fo** exit2 gives 75 6e 64 20 62 79 = \textbf{und by}**und by** exit1 gives blank read gives You see the following written on the panel: n = .........

Answer Password: B@hubAl!